Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection

Zu-Ming Jiang and Jia-Ju Bai, Tsinghua University; Kangjie Lu, University of Minnesota; Shi-Min Hu, Tsinghua University

https://www.usenix.org/conference/usenixsecurity20/presentation/jiang
Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection

Zu-Ming Jiang, Jia-Ju Bai
Tsinghua University

Kangjie Lu
University of Minnesota

Shi-Min Hu
Tsinghua University

Abstract
Error handling code is often critical but difficult to test in reality. As a result, many hard-to-find bugs exist in error handling code and may cause serious security problems once triggered. Fuzzing has become a widely used technique for finding software bugs nowadays. Fuzzing approaches mutate and/or generate various inputs to cover infrequently-executed code. However, existing fuzzing approaches are very limited in testing error handling code, because some of this code can be only triggered by occasional errors (such as insufficient memory and network-connection failures), but not specific inputs. Therefore, existing fuzzing approaches in general cannot effectively test such error handling code.

In this paper, we propose a new fuzzing framework named FIFUZZ, to effectively test error handling code and detect bugs. The core of FIFUZZ is a context-sensitive software fault injection (SFI) approach, which can effectively cover error handling code in different calling contexts to find deep bugs hidden in error handling code with complicated contexts. We have implemented FIFUZZ and evaluated it on 9 widely-used C programs. It reports 317 alerts which are caused by 50 unique bugs in terms of the root causes. 32 of these bugs have been confirmed by related developers. We also compare FIFUZZ to existing fuzzing tools (including AFL, AFLFast, AFLSmart and FairFuzz), and find that FIFUZZ finds many bugs missed by these tools. We believe that FIFUZZ can effectively augment existing fuzzing approaches to find many real bugs that have been otherwise missed.

1 Introduction
A program may encounter various errors and needs to handle these errors at runtime. Otherwise, the program may suffer from security or reliability issues. While error handling is critical, itself is error-prone. Firstly, error handling code is difficult to correctly implement [14, 23, 34, 54] because such code is infrequently executed and often receives insufficient attention. For these reasons, many bugs may exist in error handling code, and they are often difficult to find in real execution. Some recent works [8, 32, 37, 68] have shown that many bugs in error handling code can cause serious security problems, such as denial of service (DoS) and information disclosure. In fact, many CVE-assigned vulnerabilities (such as CVE-2019-7846 [19], CVE-2019-2240 [20], CVE-2019-1750 [21] and CVE-2019-1785 [22]) stem from bugs in error handling code.

Considering that error handling code is critical but buggy, various tools have been proposed to detect bugs in error handling code. Some approaches [28, 32, 33, 37, 53] use static analysis, but they often introduce many false positives, due to the lack of runtime information and inherent limitations with static analysis. To reduce false positives, recent approaches [1, 6, 7, 13, 15, 26, 27, 29, 30, 38, 45, 50, 51, 59, 60, 64, 65] instead use fuzzing to test infrequently executed code. They generate effective program inputs to cover infrequently executed code, according to the input specification or the feedback of program execution. However, the input-driven fuzzing cannot effectively cover error handling code, as some of this code can be only triggered by non-input occasional errors, such as insufficient memory and network-connection failures. As a result, existing fuzzing approaches cannot effectively test error handling code.

Testing error handling code is challenging by nature, as errors are often hard to deterministically produce. An intuitive solution to triggering error handling code is to use software fault injection (SFI) [52]. SFI 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. Specifically, the faults are injected into the sites that can fail and trigger error handling code, and we call each such site an error site. In this way, SFI can intentionally cover error handling code at runtime. Existing SFI-based approaches [9–11, 18, 25, 39, 40, 55, 67] have shown encouraging results in testing error handling code and detecting hard-to-find bugs.
However, existing SFI-based approaches suffer from a critical limitation: to our knowledge, they perform only context-insensitive fault injection, which often stops testing from going deep. Specifically, they inject faults according to the locations of error sites in source code, without considering the execution contexts of these error sites, i.e., the execution paths reaching to the error sites. Thus, if a fault is constantly injected into an error site, this error site will always fail when being executed at runtime. However, an error site is typically executed in different calling contexts, and real bugs can be only triggered when this error site fails in a specific calling context but succeeds in other calling contexts. In this case, existing SFI-based approaches may miss these real bugs.

Figure 1 shows a simple example of this case. In the function main, the objects x and y are allocated, and then the functions FuncA and FuncB are called. FuncA and FuncB both call FuncP, but FuncB frees the argument object before calling FuncP. In FuncP, the object z is allocated by calling malloc; if this function call fails, the argument object is freed, and the program exits abnormally by calling exit. If we perform context-insensitive fault injection by just statically injecting a fault into malloc in FuncP, the program will always exit when FuncB is executed, without finding any bug. If we consider calling context, and inject a fault into malloc in FuncP only when FuncB calls FuncP, a double-free bug of the object y can be triggered at runtime. Since such a case is fairly common, it may incur a significant impact on detecting bugs in error handling code.

```c
int main() {
    x = malloc(...);
    y = malloc(...);
    .......
    FuncA(x);
    FuncB(y);
    .......
}

void FuncA(x) {
    FuncP(x);
    .......
}

void FuncB(y) {
    free(y);
    FuncP(y);
    .......
}

void FuncP(arg) {
    z = malloc(....)
    if (!z) {
        free(arg);
        exit(-1);
    }
    .......
}
```

**Fault 1:** main -> FuncA -> FuncP -> malloc  exit abnormally...
**Fault 2:** main -> FuncB -> FuncP -> malloc  double free!

Figure 1: Examples of function calls that can fail.

In this paper, to effectively detect bugs in error handling code, we design a novel context-sensitive SFI-based fuzzing approach. The approach takes execution contexts into account to effectively guide SFI to maximize bug finding. It consists of six steps: 1) statically identifying the error sites in the source code of the tested program; 2) running the tested program and collecting runtime information about calling contexts of each executed error site and code coverage; 3) creating error sequences about executed error sites according to runtime information, and each element of such a sequence is differentiated by the location of the executed error site and the information about its calling context; 4) after running the program, mutating each created error sequence to generate new sequences; 5) running the tested program and injecting faults according to the mutated error sequences; 6) collecting runtime information, creating new error sequences and performing mutation of these error sequences again, which constructs a fuzzing loop.

Based on our approach, we propose a new fuzzing framework named FIFUZZ. At compile time, to reduce manual work of identifying error sites, FIFUZZ 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, FIFUZZ uses our context-sensitive SFI-based fuzzing approach in runtime testing. To be compatible with traditional fuzzing process for program inputs, FIFUZZ mutates the error sequences and program inputs together by analyzing runtime information of the tested program.

Overall, we make the following technical contributions:

- We perform two studies of error handling code in widely-used applications and vulnerabilities found by existing fuzzing tools, and find that: nearly 42% of sites that can trigger error handling code are related to occasional errors, but only few vulnerabilities found by existing fuzzing tools are related to error handling code triggered by occasional errors. Thus, it is important to improve fuzzing to support the testing of error handling code.

- We propose a novel context-sensitive SFI-based fuzzing approach, which can dynamically inject faults based on both locations of error sites and their calling contexts, to cover hard-to-trigger error handling code.

- Based on this approach, we develop a new fuzzing framework named FIFUZZ, to effectively test error handling code. To our knowledge, FIFUZZ is the first systematic fuzzing framework that can test error handling code in different calling contexts.

- We evaluate FIFUZZ on 9 well-tested and widely-used C applications of the latest versions as of our evaluation. It reports 317 alerts which are caused by 50 unique bugs in terms of the root causes. 32 of these bugs have been confirmed by related developers. We also compare FIFUZZ to existing fuzzing tools (including AFL, AFLFast, AFLSmart and FairFuzz) on 5 common programs in the Binutils toolset, and find that FIFUZZ finds many bugs missed by these tools.

The rest of this paper is organized as follows. Section 2 introduces background and our two studies. Section 3 introduces basic idea and our context-sensitive SFI-based fuzzing approach. Section 4 introduces FIFUZZ in detail. Section 5 shows our evaluation. Section 6 makes a discussion about FIFUZZ and its found bugs. Section 7 presents related work, and Section 8 concludes this paper.
2 Background

In this section, we first introduce error handling code with related bug examples, and then show our studies of error handling code in widely-used applications and CVEs found by existing fuzzing tools.

2.1 Error Handling Code

A program may encounter exceptional situations at runtime, due to special execution conditions such as invalid inputs from users, 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 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. While this error occurs occasionally, they can be reliably triggered in an adversarial setting. For example, by exhaustively consuming memory, an attacker can reliably result a function call to malloc() in returning a null pointer. As such, bugs in error handling code can be as critical as the ones in normal code.

2.2 Bug Examples in Error Handling Code

Figures 2 and 3 show two patches fixing bugs in error handling code of the libav library in ffmpeg [24]. In Figure 2, the variable sbr->sample_rate could be zero, but it is divided in the code, causing a divide-by-zero bug. This bug is also reported as CVE-2016-7499 [48]. To fix this bug, Patch A [46] checks whether sbr->sample_rate is zero before this variable is divided, and returns abnormally if so. The report of this bug [47] mentions that this bug was found by AFL. On the other hand, in Figure 3, the function av_frame_new_side_data is used to allocate memory for new data, and it can fail and return a null pointer when memory is insufficient. In this case, the variable dst->side_data[i]->metadata is freed after dst->side_data[i] is freed, which causes a use-after-free bug. To fix this bug, Patch B [49] frees the variable dst->side_data[i]->metadata before freeing dst->side_data[i]. Because the report of this bug or the patch does not mention any tool, the bug might be found by manual inspection or real execution.

The bug in Figure 2 is caused by missing handling of an input-related error, because the variable sbr->sample_rate is related to the function argument sbr affected by inputs.

2.3 Study of Error Handling Code

To understand the proportion of input-related errors and occasional errors that can trigger error handling code in software, we perform a manual study of the source files (.c and .h) of 9 widely-used applications (vim, bison, ffmpeg, nasm, catdoc, clamav, cflow, gif2png+libpng, and openssl). Due to time constraints, if an application contains over 100 source files, we randomly select 100 source files of this application to study. Otherwise, we study all the source files of this application. Specifically, we first manually identify the sites that can fail and trigger error handling code by looking for if or goto statements, which are often used as entries of error handling code in C applications [33]. Then, we manually check whether the identified sites are related to input-related errors or occasional errors. Table 1 shows the study results.

We find that 42% of the sites that can fail and trigger error handling code are related to occasional errors. Besides, in the study, we also observe that about 70% of the identified error sites are related to checking error-indicating return values of function calls (such as the example in Figure 3). This observation indicates that manipulating the return values of specific function calls can cover most error handling code, which has been adopted by some existing SFI-based approaches [10,18].
To effectively test error handling code, we introduce SFI in fuzz testing by “fuzzing” injected faults according to the runtime information of the tested program. To achieve this idea, we build an error sequence that contains multiple error points. An error point represents an execution point where an error can occur and trigger error handling code. When performing fault injection, each error point in an error sequence can normally run (indicated as 0) or fail by injecting a fault (indicated as 1). Thus, an error sequence is actually a 0-1 sequence that describes the failure situation of error points at runtime:

\[ ErrSeq = [ErrPt_1, ErrPt_2, ..., ErrPt_x], \quad ErrPt_i = \{0, 1\} \quad (1) \]

Similar to program inputs, an error sequence also affects program execution. This sequence can be regarded as the “input” of possibly triggered errors. A key problem here is which error points in an error sequence should be injected with faults to cover as much error handling code as possible. Inspired by existing fuzzing that fuzz program inputs using the feedback of program execution, our basic idea is to fuzz error sequence for fault injection to test error handling code.

### 3 Basic Idea and Approach

#### 3.1 Basic Idea

To effectively test error handling code, we introduce SFI in fuzz testing by “fuzzing” injected faults according to the runtime information of the tested program. To achieve this idea, we build an error sequence that contains multiple error points. An error point represents an execution point where an error can occur and trigger error handling code. When performing fault injection, each error point in an error sequence can normally run (indicated as 0) or fail by injecting a fault (indicated as 1). Thus, an error sequence is actually a 0-1 sequence that describes the failure situation of error points at runtime:

\[ ErrSeq = [ErrPt_1, ErrPt_2, ..., ErrPt_x], \quad ErrPt_i = \{0, 1\} \quad (1) \]

To describe 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:

\[ CallCtx = [CallInfo_1, CallInfo_2, ..., CallInfo_x] \quad (3) \]

Based on the above description, the information about each error point can be hashed as a key, and whether this error point should fail can be represented as a 0-1 value. Thus, an error sequence can be stored as a key-value pair in a hash table:
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 performing fault injection using error sequences, the faults should be injected into error points during program execution.

According to our method, when an error site is executed in $N$ different calling contexts, there will be $N$ different error points for fault injection, instead of just one error point identified by context-insensitive fault injection. Thus, our method can perform finer-grained fault injection.

### 3.3 Context-Sensitive SFI-based Fuzzing

To effectively cover as much error handling code as possible, based on our context-sensitive SFI method, we propose a novel context-sensitive SFI-based fuzzing approach to perform fault injection using the feedback of program execution.

As shown in Figure 4, our approach has six steps: 1) statically identifying the error sites in the source code of the tested program; 2) running the tested program and collecting runtime information about calling contexts of each executed error site and code coverage; 3) creating error sequences about executed error sites according to runtime information; 4) after running the program, mutating each created error sequence to generate new sequences; 5) running the tested program and injecting faults into error sites in specific calling contexts according to the mutated error sequences; 6) collecting runtime information, creating new error sequences and performing mutation of these error sequences again, which constructs a fuzzing loop. When no new error sequences are generated or the time limit is reached, the fuzzing loop ends.

Figure 5: Example of the initial mutation.

Subsequent generation and mutation. After executing the tested program by injecting faults according to an original error sequence, some new error points may be executed, making a new error sequence created. Our approach checks whether the code coverage is increased (namely new basic blocks or code branches are covered) during this execution. If not, the original error sequence and the created error sequence (if it exists) are dropped; if so, our approach separately mutates the original error sequence and the created error sequence (if it exists) to generate each new error sequence by changing the value of just one error point ($0 \rightarrow 1$ or $1 \rightarrow 0$). Then, our approach compares these generated error sequences with existing error sequences, to drop repeated ones. Figure 6 shows an example of this procedure for two error sequences. For the first error sequence $ErrSeq_1$, a new error point $ErrPt_1$ is executed, and thus our approach creates an error sequence containing $ErrPt_1$. As the code coverage is increased, our approach mutates the two error sequences and generates nine new error sequences. However, one of them is the same with existing error sequence $ErrSeq_2$, thus this new error sequence is dropped. For the second error sequence $ErrSeq_2$, a new error point $ErrPt_2$ is executed, and thus our approach creates an error sequence containing $ErrPt_2$. As the code coverage is not increased, our approach drops the two error sequences.

Note that each error point in an error sequence is related to runtime calling context, 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.
4 FIFUZZ Framework

Based on our context-sensitive SFI-based fuzzing approach, we design a new fuzzing framework named FIFUZZ, to effectively test error handling code. We have implemented FIFUZZ using Clang [16]. FIFUZZ performs code analysis and code instrumentation on the LLVM bytecode of the tested program. To be compatible with traditional fuzzing process, FIFUZZ mutates the error sequences and program inputs together. Figure 7 shows its architecture, consisting of six parts:

- **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 performs code instrumentation on the program code, including identified 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 generated inputs, collects runtime information of the tested program, and performs fault injection according to generated error sequences.

- **Error-sequence generator.** It creates error sequences, and mutates error sequences to generate new error sequences, according to collected runtime information.

- **Input generator.** It performs traditional fuzzing process to mutate and generate new inputs, according to collected runtime information.

- **Bug checkers.** They check the collected runtime information to detect bugs and generate bug reports.

Based on the above architecture, FIFUZZ consists of two phases, which are introduced as follows:

4.1 Compile-Time Analysis

In this phase, FIFUZZ 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 [18, 40, 55] require the user to manually provide error sites, which requires much manual work and cannot scale to large programs. To reduce manual work, the error-site extractor uses a static analysis against the source code of the tested program, to identify possible error sites, from which the user can select realistic ones.

Our analysis focuses on extracting specific function calls as error sites, because our study in Section 2.3 reveals that most of error sites are related to 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_frame_new_side_data in Figure 3 is an example that satisfies the two requirements.

S2: Selecting library functions. A called function can be defined in the tested program or an external library. In most cases, a function defined in the tested program can fail, as it calls specific library functions that can fail. If this function and its called library functions are both considered for fault injection, repeated faults may be injected. To avoid repetition, from all the identified function calls, our analysis only selects those whose called functions are library functions.

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 use 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 $R$, this method identifies this function as an error function. Finally, this method extracts all function calls to this function are identified error sites. For accuracy and generality, if there are multiple tested programs, this method analyzes the source code of all the tested programs together.

In our analysis, the value of the threshold $R$ in the third step heavily affects the identified error functions and identified error sites (function calls). For example, less error functions and error sites can be identified, as $R$ becomes larger. In this case, more unrealistic error functions and error sites can be dropped, but more realistic ones may be also missed. We study the impact of the value of $R$ in Section 5.2.

![Code instrumentation](image)

**Figure 8: Example of code instrumentation.**

**Code instrumentation.** The code instrumentation serves for two purposes: collecting runtime information about error sites and injecting faults. To collect the information about 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 in each function definition. Besides, on the other hand, to monitor the execution of error sites and perform fault injection into them, the program generator instruments code before each error site. During program execution, the runtime calling context of this error site and its location are collected to create an error point. Then, if this error point can be found in the current error sequence, and its value is 1 (indicating this error point should fail for fault injection) a fault is injected into the error point. In this case, the function call of related error site is not executed, and its return values is assigned to a null pointer or a random negative integer. If the value of this error point in the error sequence is 0, the function call of related error site is normally executed. Figure 8 shows an example of instrumented code in the C code. Note that code instrumentation is actually performed on the LLVM bytecode.

### 4.2 Runtime Fuzzing

In this phase, with the identified error sites and instrumented code, FIFUZZ performs our context-sensitive SFI-based fuzzing approach, and uses traditional fuzzing process of program inputs referring to AFL [1].

The runtime fuzzer executes the tested program using the program inputs generated by traditional fuzzing process, and injects faults into the program using the error sequences generated by our SFI-based fuzzing approach. It also collects runtime information about executed error points, code branches, etc. According to the collected runtime information, the error-sequence generator creates error sequences and performs mutation to generate new error sequences; the input generator performs coverage-guided mutation to generate new inputs. Then, FIFUZZ combines these generated error sequences and inputs together, and use them in runtime fuzzer to execute the tested program again. To detect bugs, the bug checkers analyze the collected runtime information. These bug checkers can be third-party sanitizers, such as ASan [4] and MSan [41].

### 5 Evaluation

#### 5.1 Experimental Setup

To validate the effectiveness of FIFUZZ, we evaluate it on 9 extensively-tested and widely-used C applications of the latest versions as of our evaluation. These applications are used for different purposes, such as text editor (`vim`), media processing (`ffmpeg`), virus scan (`clamav`) and so on. The information of these applications are listed in Table 3 (the lines of source code are counted by CLOC [17]). The experiment runs on a regular desktop with eight Intel i7-3770@3.40G processors and 16GB physical memory. The used compiler is Clang 6.0 [16], and the operating system is Ubuntu 18.04.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Version</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim</td>
<td>Text editor</td>
<td>v8.1.1764</td>
<td>349K</td>
</tr>
<tr>
<td>bison</td>
<td>Parser generator</td>
<td>v3.4</td>
<td>82K</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>Solution for media processing</td>
<td>n4.3-dev</td>
<td>1.1M</td>
</tr>
<tr>
<td>nasm</td>
<td>80x86 and x86-64 assembler</td>
<td>v2.14.02</td>
<td>94K</td>
</tr>
<tr>
<td>catdoc</td>
<td>MS Word file viewer</td>
<td>v0.95</td>
<td>4K</td>
</tr>
<tr>
<td>clamav</td>
<td>Antivirus engine</td>
<td>v0.101.2</td>
<td>844K</td>
</tr>
<tr>
<td>cflow</td>
<td>Code analyzer of C source files</td>
<td>v1.6</td>
<td>37K</td>
</tr>
<tr>
<td>gif2png+libpng</td>
<td>File converter for pictures</td>
<td>v2.5.14+v1.6.37</td>
<td>59K</td>
</tr>
<tr>
<td>openssl</td>
<td>Cryptography library</td>
<td>v1.1.1d</td>
<td>415K</td>
</tr>
</tbody>
</table>

**Table 3: Basic information of the tested applications.**
The above results are obtained with R = 0.6. To understand the variation caused by R, we test R from 0.5 to 1 with 0.05 step. Figure 9 shows the results. 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

Using the 1,822 realistic error sites identified with R = 0.6, we test the 9 target applications. We fuzz each application with a well-know sanitizer ASan [4] and then without ASan (because it often introduces much runtime overhead), for three times. The time limit of each fuzzing is 24 hours. For the alerts found by fault injection, we count them by trigger location and error point (not error site). Table 5 shows the fuzzing results with ASan and without ASan. The columns “Error sequence” and “Input” show the results about generated error sequences and inputs; in these columns, the columns “Gen” show the number of generated ones, and the columns “Useful” show the number of ones that increase code coverage. From the results, we find that:

**Error sequence.** FIFUZZ generates many useful error sequences for fault injection to cover error handling code. In total, 3% and 2% of generated error sequences increase code coverage by covering new code branches, with and without ASan, respectively. These proportions are larger than those (0.02% with ASan and 0.007% without ASan) for generated program inputs. To know about the variation of useful error sequences and program inputs increasing code coverage, we select vim as an example to study. Figure 10 shows the results. We find that the number of useful 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Function call</th>
<th>Identified</th>
<th>Realistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim</td>
<td>67,768</td>
<td>1,589 (2.3%)</td>
<td>283 (17.8%)</td>
</tr>
<tr>
<td>bison</td>
<td>11,861</td>
<td>966 (8.1%)</td>
<td>143 (12.0%)</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>439,986</td>
<td>2,157 (0.5%)</td>
<td>190 (8.8%)</td>
</tr>
<tr>
<td>harun</td>
<td>10,246</td>
<td>4,29 (4.2%)</td>
<td>44 (10.3%)</td>
</tr>
<tr>
<td>clamav</td>
<td>1,293</td>
<td>103 (8.0%)</td>
<td>45 (43.7%)</td>
</tr>
<tr>
<td>flow</td>
<td>52,830</td>
<td>2,183 (4.1%)</td>
<td>816 (37.4%)</td>
</tr>
<tr>
<td>gifs</td>
<td>4,049</td>
<td>149 (3.7%)</td>
<td>54 (17.8%)</td>
</tr>
<tr>
<td>openssl</td>
<td>2,183</td>
<td>191 (0.5%)</td>
<td>45 (43.7%)</td>
</tr>
<tr>
<td>nasm</td>
<td>303</td>
<td>303 (2.7%)</td>
<td>54 (17.8%)</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>4,049</td>
<td>149 (3.7%)</td>
<td>54 (17.8%)</td>
</tr>
<tr>
<td>bison</td>
<td>10,246</td>
<td>4,29 (4.2%)</td>
<td>44 (10.3%)</td>
</tr>
<tr>
<td>vim</td>
<td>67,768</td>
<td>1,589 (2.3%)</td>
<td>283 (17.8%)</td>
</tr>
<tr>
<td>total</td>
<td>777,867</td>
<td>9,795 (1.3%)</td>
<td>1,822 (18.6%)</td>
</tr>
</tbody>
</table>

Table 4: Results of error-site extraction.
### Table 5: Fuzzing results.

<table>
<thead>
<tr>
<th>Program</th>
<th>With ASan</th>
<th>Error sequence</th>
<th>Input</th>
<th>Reported alert</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen</td>
<td>Useful</td>
<td>Gen</td>
<td>Useful</td>
</tr>
<tr>
<td>vim</td>
<td>9,399</td>
<td>872</td>
<td>504,736</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>1,450</td>
<td>221</td>
<td>1,995,831</td>
<td>1,108</td>
</tr>
<tr>
<td>bison</td>
<td>391</td>
<td>11</td>
<td>139,286</td>
<td>758</td>
</tr>
<tr>
<td></td>
<td>4,203</td>
<td>772</td>
<td>4,551,244</td>
<td>0</td>
</tr>
<tr>
<td>cflow</td>
<td>84</td>
<td>4</td>
<td>4,721,301</td>
<td>158</td>
</tr>
<tr>
<td>clamav</td>
<td>482</td>
<td>139</td>
<td>98,352</td>
<td>26</td>
</tr>
<tr>
<td>cow</td>
<td>1,623</td>
<td>159</td>
<td>4,551,244</td>
<td>724</td>
</tr>
<tr>
<td>cc2png+libpng</td>
<td>71</td>
<td>9</td>
<td>15,018,120</td>
<td>330</td>
</tr>
<tr>
<td>openssl</td>
<td>73,200</td>
<td>626</td>
<td>369,613</td>
<td>13</td>
</tr>
<tr>
<td>Total</td>
<td>92,126</td>
<td>2,336</td>
<td>29,971,722</td>
<td>6,223</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Without ASan</th>
<th>Error sequence</th>
<th>Input</th>
<th>Reported alert</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen</td>
<td>Useful</td>
<td>Gen</td>
<td>Useful</td>
</tr>
<tr>
<td>vim</td>
<td>44,322</td>
<td>1,264</td>
<td>2,315,965</td>
<td>351</td>
</tr>
<tr>
<td></td>
<td>8,692</td>
<td>289</td>
<td>14,602,760</td>
<td>1,207</td>
</tr>
<tr>
<td>bison</td>
<td>306</td>
<td>5</td>
<td>4,811,284</td>
<td>1,766</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>451</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cflow</td>
<td>58</td>
<td>289</td>
<td>11,325,615</td>
<td>4,202</td>
</tr>
<tr>
<td>clamav</td>
<td>482</td>
<td>325</td>
<td>331,930</td>
<td>34</td>
</tr>
<tr>
<td>cow</td>
<td>12,209</td>
<td>217</td>
<td>29,996,026</td>
<td>1,235</td>
</tr>
<tr>
<td>cc2png+libpng</td>
<td>1,408</td>
<td>6</td>
<td>27,117,556</td>
<td>409</td>
</tr>
<tr>
<td>openssl</td>
<td>82,214</td>
<td>671</td>
<td>428,447</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>191,942</td>
<td>3,804</td>
<td>139,847,650</td>
<td>9,558</td>
</tr>
</tbody>
</table>

### Table 6: Summary of reported alerts and found bugs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Null</th>
<th>MemErr</th>
<th>Assert</th>
<th>All</th>
<th>Error handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique alert</td>
<td>152</td>
<td>132</td>
<td>3</td>
<td>317/313</td>
<td></td>
</tr>
<tr>
<td>Found bug</td>
<td>36</td>
<td>13</td>
<td>1</td>
<td>50/46</td>
<td></td>
</tr>
<tr>
<td>Confirmed bug</td>
<td>26</td>
<td>6</td>
<td>0</td>
<td>32/32</td>
<td></td>
</tr>
</tbody>
</table>

**Reported alerts.** With ASan, FIFUZZ reports 255 alerts, including 126 null-pointer dereferences, 126 memory errors (such as use-after-free and buffer-overflow alerts) and 3 assertion failures. Among these alerts, 114 are reported by ASan, and 82 are found due to causing crashes. Without ASan, FIFUZZ reports 298 alerts, including 178 null-pointer dereferences, 117 memory errors and 3 assertion failures. All these alerts are found due to causing crashes. Indeed, ASan can find memory errors that do not cause crashes. Thus, with ASan, FIFUZZ finds more memory errors. However, due to monitoring memory accesses, ASan often introduces over 2x runtime overhead [5]. Thus, with ASan, FIFUZZ executes less test cases within given time and some null-pointer dereferences causing crashes are missed.

**Alert summary.** In Table 6, we summarize the alerts found by FIFUZZ with and without ASan, and identify 317 unique alerts, including 182 null-pointer dereferences, 132 memory errors and 3 assertion failures. 313 of them are related to incorrect error handling caused by occasional errors, and only 4 alerts are caused by program inputs. Section Appendix shows 50 randomly-selected alerts.

**Found bugs.** In Table 6, we check the root causes of the 317 reported alerts, and identify 50 new and unique bugs in terms of their root causes. Specifically, 313 alerts are related to incorrect error handling, which are caused by 46 bugs. The remaining 4 alerts are caused by four bugs that are not in error handling code. We have reported all these bugs to related developers. 32 of them have been confirmed, and we are still waiting for the response of remaining ones.

**Error handling bugs.** The 46 found bugs related to incorrect error handling are caused by only 18 error sites but in different calling contexts. Most of the error sites are related to occasional errors of memory allocation. Figure 11 shows such examples of four bugs found in bison, and these bugs have different root causes according to our manual checking. Additionally, the developer fixes each of these bugs by adding 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 four bugs are all caused by the failures of the function call to `callloc in hash_initialize`, but the failures occur in different calling contexts. Besides, the call stacks of Bug 3 and Bug 4 are the same except for the different calls to `hash_initialize` in symbols_new. If a fault is injected into the call to `callloc without considering calling context, Bug 3 can be found, but Bug 4 will be missed. The results confirm the advantages of context-sensitive SFI over traditional context-insensitive one.

**Bug features.** Reviewing the bugs found by FIFUZZ, we find two interesting features. Firstly, among the 46 found bugs related to incorrect error handling, only 4 are triggered by two or more error points’ failures, and the remaining 42 bugs are triggered by only one error point’s failure. The results indicate that error-handling bugs are often triggered by just one error. Secondly, most of found bugs are caused by the case that an error is correctly handled in the function containing related error site but incorrectly handled in this function’s ancestors in the call stack. For example in Figure 11, the failure of the function call to `callloc is correctly handled in hash_initialize, and hash_initialize returns a null pointer. In this case, the functions calling `hash_initialize make some global variables become NULL, but these global variables are still dereferenced in subsequent execution. Indeed, developers can often implement correct error handling code in current functions, but often make mistakes in error propagation due to complex calling contexts of error sites.

---

**Figure 11:** Example bugs caused by the same error site.
null pointer dereference & 36 & 0 & 0 & 0 & 0 \\
double free & 0 & 0 & 0 & 0 & 0 \\
free after free & 0 & 0 & 0 & 0 & 0 \\
buffer overflow & 0 & 0 & 0 & 0 & 0 \\
free invalid pointer & 0 & 0 & 0 & 0 & 0 \\
assertion failure & 0 & 0 & 0 & 0 & 0 \\
total & 39 & 6 & 2 & 3 & 0 \\

Table 7: Security impact classified by bug type.

![Figure 12: Two-use-after-free bugs found in clamav.](image)

5.4 Security Impact of Found Bugs

We manually review the 50 found bugs to estimate their security impact. The results are shown in Table 7, classified by bug type, including double-free, use-after-free, buffer-overflow and free-invalid-pointer bugs. The results show that many found bugs can cause serious security problems, such as memory corruption and arbitrary read.

Figure 12 shows two use-after-free bugs reported in clamav. When the program starts, the function cli_ac_addsig is executed, and it calls cli_ac_adddatt that can fail and trigger error handling code. In this code, mpool_free is called to free the pointer new. When the program exits, the function cli_ac_free is called, and it executes a loop to handle each element patt in the pointer array root->ac_pattable. When i is a specific value, patt is an alias of new which has been freed in cli_ac_addsig, and then patt is used to access patt->virname (a pointer) and patt->special (a condition variable), causing two use-after-free bugs. Once these bugs are triggered, the attacker can exploit them to control the values of patt->virname and patt->special, and thus to corrupt memory and switch the control flow between the branches of the if statement in line 581.

5.5 Comparison to Context-Insensitive SFI

In FIFUZZ, our context-sensitive SFI method is an important technique of covering error handling code in different calling contexts. To show the value of this technique, we modify FIFUZZ by replacing it with a context-insensitive SFI method, which builds error sequences using error sites, without considering their calling contexts. We evaluate the resulting tool on the 9 tested applications in Table 3, without using any sanitizer. Each application is also tested for three times, and the time limit of each testing is 24 hours. Table 8 shows the results of the resulting tool (FIFUZZ_insensitive) and FIFUZZ.

Compared to FIFUZZ, the resulting tool generates less useful error sequences that increase code coverage. Indeed, some error handling code is only triggered when related error sites fail in specific calling contexts and succeed in other calling contexts, but the resulting tool always makes these error sites fail and cannot cover such code. The results indicate that our context-sensitive SFI method is effective in covering hard-to-trigger error handling code.

Besides, the resulting tool finds 9 bugs (including 8 null-pointer dereferences and 1 memory error). All these bugs are also reported by FIFUZZ, but 41 bugs found by FIFUZZ are missed by this tool, because it does not consider calling contexts of error sites. The results indicate that our context-sensitive SFI method is effective in finding deep bugs in different calling contexts.

5.6 Comparison to Existing Fuzzing Tools

Many fuzzing approaches have proposed to test infrequently executed code and show promising results in bug detection. Among them, we select four state-of-the-art and open-source fuzzing tools to make detailed comparison, including AFL [1], AFLFast [13], AFLSmart [50] and FairFuzz [38]. Meanwhile, to validate the generality of FIFUZZ, we select 5 common programs (including nm, objdump, size, ar and readelf) in the Binutils toolset [12] of an old version 2.26 (release in January 2016) as tested programs, instead of the 8 applications of the latest versions in the above experiments. We use FIFUZZ and the four fuzzing tools to fuzz each program without using any sanitizer for three times, and the time limit of each fuzzing is 24 hours. For the alerts or crashes reported by these tools, we also check their root causes to count unique bugs.

Figure 13 plots the covered code branches of each tested program during fuzzing. Compared to AFL and AFLFast, FIFUZZ covers more code branches in all the tested programs, by covering much more error handling code. Compared to

<table>
<thead>
<tr>
<th>Program</th>
<th>FIFUZZ_insensitive</th>
<th>FIFUZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usefull error sequence</td>
<td>Alert</td>
<td>Bug</td>
</tr>
<tr>
<td>clam</td>
<td>489</td>
<td>1</td>
</tr>
<tr>
<td>binom</td>
<td>108</td>
<td>3</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>fast</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>canvas</td>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td>clamav</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>cllow</td>
<td>105</td>
<td>1</td>
</tr>
<tr>
<td>gif2png+libpng</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>openssl</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>994</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 8: Results of sensitivity analysis.
AFLSmart and FairFuzz, FIFUZZ covers more code branches in *nm*, *size* and *ar*, but covers less code branches in *objdump* and *readelf*. The main reason is that the fuzzing process of program inputs in FIFUZZ is implemented by referring to AFL, while AFLSmart and FairFuzz use some techniques to improve mutation and seed selection of fuzzing program inputs compared to AFL. For this reason, AFLSmart and FairFuzz can cover more infrequently executed code related to inputs than FIFUZZ, though they still miss much error handling code covered by FIFUZZ. We believe that if we implement their fuzzing process of program inputs in FIFUZZ, it can cover more code branches than AFLSmart and FairFuzz in all the tested programs.

Table 9 shows the results of bug detection. Firstly, the two bugs found by AFL and AFLFast are also found by AFLSmart, FairFuzz and FIFUZZ. Secondly, AFLSmart and FairFuzz respectively find one bug missed by AFL, AFLFast and FIFUZZ. The one extra bug found by AFLSmart is different from that found by FairFuzz, as they improve mutation and seed selection for program inputs in different ways. Finally, FIFUZZ finds 14 bugs, and 12 of them related to error handling code are missed by AFL, AFLFast, AFLSmart and FairFuzz.

6 Discussion

6.1 False Positives of Error-Site Extraction

Our static analysis in Section 4.1 describes how to identify possible error sites from the tested program code. However, as shown in Section 5.2, our static analysis still has some false positives in identifying error sites, due to two main reasons:

Firstly, some functions that return pointers or integers never cause errors, even though their return values are often checked in the code. The functions `strcmp` and `strstr` are examples. However, our static analysis still considers that such functions can cause error, and identifies the function calls to them as possible error sites, causing false positives. To solve this problem, we plan to analyze the definition and call graph of each such function, to check whether it can indeed return an erroneous value that represents an error.

Secondly, a function can indeed fail and trigger error handling code, but some function calls to this function never fail considering their code contexts. This case can occur for some function calls that can cause input-related errors, when all possible inputs may have been changed into valid data before these function calls are used. However, our static analysis still identifies these function calls as possible error sites, causing false positives. To solve this problem, we plan to use symbolic execution [36] to analyze code context and calculate the constraints for each identified function call.

6.2 False Negatives of Bug Detection

FIFUZZ may miss real bugs in error handling code due to three possible reasons:

Firstly, as described in Section 4.1, to avoid injecting repeated faults, we only consider library functions for fault injection. However, some functions defined in the tested program can also fail, and they do not call any library function. Thus, FIFUZZ does not cover the error handling code caused by the failures of the calls to such functions.

Secondly, some error sites are executed only when specific program inputs and configuration are provided. In the evaluation, FIFUZZ cannot provide all possible program inputs and configuration. As a result, some error sites may not be executed, and thus their error handling code cannot be covered.
Thirdly, we only detect the bugs causing crashes and those reported by ASan. We can use other checkers to detect other kinds of bugs, such as MSan [41] which detects uninitialized uses, UBSan [57] which detects undefined behaviors, and TSan [56] which detects concurrency bugs.

6.3 Manual Analyses

FIFUZZ requires two manual analyses in this paper. Firstly, we perform a manual study in Section 2.3. This manual study is required for gaining the insights into building the automated static analysis, and we believe that the manual study provides the most representative and comprehensive results that help estimate the causes of errors. Secondly, in Section 5.2, we manually select realistic error sites from the possible error sites identified by FIFUZZ. This manual selection is required, as the static analysis of identifying possible error sites still has many false positives. For example, as shown in Table 4, we manually check the possible error sites identified by the static analysis, and find that only 18.6% of them are real. We believe that improving the accuracy of this static analysis can help reduce such manual work.

6.4 Performance Improvement

The performance of FIFUZZ can be improved in several ways:

**Dropping useless error sequences.** As shown in Table 5, FIFUZZ generates many useless error sequences that fail to increase code coverage. However, they are still used in fault injection to execute the tested program, reducing the fuzzing efficiency. We believe that static analysis can be helpful to dropping these useless error sequences. For example, after an original error sequence mutates and generates new error sequences, a static analysis can be used to analyze the code of the tested program, and infer whether each new error sequence can increase code coverage compared to the original error sequence. If not, this error sequence will be dropped, before being used in fault injection to execute the tested program.

**Lightweight runtime monitoring.** As shown in Figure 8, to collect runtime calling context, FIFUZZ instruments each function call to the function defined in the tested program code and each function definition. Thus, obvious runtime overhead may be introduced. To reduce runtime overhead, FIFUZZ can use some existing techniques of lightweight runtime monitoring, such as hardware-based tracing [3, 31] and call-path inferring [42].

**Multi-threading.** At present, FIFUZZ works on simple thread. Referring to AFL, to improve efficiency, FIFUZZ can work on multiple threads. Specifically, after an original error sequence mutates and generates new error sequences, FIFUZZ can use each new error sequence for fault injection and execute the tested program on a separate thread. When synchronization is required, all the execution results and generated error sequences can be managed in a specific thread.

6.5 Exploitability of Error Handling Bugs

To detect bugs in error handling code, FIFUZZ injects errors in specific orders according to calling context. Thus, to actually reproduce and exploit a bug found by FIFUZZ, 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 experiments are related to failures of heap-memory allocations. Thus, an intuitive exploitation way is to exhaustively consume the heap memory, which has been used in some approaches [58, 66] to perform attacks.

For the second requirement, as we have the error sequence of 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 [62, 63], if the window is too small, the exploitation may not be feasible.

7 Related Work

7.1 Fuzzing

Fuzzing is a promising 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 the hybrid of them.

Generation-based fuzzing approaches [15, 27, 59, 64] generate inputs according to the specific input format or grammar. Csmith [64] 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. These generated programs contain complex code using different kinds of C-language features free of undefined behaviors. LangFuzz [29] is a black-box fuzzing framework for programming-language (PL) interpreters based on a context-free grammar. Given a specific language grammar, LangFuzz generates many programs in this language as inputs for the tested language interpreter. To improve possibility of finding bugs, LangFuzz uses the language grammar to learn code fragments from a given code base.

Mutation-based fuzzing approaches [1, 7, 13, 26, 30, 38, 51, 65] start from some original seeds, and perform mutation of the selected seeds, to generate new inputs, without requirement of specific format or grammar. 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 [1] is a well-known coverage-guided fuzzing framework, which has been widely-used in industry and research. It uses many effective fuzzing
strategies and technical tricks to reduce runtime overhead and improve fuzzing efficiency. To improve mutation for inputs, FairFuzz [38] first identifies the code branches that are rarely hit by previously-generated inputs, and then uses a new lightweight mutation method to increase the probability of hitting the identified branches. Specifically, this method analyzes the input hitting a rarely hit branch, to identify the parts of this input that are crucial to satisfy the conditions of hitting that branch; this method never changes the identified parts of the input during mutation.

Some approaches [6, 45, 50, 60] combine generation-based and mutation-based fuzzing to efficiently find deep bugs. AFLSmart [50] uses a high-level structural representation of the seed file to generate new files. It mutates on the file-structure level instead of on the bit level, which can completely explore new input domains without breaking file validity. Superior [60] is a grammar-aware and coverage-based fuzzing approach to test programs that process structured inputs. Given the grammar of inputs, it uses a grammar-aware trimming strategy to trim test inputs using the abstract syntax trees of parsed inputs. It also uses two grammar-aware mutation strategies to quickly carry the fuzzing exploration.

Existing fuzzing approaches focus on generating inputs to cover infrequently executed code. However, this way cannot effectively cover error handling code triggered by non-input occasional errors. To solve this problem, FIFUZZ introduces software fault injection in fuzzing, and fuzzes injected faults according to the feedback of program execution. In this way, it can effectively cover error handling code.

7.2 Software Fault Injection

Software fault injection (SFI) [52] is a classical and widely-used technique of runtime testing. SFI 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 during execution. Many existing SFI-based approaches [9–11, 18, 25, 39, 40, 55, 67] have shown promising results in testing error handling code.

Some approaches [9, 10, 55] inject single fault in each test case to efficiently cover error handling code triggered by just one error. PairCheck [9] first injects single fault by corrupting the return values of specific function calls that can fail and trigger error handling code, to collect runtime information about error handling code. Then, it performs a statistical analysis of the collected runtime information to mine pairs of resource-acquire and resource-release functions. Finally, based on the mined function pairs, it detects resource-release omissions in error handling code.

To cover more error handling code, some approaches [11, 18, 25, 39, 40, 67] inject multiple faults in each test case. Some of them [25, 39, 40] inject random faults, namely they inject faults on random sites or randomly change program data. However, some studies [35, 43, 44] 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 [11, 18, 67] analyze program information to guide fault injection, which can achieve higher code coverage and detect more bugs. ADFI [18] uses a bounded trace-based iterative generation strategy to reduce fault scenario searching, and uses a permutation-based replay mechanism to ensure the fidelity of runtime fault injection.

To our knowledge, existing SFI-based approaches perform only context-insensitive fault injection. Specifically, they inject faults based on the locations of error sites in source code, without considering the execution contexts of these error sites. Thus, if an fault is constantly injected into an error site, this error site will always fail when being executed at runtime. However, some error handling code is only triggered when related error site fails in a specific calling context but succeeds in other calling contexts. In this case, existing SFI-based approaches cannot effectively cover such error handling code, and thus often miss related bugs.

7.3 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 [28, 32, 33, 37, 53] use static analysis to detect bugs in error handling code. EDP [28] 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 [33] 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 is 22%). However, static analysis could be introduced in FIFUZZ to drop useless error sequences, which can improve its fuzzing efficiency.

8 Conclusion

Error handling code is error-prone and hard-to-test, and existing fuzzing approaches cannot effectively test such code especially triggered by occasional errors. To solve this problem, we propose a new fuzzing framework named FIFUZZ, to effectively test error handling code and detect bugs. The core of FIFUZZ is a context-sensitive software fault injection (SFI) approach, which can effectively cover error handling code in different calling contexts to find deep bugs hidden in error handling code with complicated contexts. We have evaluated FIFUZZ on 9 widely-used C applications. It reports 317 alerts, which are caused by 50 new and unique bugs in terms of their root causes. 32 of these bugs have been confirmed by related
developers. The comparison to existing fuzzing tools shows that, FIFUZZ can find many bugs missed by these tools.

FIFUZZ can be still improved in some aspects. Firstly, the static analysis of identifying possible error sites still has many false positives. We plan to reduce these false positives using the ways mentioned in Section 6.1. Secondly, we plan to improve FIFUZZ’s performance in some ways, such as dropping useless error sequences, performing lightweight runtime monitoring and exploiting multi-threading mentioned in Section 6.4. Finally, we only use FIFUZZ to test C programs at present, and we plan to test the program in other programming languages (such as C++ and Java).

Acknowledgment

We thank our shepherd, Deian Stefan, and anonymous reviewers of the 4th International Workshop on Programming Languages and Analysis for Security (PLAS) (2009), pp. 45–57.

References


Appendix

We randomly select 50 of the 317 alerts reported by FIFUZZ in the 9 tested applications, and show their information in the table. These 50 alerts are caused by 36 bugs in terms of their root causes.

The column “Error points” shows the call stacks of error points (ErrPt) that trigger the alert. A call stack presents the information of each function call in the stack, including the name of the called function and code line number of this function call.

The columns “Source file” and “Line” respectively show the source file name and code line number where the alert occurs.

The column “State” shows the current state of our bug report. “F” means that the bug has been confirmed and fixed; “C” means that the bug has been confirmed but not fixed yet; “R” means that the bug report has not been replied.

<table>
<thead>
<tr>
<th>Program</th>
<th>Error points</th>
<th>Source file</th>
<th>Line</th>
<th>Alert type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>vim</td>
<td>ErrPt: main -&gt; common_init(173) -&gt; alloc(934) -&gt; laollo(827) -&gt; malloc(924)</td>
<td>message.c</td>
<td>4334</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>vim</td>
<td>ErrPt: main -&gt; mch_early.init(115) -&gt; alloc(3212) -&gt; laollo(827) -&gt; malloc(924)</td>
<td>message.c</td>
<td>4334</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>vim</td>
<td>ErrPt: main -&gt; termcapinit(384) -&gt; set_termname(2571) -&gt; set_shellsize(2069) -&gt; screen_clear(3466) -&gt; screenalloc(8744) -&gt; laollo(8495) -&gt; malloc(924)</td>
<td>screen.c</td>
<td>8664</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>vim</td>
<td>ErrPt: main -&gt; find_output_file(3317) -&gt; open_output_file(3277) -&gt; avformat_alloc_output_context2(2152) -&gt; avformat_alloc_option(4872) -&gt; ff_ac3_float_encode_init(491)</td>
<td>utils.c</td>
<td>205</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>vim</td>
<td>ErrPt: main -&gt; transcode(4894) -&gt; transcodedatestep(4692) -&gt; process_input(4638) -&gt; process_input(packet(4518)</td>
<td>avformat.c</td>
<td>87</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; option.c</td>
<td>205</td>
<td>null-pointer dereference</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; process_input(4638) -&gt; process_input(packet(4518)</td>
<td>avformat.c</td>
<td>87</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; avformat_find_stream_info(1126) -&gt; avformat_open(23674) -&gt; avmalloc(624) -&gt; avmalloc(238) -&gt; posix_memalign(87)</td>
<td>mem.c</td>
<td>223</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>Program</td>
<td>Error points</td>
<td>File name</td>
<td>Line</td>
<td>Bug type</td>
<td>State</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; transcode(4984) -&gt; transcode_step(4692) -&gt; process_input(4638) -&gt; process_input_packet(4518) -&gt; decode_audio(2619) -&gt; send_frame_to_filters(2337) -&gt; send_frame(2270) -&gt; configure_filtergraph(2189) -&gt; avfilter_graph_config(1109) -&gt; graph_configs(1275) -&gt; query_formats(1164) -&gt; ff_merge_channel_layouts(499) -&gt; av_realloc_array(242) -&gt; av_realloc(202) -&gt; realloc(144)</td>
<td>avfiltergraph.c</td>
<td>583</td>
<td>use after free</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; transcode(4984) -&gt; transcode_step(4692) -&gt; process_input(4638) -&gt; process_input_packet(4518) -&gt; decode_audio(2619) -&gt; send_frame_to_filters(2337) -&gt; send_frame(2270) -&gt; configure_filtergraph(2189) -&gt; configure_output_filter(1106) -&gt; configure_output_audio_filter(685) -&gt; choose_channel_layouts(606) -&gt; avio_close_dyn_buf(194) -&gt; avio_flush(1431) -&gt; flush_buffer(241) -&gt; writelnout(184) -&gt; dyn_buf_write(163) -&gt; av_realloc(1139) -&gt; av_realloc(173) -&gt; realloc(144)</td>
<td>ffmpeg_filter.c</td>
<td>179</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; ffmpeg_parse_options(4872) -&gt; open_files(3317) -&gt; open_output_file(3277) -&gt; avio_open(2358) -&gt; ffio_open(1180) -&gt; ffo_open_whiteclist(124) -&gt; ffo_open(1169) -&gt; av_realloc(256) -&gt; realloc(144)</td>
<td>mem.c</td>
<td>233</td>
<td>double free</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; ffmpeg_parse_options(4872) -&gt; open_files(3317) -&gt; open_input_file(3277) -&gt; avformat_open_input(1104) -&gt; init_input(573) -&gt; io_open_default(438) -&gt; ffo_open_whiteclist(124) -&gt; ffo_open(1169) -&gt; av_realloc(256) -&gt; realloc(144)</td>
<td>mem.c</td>
<td>233</td>
<td>double free</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; ffmpeg_parse_options(4872) -&gt; open_files(3317) -&gt; open_input_file(3277) -&gt; avformat_find_stream_info(1126) -&gt; try_decode_frame(3903) -&gt; avcodec_open2(3050) -&gt; ff_decode_bsf_init(736) -&gt; av_bsf_alloc(232) -&gt; av_strdup(87) -&gt; av_realloc(173) -&gt; av_realloc(256) -&gt; realloc(144)</td>
<td>decode.c</td>
<td>2059</td>
<td>freeing dangling pointer</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; ffmpeg_parse_options(4872) -&gt; open_files(3317) -&gt; open_input_file(3277) -&gt; avformat_find_stream_info(1126) -&gt; av_dict_set(1590) -&gt; av_dict_set1(1605) -&gt; av_dict_set(1900) -&gt; av_realloc(256) -&gt; realloc(144)</td>
<td>aacdec_template.c</td>
<td>2659</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>ErrPt: main -&gt; ffmpeg_parse_options(4872) -&gt; open_files(3317) -&gt; open_input_file(3277) -&gt; avformat_find_stream_info(1126) -&gt; av_dict_set(1900) -&gt; av_realloc(256) -&gt; realloc(144)</td>
<td>aacdec_template.c</td>
<td>2659</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>naasm</td>
<td>ErrPt: main -&gt; naas_init(479) -&gt; naasm_malloc(56) -&gt; malloc(75)</td>
<td>naasm.c</td>
<td>1909</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>naasm</td>
<td>ErrPt: main -&gt; init_labels(476) -&gt; naasm_malloc(563) -&gt; malloc(75)</td>
<td>naasm.c</td>
<td>1909</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>naasm</td>
<td>ErrPt: main -&gt; init_labels(479) -&gt; naasm_malloc(47) -&gt; calloc(85)</td>
<td>naasm.c</td>
<td>1909</td>
<td>null-pointer dereference</td>
<td>F</td>
</tr>
<tr>
<td>callcc</td>
<td>ErrPt: main -&gt; read_charsert(112) -&gt; calloc(93)</td>
<td>callcc.c</td>
<td>95</td>
<td>null-pointer dereference</td>
<td>R</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_loadfmt(5184) -&gt; cli_parse_add(2156) -&gt; cli_ac_addsig(497) -&gt; cli_ac_addpatt(2835) -&gt; cli_ac_addpatt_recursive(340) -&gt; add_new_node(299) -&gt; calloc(236) -&gt; calloc(216)</td>
<td>matcher-ac.c</td>
<td>578</td>
<td>use after free</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_loadfmt(5184) -&gt; cli_parse_add(2156) -&gt; cli_ac_addsig(497) -&gt; cli_ac_addpatt(2835) -&gt; cli_ac_addpatt_recursive(340) -&gt; cli_ac_addpatt_recursive(105) -&gt; add_new_node(299) -&gt; calloc(229) -&gt; calloc(216)</td>
<td>matcher-ac.c</td>
<td>578</td>
<td>use after free</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_load(833) -&gt; cli_dbaddbr(4726) -&gt; cli_load(4581) -&gt; cli_cvlodec(4341) -&gt; cli_tgload(706) -&gt; cli_load(4392) -&gt; cli_introut(1661) -&gt; cli_bm_init(678) -&gt; calloc(147) -&gt; calloc(216)</td>
<td>matcher-bm.c</td>
<td>224</td>
<td>double free</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_load(833) -&gt; cli_dbaddbr(4726) -&gt; cli_load(4581) -&gt; cli_cvlodec(4341) -&gt; cli_tgload(706) -&gt; cli_load(4392) -&gt; cli_introut(1661) -&gt; cli_bm_init(678) -&gt; calloc(147) -&gt; calloc(216)</td>
<td>bytecode.c</td>
<td>1931</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
<tr>
<td>clamav</td>
<td>ErrPt: main -&gt; scanmanager(161) -&gt; cl_engine_compile(861) -&gt; cli_bytecode_prepare(2520) -&gt; selfcheck(2683) -&gt; add_selfcheck(2479) -&gt; calloc(2397) -&gt; calloc(216)</td>
<td>doig.c</td>
<td>136</td>
<td>null-pointer dereference</td>
<td>C</td>
</tr>
</tbody>
</table>