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

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

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 handing is critical, itself is error-prone. Firstly, error handling code is difficult to correctly implement [14, 23, 34, 54] because it often involves special and complicated semantics. Secondly, error handling code is also challenging to test [25, 28, 53, 61], 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, 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 handing 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 `FuncA` 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.

```
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 handing 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, *PatchB* [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.

```
--- a/libavcodec/aacsbr.c
+++ b/libavcodec/aacsbr.c
@@ -334,6 +334,9
static int sbr_make_f_master(AACContext *ac,
                   SpectralBandReplication *sbr, ...) {
    ...
+   if (!sbr->sample_rate)
+       return -1;
    // BUG: sbr->sample_rate may be zero
    start_min = ... / sbr->sample_rate;
    ...
}
```

Figure 2: Patch A: fixing a divide-by-zero bug.

```
--- a/libavutil/frame.c
+++ b/libavutil/frame.c
@@ -383,8 +383,8
int av_frame_copy_props(...) {
    ...
    AVFrameSideData *sd_dst = av_frame_new_side_data(...);
    if (!sd_dst) {
        for (i = 0; i < dst->nb_side_data; i++) {
            av_freep(&dst->side_data[i]->data);
-           av_freep(&dst->side_data[i]);
            av_dict_free(&dst->side_data[i]->metadata);
+           av_freep(&dst->side_data[i]);
        }
    }
}
```

Figure 3: Patch B: fixing a use-after-free bug.

The bug in Figure 3 is instead caused by incorrect handling of an occasional error, because *av_frame_new_side_data* fails only when memory is insufficient, which occasionally occurs at runtime.

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].

Application	Studied file	Error site	Input-related	Occasional
vim	100	1163	530 (46%)	633 (54%)
bison	100	184	96 (52%)	88 (48%)
ffmpeg	100	881	518 (59%)	363 (41%)
nasm	100	673	564 (84%)	109 (16%)
catdoc	29	91	43 (47%)	48 (53%)
clamav	100	1089	522 (48%)	567 (52%)
cflow	100	286	170 (59%)	116 (41%)
gif2png+libpng	95	830	556 (67%)	274 (33%)
openssl	100	989	571 (58%)	418 (42%)
Total	824	6,168	3,570 (58%)	2,616 (42%)

Table 1: Study results of error handling code.

Tool	CVE	Error handling	Occasional error
AFL	218	85	3
Honggfuzz	57	17	3
AFLFast	8	2	0
CollAFL	93	15	4
QSYM	6	0	0
REDQUEEN	11	2	1
Total	393	121	11

Table 2: Study results of existing fuzzing tools.

2.4 Study of CVEs Found by Existing Fuzzing

To understand how existing fuzzing tools perform in detecting bugs in error handling code, we further study the CVEs found by some start-of-the-art fuzzing tools, including AFL [1], Honggfuzz [30], AFLFast [13], CollAFL [26], QSYM [65] and REDQUEEN [7]. We select these fuzzing tools because CVEs found by them are publicly available. Specifically, for AFL, a website [2] collects its found CVEs; for Honggfuzz, the found CVEs are listed in its homepage; for AFLFast, CollAFL, QSYM and REDQUEEN, the found CVEs are listed in their papers as well. We manually read these CVEs and identify the ones related to error handling code, and also check whether the identified CVEs are related to occasional errors. Table 2 shows the study results.

We find that 31% of CVEs found by these fuzzing tools are caused by incorrect error handling code, such as the bug shown in Figure 2. Only 9% of these CVEs are related to occasional errors. This proportion is far less than the proportion (42%) of occasional error sites among all error sites (found in Section 2.3). The results indicate that *existing fuzzing tools may have missed many real bugs in error handling code triggered by occasional errors*. Thus, it is important to improve fuzzing to support the testing of 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 as 0-1 sequence that describes the failure situation of error points at runtime:

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

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.*

3.2 Error Sequence Model

Existing SFI-based approaches often use context-insensitive fault injection. Specifically, 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 an fault is injected into an error site, this error site will always fail when being executed at runtime. However, an error site can be executed in different calling contexts, and some real bugs (such as the double-free 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 propose a context-sensitive software fault injection (SFI) method. Besides the location of each error site, our method also considers the calling context of the error site to describe error points, namely:

$$ErrPt = < ErrLoc, CallCtx > \quad (2)$$

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:

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

$$CallInfo = < CallLoc, FuncLoc > \quad (4)$$

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:

KEY	Hash($ErrPt_1$)	Hash($ErrPt_2$)	.....	Hash($ErrPt_x$)
VALUE	0 or 1	0 or 1	.....	0 or 1

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.

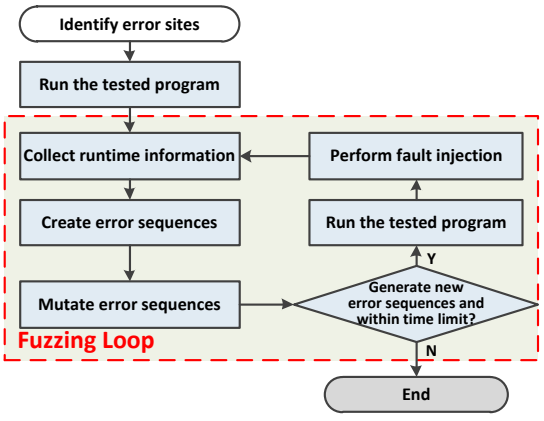


Figure 4: Procedure of our SFI-based fuzzing approach.

In our approach, mutating and generating error sequences are important operations. Given a program input, our approach considers code coverage in these operations and drops repeated error sequences. 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, it performs the subsequent generation and mutation of error sequences. All the generated error sequences that increase code coverage are stored in a pool, and they are ranked by contribution to code coverage. Our approach preferentially selects error sequences for mutation.

Initial mutation. Our approach first executes the tested program normally, and creates an initial error sequence according to runtime information. This error sequence contains 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 related calling context. Figure 5 shows an example of the initial mutation for an error sequence, which generates four new error sequences.

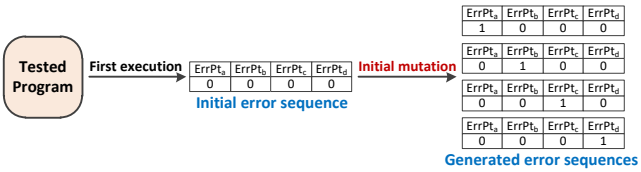


Figure 5: Example of the initial mutation.

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→1 or 1→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 $ErrPt_x$ is executed, and thus our approach creates an error sequence containing $ErrPt_x$. 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 $ErrPt_y$ is executed, and thus our approach creates an error sequence containing $ErrPt_y$. 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.

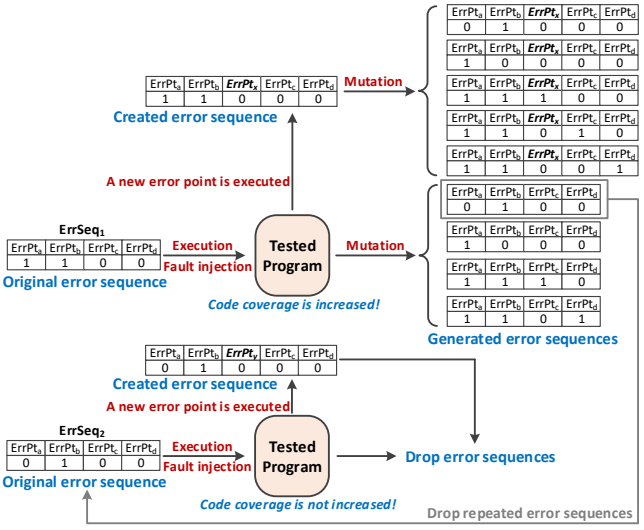


Figure 6: Example of the normal mutation.

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 idenfity possible error sites.
- ***Program generator.*** It performs code instrumention 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.

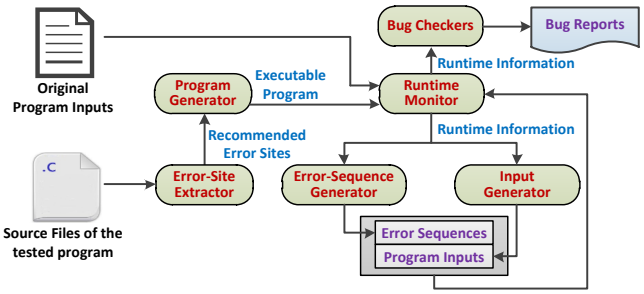


Figure 7: Overall architecture of FIFUZZ.

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.

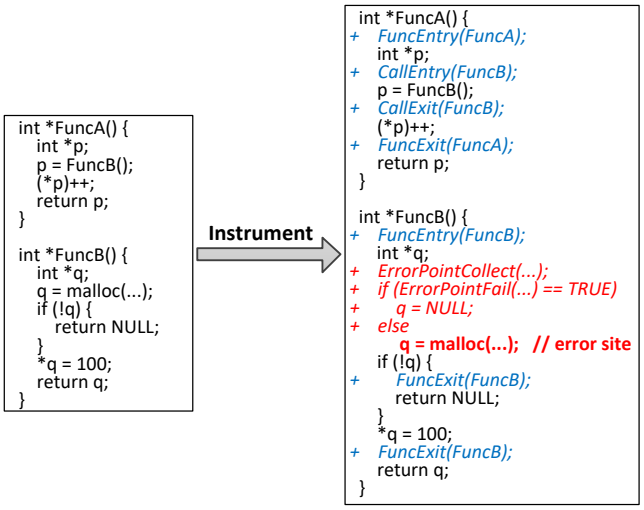


Figure 8: Example of code instrumentation.

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.

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

Table 3: Basic information of the tested applications.

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.

5.2 Error-Site Extraction

Before testing programs, FIFUZZ first performs a static analysis of their source code to first identify error functions that can fail, and then to identify error sites. We set $R = 0.6$ in this analysis, and perform the third step of this analysis for the source code of all the tested programs. After FIFUZZ produces identified error sites, we manually select realistic

Program	Function call	Identified	Realistic
vim	67,768	1,589 (2.3%)	283 (17.8%)
bison	11,861	966 (8.1%)	145 (15.0%)
ffmpeg	459,986	2,157 (0.5%)	190 (8.8%)
nasm	10,246	429 (4.2%)	44 (10.3%)
catdoc	1,293	103 (8.0%)	45 (43.7%)
clamav	52,830	2,183 (4.1%)	816 (37.4%)
cflow	4,049	149 (3.7%)	88 (59.1%)
gif2png+libpng	11,209	303 (2.7%)	54 (17.8%)
openssl	158,625	1916 (1.2%)	157 (8.2%)
Total	777,867	9,795 (1.3%)	1,822 (18.6%)

Table 4: Results of error-site extraction.

ones that can actually fail and trigger error handling code, by reading related source code. Table 4 shows the results. The first column presents the application name; the second column presents the number of all function calls in the application; the third column presents the number of error sites identified by FIFUZZ; the last column presents the number of realistic error sites that we manually select.

In total, FIFUZZ identifies 287 error functions, and identifies 9,795 function calls to these error functions as possible error sites. Among them, we manually select 150 error functions as realistic ones, and 1,822 function calls to these error functions that are considered as realistic error sites are automatically extracted from the source code. Thus, the accuracy rates of FIFUZZ for identifying realistic error functions and error sites are 52.3% and 18.6%. The manual confirmation is easily manageable and not hard. The user only needs to scan the definition of each error function, to check whether it can trigger an error by returning an error number or a null pointer. One master student spent only 2 hours on the manual selection of error functions for the 9 tested applications. Considering there are over 600K function calls in the tested programs, FIFUZZ is able to drop 99% of them, as they are considered not to be fail and trigger error handling code according to their contexts in source code. We find that many of the selected error functions and error sites are related to memory allocation that can indeed fail at runtime, and nearly half of the selected error functions and error sites are related to occasional errors. The results show that FIFUZZ can dramatically help reduce the manual work of identifying realistic error sites.

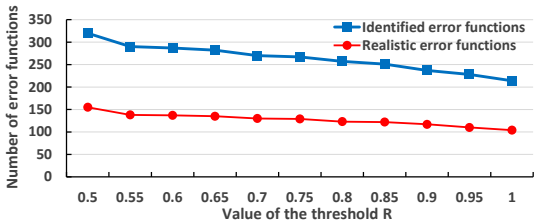


Figure 9: Variation of results affected by the value of R.

As described in Section 4.1, the value of $R = 0.6$ in the static analysis heavily affects the identified error functions. 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.

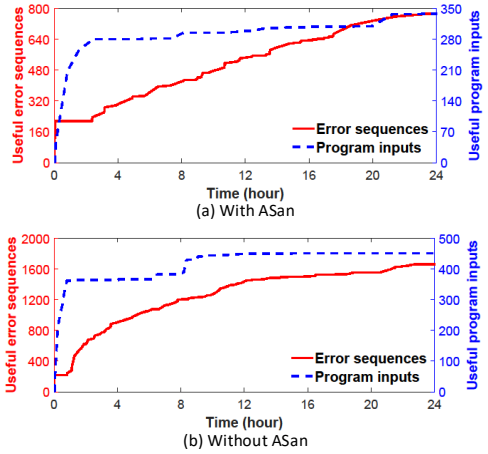


Figure 10: Variation of error sequences and inputs for *vim*.

Program	With ASan								Without ASan							
	Error sequence		Input		Reported alert				Error sequence		Input		Reported alert			
	Gen	*Useful*	*Gen*	*Useful*	*Null*	*MemErr*	*Assert*	*All*	*Gen*	*Useful*	*Gen*	*Useful*	*Null*	*MemErr*	*Assert*	*All*
vim	9,199	772	504,736	338	27	5	0	32	44,322	1,664	2,355,965	451	55	3	0	58
bison	1,450	221	1,995,831	1,168	11	0	0	11	8,692	289	14,602,760	1,207	11	0	0	11
ffmpeg	591	311	139,543	758	13	13	3	29	3,060	516	4,817,284	1,766	14	18	3	35
nasm	5,316	65	2,571,182	2,748	8	0	0	8	38,667	78	17,326,673	4,203	8	0	0	8
catdoc	84	34	4,721,501	158	1	1	0	2	798	38	40,357,609	234	2	0	0	2
clamav	482	339	98,352	26	7	106	0	113	482	325	331,930	34	7	96	0	103
cflow	1,623	159	4,551,244	724	0	0	0	0	12,209	217	29,909,026	1,235	1	0	0	1
gif2png+libpng	781	9	15,019,720	320	0	1	0	1	1,498	6	29,717,956	409	0	0	0	0
openssl	73,200	626	369,613	13	59	0	0	59	82,214	671	428,447	15	80	0	0	80
Total	92,726	2,536	29,971,722	6,253	126	126	3	255	191,942	3,804	139,847,650	9,554	178	117	3	298

Table 5: Fuzzing results.

Type	*Null*	*MemErr*	*Assert*	*All / Error handling*
Unique alert	182	132	3	317 / 313
Found bug	36	13	1	50 / 46
Confirmed bug	26	6	0	32 / 32

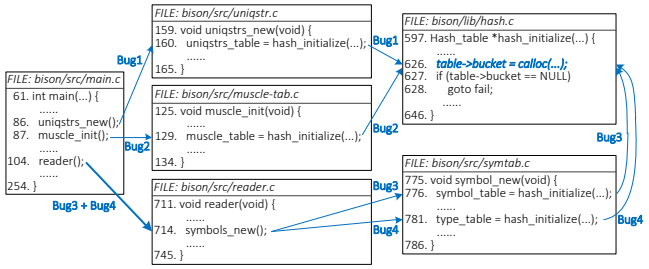
Table 6: Summary of reported alerts and found bugs.

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



Bug1: main -> uniqstrs_new(86) -> hash_initialize(160) -> *calloc(626)*
Bug2: main -> muscle_init(87) -> hash_initialize(129) -> *calloc(626)*
Bug3: main -> reader(104) -> symbols_new(714) -> hash_initialize(776) -> *calloc(626)*
Bug4: main -> reader(104) -> symbols_new(714) -> hash_initialize(781) -> *calloc(626)*

Figure 11: Example bugs caused by the same error site.

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 `calloc` 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 `calloc` 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 *calloc* 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.

Bug type	*Crash/DoS*	*Memory corruption*	*Arbitrary read*	*Memory overread*
Null pointer dereference	36	0	0	0
Double free	0	5	0	0
Use after free	0	1	2	2
Buffer overflow	0	0	0	1
Free invalid pointer	2	0	0	0
Assertion failure	1	0	0	0
Total	39	6	2	3

Table 7: Security impact classified by bug type.

```
FILE: clamav/libclamav/matcher-ac.c
572. void cli_ac_free(struct cli_matcher *root) {
       ......
577.   for (i = 0; i < root->ac_patterns; i++) {
         // "patt" can be "new" given specific "i"
578.     patt = root->ac_pattable[i];
         ......
580.     mpool_free(root->mempool, patt->virname)  // use after free
581.     if (patt->special) // use after free
         ......
620. }
-------------------------------------------------------------
2413. int cli_ac_addsig(struct cli_matcher *root, ...) {
       ......
2835.  if ((ret = cli_ac_addpatt(root, new))) {
         ......
2839.    mpool_free(root->mempool, new);  // free "new"
2840.    return ret;
2841.  }
       ......
2857. }
```

Figure 12: Two use-after-free bugs found in *clamav*.

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_addpatt` 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.

Program	FIFUZZ_insensitive			FIFUZZ		
	Useful error sequence	*Alert*	*Bug*	*Useful error sequence*	*Alert*	*Bug*
vim	689	1	1	1,664	58	12
bison	108	3	3	289	11	6
ffmpeg	5	0	0	516	35	12
nasm	7	2	1	78	8	1
catdoc	29	2	2	38	2	3
clamav	29	1	1	325	103	6
cflow	105	1	1	217	1	1
gif2png+libpng	4	0	0	6	0	1
openssl	18	0	0	671	80	8
Total	994	10	9	3,804	298	50

Table 8: Results of sensitivity analysis.

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 shown 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 lasted 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

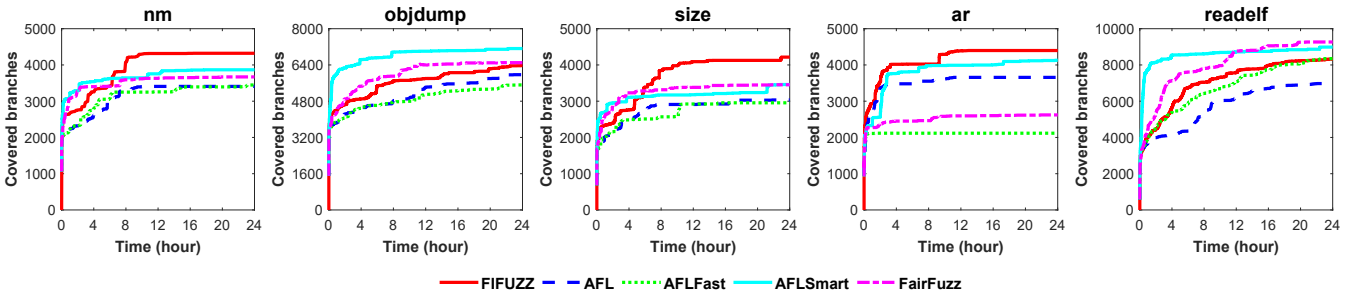


Figure 13: Code coverage of FIFUZZ and the four fuzzing tools.

Program	AFL			AFLFast			AFLSmart			FairFuzz			FIFUZZ		
	Null	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*
nm	0	1	1	0	1	1	0	1	1	0	1	1	4	1	5
objdump	0	1	1	0	1	1	1	1	2	0	1	1	2	1	3
size	0	0	0	0	0	0	0	0	0	0	1	1	2	0	2
ar	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4
readelf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	0	2	2	0	2	2	1	2	3	0	3	3	12	2	14

Table 9: Results of bug detection for comparison.

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 grammer. 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 grammer, LangFuzz generates many programs in this language as inputs for the tested language interpreter. To improve possibility of finding bugs, LangFuzz uses the language grammer 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 grammer. 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 explores new input domains without breaking file validity. Superion [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 resouce-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 for their helpful advice on the paper. This work was mainly supported by the China Postdoctoral Science Foundation under Project 2019T120093. Kangjie Lu was supported in part by the NSF award CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Jia-Ju Bai is the corresponding author.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] A collection of vulnerabilities discovered by the AFL fuzzer. https://github.com/mrash/afl-cve.

[3] ARM System Trace Macrocell (STM). https://community.arm.com/tools/b/blog/posts/introduction-to-arm-s-system-trace-macrocell.

[4] ASan: address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[5] ASan performance. https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers.

[6] ASCHERMANN, C., FRASSETTO, T., HOLZ, T., JAUERNIG, P., SADEGHI, A.-R., AND TEUCHERT, D. NAUTILUS: fishing for deep bugs with grammars. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[7] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. REDQUEEN: fuzzing with input-to-state correspondence. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[8] ASKAROV, A., AND SABELFELD, A. 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.

[9] BAI, J.-J., WANG, Y.-P., LIU, H.-Q., AND HU, S.-M. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology (IST) 73* (2016), 122–133.

[10] BAI, J.-J., WANG, Y.-P., YIN, J., AND HU, S.-M. Testing error handling code in device drivers using characteristic fault injection. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 635–647.

[11] BANABIC, R., AND CANDEA, G. Fast black-box testing of system recovery code. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)* (2012), pp. 281–294.

[12] GNU Binutils. http://www.gnu.org/software/binutils/.

[13] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)* (2016), pp. 1032–1043.

[14] CABRAL, B., AND MARQUES, P. 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.

[15] CHEN, Y., GROCE, A., ZHANG, C., WONG, W.-K., FERN, X., EIDE, E., AND REGEHR, J. Taming compiler fuzzers. In *Proceedings of the 34th International Conference on Programming Language Design and Implementation (PLDI)* (2013), pp. 197–208.

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

[17] CLOC: count lines of code. https://cloc.sourceforge.net.

[18] CONG, K., LEI, L., YANG, Z., AND XIE, F. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)* (2015), pp. 361–372.

[19] CVE-2019-7846. https://nvd.nist.gov/vuln/detail/CVE-2019-7846.

[20] CVE-2019-2240. https://nvd.nist.gov/vuln/detail/CVE-2019-2240.

[21] CVE-2019-1750. https://nvd.nist.gov/vuln/detail/CVE-2019-1750.

[22] CVE-2019-1785. https://nvd.nist.gov/vuln/detail/CVE-2019-1785.

[23] EBERT, F., AND CASTOR, F. A study on developers' perceptions about exception handling bugs. In *Proceedings of the 2013 International Conference on Software Maintenance (ICSM)* (2013), pp. 448–451.

[24] FFmpeg: a complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/.

[25] FU, C., RYDER, B. G., MILANOVA, A., AND WONNACOTT, D. Testing of Java web services for robustness. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA)* (2004), pp. 23–34.

[26] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. CollAFL: path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy* (2018), pp. 679–696.

[27] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)* (2008), pp. 206–215.

[28] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIBLIT, B. EIO: error handling is occasionally correct. In *Proceedings of the 6th International Conference on File and Storage Technologies (FAST)* (2008), pp. 207–222.

[29] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 445–458.

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

[31] Intel Processor Tracing (PT). https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[32] JANA, S., KANG, Y. J., ROTH, S., AND RAY, B. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Security Symposium* (2016), pp. 345–362.

[33] KANG, Y., RAY, B., AND JANA, S. 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.

[34] KERY, M. B., LE GOUES, C., AND MYERS, B. A. Examining programmer practices for locally handling exceptions. In *Proceedings of the 13th International Working Conference on Mining Software Repositories (MSR)* (2016), pp. 484–487.

[35] KIKUCHI, N., YOSHIMURA, T., SAKUMA, R., AND KONO, K. 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.

[36] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[37] LAWALL, J., LAURIE, B., HANSEN, R. R., PALIX, N., AND MULLER, G. Finding error handling bugs in openssl using Coccinelle. In *Proceedings of the 2010 European Dependable Computing Conference (EDCC)* (2010), pp. 191–196.

[38] LEMIEUX, C., AND SEN, K. 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.

[39] MARINESCU, P. D., AND CANDEA, G. 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.

[40] MENDONCA, M., AND NEVES, N. Robustness testing of the Windows DDK. In *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)* (2007), pp. 554–564.

[41] MSan: memory sanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer.

[42] MYTKOWICZ, T., COUGHLIN, D., AND DIWAN, A. Inferred call path profiling. In *Proceedings of the 24th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2009), pp. 175–190.

[43] NATELLA, R., COTRONEO, D., DURAES, J., AND MADEIRA, H. 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.

[44] NATELLA, R., COTRONEO, D., DURAES, J. A., AND MADEIRA, H. S. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering (TSE) 39*, 1 (2013), 80–96.

[45] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND LE TRAON, Y. Semantic fuzzing with Zest. In *Proceedings of the 2019 International Symposium on Software Testing and Analysis (ISSTA)* (2019), pp. 329–340.

[46] Aacsbr: check that sample_rate is not 0 before division. http://github.com/ffmpeg/ffmpeg/commit/a50a5ff29e.

[47] Found bug: libav: divide-by-zero in sbr_make_f_master. https://blogs.gentoo.org/ago/2016/09/21/libav-divide-by-zero-in-sbr_make_f_master-aacsbr-c/.

[48] CVE-2016-7499. https://nvd.nist.gov/vuln/detail/CVE-2016-7499.

[49] Frame: fix the error path in av_frame_copy_props. http://github.com/ffmpeg/ffmpeg/commit/a53551cba8.

[50] PHAM, V.-T., BÖHME, M., SANTOSA, A. E., CACIULESCU, A. R., AND ROYCHOUDHURY, A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* (2019).

[51] RAWAT, S., JAIN, V., KUMAR, A., COJOCAR, L., GIUFFRIDA, C., AND BOS, H. VUzzer: application-aware evolutionary fuzzing. In *Proceedings of the 24th Network and Distributed Systems Security Symposium (NDSS)* (2017), pp. 1–14.

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

[53] SAHA, S., LOZI, J., THOMAS, G., LAWALL, J. L., AND MULLER, G. 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.

[54] SHAH, H., GÖRG, C., AND HARROLD, M. J. Why do developers neglect exception handling? In *Proceedings of the 4th International Workshop on Exception Handling (WEH)* (2008), pp. 62–68.

[55] SUSSKRAUT, M., AND FETZER, C. Automatically finding and patching bad error handling. In *Proceedings of the 2006 European Dependable Computing Conference (EDCC)* (2006), pp. 13–22.

[56] TSan: thread sanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[57] UBSan: undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)* (2016), pp. 1675–1689.

[59] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy* (2017), pp. 579–594.

[60] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)* (2019), pp. 724–735.

[61] WEIMER, W., AND NECULA, G. C. 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.

[62] WU, W., CHEN, Y., XU, J., XING, X., GONG, X., AND ZOU, W. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 781–797.

[63] XU, W., LI, J., SHU, J., YANG, W., XIE, T., ZHANG, Y., AND GU, D. 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.

[64] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. 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.

[65] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 745–761.

[66] ZHANG, H., SHE, D., AND QIAN, Z. 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.

[67] ZHANG, P., AND ELBAUM, S. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (2012), pp. 595–605.

[68] ZUO, C., WU, J., AND GUO, S. 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.

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 ($ErrPt_x$) 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.

Program	Error points	Source file	Line	Alert type	State
vim	*ErrPt*$_1$: main -> common_init(173) -> alloc(934) -> lalloc(827) -> malloc(924)	message.c	4334	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> mch_early_init(115) -> alloc(3212) -> lalloc(827) -> malloc(924)	message.c	4334	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> termcapinit(384) -> set_termname(2571) -> set_shellsize(2069) -> screenclear(3466) -> screenalloc(8744) -> lalloc(8495) -> malloc(924)	screen.c	8664	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> termcapinit(384) -> set_termname(2571) -> set_shellsize(2069) -> screenclear(3466) -> screenalloc(8744) -> win_alloc_lines(8507) -> alloc_clear(5085) -> lalloc(851) -> malloc(924)	screen.c	8664	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> create_windows(728) -> open_buffer(2750) -> ml_open(167) -> ml_new_data(392) -> mf_new(4015) -> mf_alloc_bhdr(379) -> alloc(898) -> lalloc(827) -> malloc(924)	misc2.c	4446	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> common_init(173) -> set_init_1(1010) -> set_options_default(3522) -> set_option_default(3847) -> set_string_option_direct(3769) -> vim_strsave(5976) -> alloc(1279) -> lalloc(827) -> malloc(924)	charset.c	1456	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> common_init(173) -> set_init_1(1010) -> set_options_default(3522) -> set_option_default(3847) -> set_string_option_direct(3769) -> set_string_option_global(5987) -> vim_strsave(6083) -> alloc(1279) -> lalloc(827) -> malloc(924)	charset.c	1456	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> alist_add(2495) -> buflist_add(6688) -> buflist_new(3309) -> buf_copy_options(2036) -> vim_strsave(11649) -> alloc(1279) -> lalloc(827) -> malloc(924)	option.c	8422	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> save_typebuf(2365) -> alloc_typebuf(1332) -> alloc(1286) -> lalloc(827) -> malloc(924)	getchar.c	1313	double free	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> save_typebuf(2365) -> alloc_typebuf(1332) -> alloc(1287) -> lalloc(827) -> malloc(924)	getchar.c	1317	double free	F
vim	*ErrPt*$_1$: main -> init_highlight(413) -> do_highlight(415) -> syn_check_group(859) -> vim_strsave_up(3066) -> lalloc(827) -> malloc(924)	highlight.c	871	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> create_windows(728) -> open_buffer(2750) -> readfile(233) -> next_fenc(893) -> enc_canonize(2789) -> alloc(4323) -> lalloc(827) -> malloc(924)	fileio.c	2320	freeing invalid pointer	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> main_loop(903) -> msg_attr(1286) -> msg_attr_keep(122) -> set_vim_var_string(142) -> vim_strsave(7119) -> alloc(1279) -> lalloc(827) -> malloc(924)	message.c	1437	use after free	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> main_loop(903) -> normal_cmd(1370) -> do_pending_operator(1133) -> op_delete(1816) -> do_join(2079) -> alloc(4557) -> lalloc(827) -> malloc(924)	ops.c	4559	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> load_start_packages(492) -> do_in_path(2317) -> alloc(1864) -> lalloc(827) -> malloc(924) *ErrPt*$_2$: main -> vim_main2(444) -> wait_return(680) -> hit_return_msg(1078) -> msg_puts_attr_len(1961) -> alloc(2588) -> lalloc(827) -> malloc(924)	message.c	2589	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> source_startup_scripts(432) -> do_source(3051) -> fix_fname(2759) -> FullName_save(4817) -> vim_FullName(3082) -> mch_FullName(4479) -> fchdir(2589) *ErrPt*$_2$: main -> vim_main2(444) -> wait_return(680) -> hit_return_msg(1078) -> msg_putchar(1267) -> msg_putchar_attr(1369) -> msg_puts_attr(1386) -> msg_puts_attr_len(1961) -> msg_puts_printf(2008) -> alloc(2588) -> lalloc(827) -> malloc(924)	message.c	2589	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> uniqstrs_new(86) -> hash_initialize(160) -> malloc(605)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> reader(104) -> symbols_new(714) -> hash_initialize(776) -> malloc(605)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> muscle_init(87) -> hash_initialize(129) -> calloc(626)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> generate_states(124) -> allocate_storage(358) -> state_hash_new(168) -> hash_initialize(362) -> calloc(626)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> tables_generate(152) -> pack_table(802) -> bitset_create(727) -> bitset_alloc(163) -> bitset_init(138) -> vbitset_init(88) -> vbitset_resize(989) -> realloc(77)	vector.c	81	null-pointer dereference	F
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_alloc_context(1041) -> av_mallocz(151) -> av_malloc(238) -> posix_memalign(87)	dict.c	205	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> avformat_alloc_output_context2(2152) -> avformat_alloc_context(151) -> av_mallocz(151) -> av_malloc(238) -> posix_memalign(87)	dict.c	205	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> new_audio_stream(2236) -> new_output_stream(1859) -> avcodec_alloc_context3(1387) -> init_context_defaults(163) -> av_opt_set(141) -> set_string_number(484) -> av_expr_parse_and_eval(292) -> av_expr_parse(751) -> av_malloc(687) -> posix_memalign(87)	options.c	141	assertion failure	R
ffmpeg	*ErrPt*$_1$: main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> avfilter_graph_parse2(1056) -> parse_filter(427) -> av_get_token(184) -> av_malloc(151) -> posix_memalign(87)	avstrings.c	87	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_mallocz(624) -> av_malloc(238) -> posix_memalign(87)	utils.c	491	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> transcode(4894) -> transcode_step(4692) -> reap_filters(4648) -> init_output_stream(1442) -> avcodec_open2(3517) -> ff_ac3_float_encode_init(935) -> ff_ac3_encode_init(138) -> allocate_buffers(2481) -> ff_ac3_float_allocate_sample_buffers(2331) -> av_mallocz(49) -> av_malloc(238) -> posix_memalign(87)	mem.c	223	null-pointer dereference	R

Program	Error points	File name	Line	Bug type	State
ffmpeg	*ErrPt$_1$:* main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> avfilter_graph_config(1109) -> graph_config_formats(1275) -> query_formats(1164) -> ff_merge_channel_layouts(499) -> av_realloc_array(242) -> av_realloc(202) -> realloc(144)	avfiltergraph.c	583	use after free	R
ffmpeg	*ErrPt$_1$:* main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> configure_output_filter(1106) -> configure_output_audio_filter(685) -> choose_channel_layouts(606) -> avio_close_dyn_buf(194) -> avio_flush(1431) -> flush_buffer(241) -> writeout(184) -> dyn_buf_write(163) -> av_reallocp(1319) -> av_realloc(173) -> realloc(144)	ffmpeg_filter.c	179	null-pointer dereference	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> avio_open2(2558) -> ffio_open_whitelist(1180) -> ffio_fdopen(1169) -> av_strdup(1007) -> av_realloc(256) -> realloc(144)	mem.c	233	double free	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_open_input(1104) -> init_input(573) -> io_open_default(438) -> ffio_open_whitelist(124) -> ffio_fdopen(1169) -> av_strdup(1007) -> av_realloc(256) -> realloc(144)	mem.c	233	double free	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> ff_decode_bsfs_init(736) -> av_bsf_alloc(232) -> av_mallocz(86) -> av_malloc(238) -> posix_memalign(87)	decode.c	2059	freeing dangling pointer	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> aac_decode_init(935) -> ff_mdct_init(1226) -> ff_fft_init(61) -> av_malloc(224) -> posix_memalign(87)	aacdec_template.c	2659	null-pointer dereference	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> aac_decode_init(935) -> ff_mdct_init(1226) -> av_malloc_array(64) -> av_malloc(188) -> posix_memalign(87)	aacdec_template.c	2659	null-pointer dereference	R
nasm	*ErrPt$_1$:* main -> saa_init(479) -> nasm_malloc(56) -> malloc(75)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> init_labels(476) -> nasm_malloc(563) -> malloc(75)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> saa_init(479) -> nasm_zalloc(47) -> calloc(85)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> init_labels(476) -> hash_init(561) -> alloc_table(66) -> nasm_zalloc(60) -> calloc(85)	nasm.c	1909	null-pointer dereference	F
catdoc	*ErrPt$_1$:* main -> read_charset(112) -> calloc(93)	charsets.c	95	null-pointer dereference	R
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_calloc(236) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(608) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> cli_ac_addpatt_recursive(305) -> add_new_node(299) -> cli_calloc(229) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> load_oneldb(2020) -> cli_parse_add(1876) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_calloc(236) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> cli_initroots(1961) -> cli_bm_init(678) -> cli_calloc(147) -> calloc(216)	matcher-bm.c	224	double free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_bytecode_prepare2(5250) -> selfcheck(2683) -> add_selfcheck(2479) -> cli_calloc(2397) -> calloc(216)	bytecode.c	1931	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_cvdverify(625) -> cli_versig(566) -> cli_str2hex(131) -> cli_calloc(242) -> calloc(216)	dsig.c	136	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(608) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_realloc(245) -> realloc(235)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> cli_ac_addpatt_recursive(305) -> cli_ac_addpatt_recursive(305) -> insert_list(268) -> cli_realloc(106) -> realloc(235)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_bytecode_prepare2(5250) -> selfcheck(2683) -> add_selfcheck(2479) -> cli_realloc2(2348) -> realloc(254)	bytecode.c	1919	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> cli_initroots(1961) -> cli_ac_init(670) -> cli_malloc(521) -> malloc(197)	matcher-ac.c	614	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> scanfile(205) -> cl_scandesc_callback(391) -> scan_common(4324) -> malloc(4128)	scanners.c	4129	null-pointer dereference	R