Fuzzing API Error Handling Behaviors using Coverage Guided Fault Injection

Shashank Sharma
sharm611@purdue.edu
Purdue University
USA

Sai Ritvik Tanksalkar
stanksal@purdue.edu
Purdue University
USA

Sourag Cherupattamoolayil
scherupa@purdue.edu
Purdue University
USA

Aravind Machiry
amachiry@purdue.edu
Purdue University
USA

ABSTRACT

Incorrect handling of Software Application Programming Interfaces (APIs) errors results in bugs or security vulnerabilities that are hard to trigger during regular testing. Most of the existing techniques to detect such errors are based on static analysis and fail to identify certain cases where API return values are incorrectly handled. Furthermore, most of these techniques suffer from a very high false positive rate (≥50%), raising concerns regarding their practical use. We propose a dynamic analysis approach to detect API error handling bugs based on coverage-guided software fault injection. Specifically, we inject faults into APIs and observe how a program handles them. Our fault injection mechanism is generic and targeted to explore a given program's error handling behavior effectively. We avoid false positives by proactively filtering out crashes caused by infeasible faults. We implemented our technique in an automated pipeline called FuzzERR and applied it to 20 different programs spanning 444 APIs. Our evaluation shows that FuzzERR found 31 new and previously unknown bugs resulting from incorrect handling of API errors. Moreover, a comparative evaluation showed that FuzzERR significantly outperformed the state-of-the-art tools.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Vulnerability scanners**.

KEYWORDS

Program Analysis, Fuzzing, LLVM, Error Handling, API Error Handling

ACM Reference Format:
Shashank Sharma, Sai Ritvik Tanksalkar, Sourag Cherupattamoolayil, and Aravind Machiry. 2024. Fuzzing API Error Handling Behaviors using Coverage Guided Fault Injection. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3634737.3637650

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3637650

1 INTRODUCTION

Application Programming Interfaces (APIs) form the building blocks for modular software development by encapsulating complex functionality. In this work, we focus on software libraries (*i.e.,* shared object or DLL) that expose certain functionality as external functions. *e.g.,* `libpoppler.so` library exposes various functions to access PDF files. We call these external functions as API functions or APIs. The complexity and requirements of APIs make it challenging to use them correctly, resulting in API misuse bugs. A recent study [36] shows that 17% of bugs in application programs are because of API misuse. In this work, we focus on incorrect handling of API errors (*e.g.,* Listing 1), which is one of the major categories of API misuse bugs.

Existing techniques to find API error handling or, more general API misuse bugs can be broadly categorized into either specification or anomaly-based. (i) The specification-based techniques [17, 35] check for violations of a given valid and precise usage specification of API. However, as shown by a recent work [44], *writing precise specifications is tedious and requires considerable effort* by the developers. Furthermore, these specifications need to be written for every API. (ii) The anomaly-based techniques [6, 14, 75, 78] exploit the intuition that the misuse bugs are anomalous and will be in the minority [24]. Given a set of API usages, these techniques identify the minority usage patterns and consider them as bugs. However, these techniques fail to precisely capture certain common usage patterns and have *a high false positive rate (~50%) for complex APIs* (Section 7). Finally, as we explain in Section 2.2.1 (using a real-world example), although an API appears to be used correctly in the local context, there could be a misuse in the whole program context. In a few cases, error handling itself could be invalid. For instance, the return value of an API call is checked (*i.e.,* used in an `if` condition), but the check is semantically incorrect [1, 2].

Random testing, especially Fuzzing [32, 82], is shown to be an effective technique for software bug finding. However, triggering API misuse bugs requires exploring deep program states, and most often, these bugs require changing the program environment. For instance, to check misuses of `fopen`, we require it to fail and return `NULL`, which depends on the external environment, *i.e.,* file system, and its permissions. Dynamic Software Fault Injection (SFI) [5, 65] is one of the well-known techniques to inject faults during program execution to mimic real failures. However, these techniques require a specification of fault injection profile. Automated techniques are either specific to a particular class of applications, such as Operating System (OS) drivers [9, 10] or specific class of API methods [52]. Furthermore, *existing SFI techniques suffer from causing infeasible program states, resulting in false positives [40, 55].* Finally, we need fine-grained control over fault injection to expose deeper issues. For instance, the bug in Listing 1 requires the API function `poppler_document_get_page` to *only fail when called from a specific context and succeed in all other cases* – this is hard to achieve with many existing SFI techniques. The recent technique FiFuzz [38] tries to handle this by context-sensitive SFI, but it has false positives and requires manual effort to filter out incorrect fault injection points.

Based on prior works [36, 56] and our observations, an effective and practical SFI technique to detect API error handling bugs should satisfy the following requirements:

- *Generic.* The technique should be able to handle different APIs.
- *False Positives Filtering.* As automated fault injection might inevitably result in false positives, we should have a way to filter out potential false positives.
- *Fine-grained Fault Injection.* We should have fine-grained control over fault injection and should be able to inject faults at specific execution points.

We present FuzzERR, a dynamic analysis technique to detect API error handling bugs based on coverage guided SFI. For a given program (or program under test) and the target library implementing a set of APIs, the goal of FuzzERR is to find bugs in the program because of incorrect error handling of these APIs. The high-level idea is to make APIs fail and observe how the program under test handles these failures. We have designed FuzzERR to satisfy all the desired requirements:

Generic SFI. We developed a generic fault injection mechanism using a library-centric technique. Specifically, we inject faults into APIs by forcing their execution in the corresponding library along error paths. This enables us to reuse existing API error behavior without explicitly modeling their error behavior.

Filtering False Positives using Program Traces. We consider all crashes that happen in the program as true positives, as we expect the program to handle all API errors. However, as we explain in Section 4.2.2, fault injection could violate certain invariants resulting in false positive crashes. We developed a lightweight mechanism based on program traces to detect potential false positives crashes.

Fully Context Sensitive SFI. Our fault injection technique is coverage guided and fully context-sensitive. For instance, for an API called in a loop, our techniques can inject fault only in the second iteration of the loop. For a given program and an input, we repeatedly execute the program with the input. During each execution, we inject faults into the target library at different fault injection points, intending to improve the code coverage of the program. The Table 1 summarizes existing tools and how FuzzERR satisfies all the desired requirements.

We have implemented FuzzERR to be an automated pipeline and demonstrate its effectiveness by evaluating on 20 programs spanning across 12 libraries and applicable bugs from APIMU4C [36] dataset. FuzzERR found a total of 5,835 unique crashes resulting from 31 previously unknown API error handling bugs. In summary, the following are our contributions:

- We designed a novel and generic SFI mechanism by forcing executions along error paths.
- We implemented FuzzERR an automated pipeline with all our techniques, along with root cause identification.
- Our evaluation on 20 programs spanning across 12 libraries and a bug dataset show that FuzzERR found a total of 31 previously unknown bugs, out of which 20 are already confirmed and fixed by the corresponding developers.
- We have made our implementation open-source and publicly available at https://github.com/purs3lab/FuzzERR-final.

2 MOTIVATION

This section presents a motivating example and explains why existing techniques fail to identify the bug.

2.1 Motivating Example

The Listing 1 shows a real heap buffer overflow found by FuzzERR in the latest version of `apvlv` PDF reader. The execution leading to the bug is shown by the numbered symbol ❯(1-5).

2.1.1 Root cause. The bug occurs when the call to `libpoppler` API method `poppler_document_get_page` fails at line 33 in the function `pagesize` (indicated by ⛓). This failure returns a **`nullptr`**, which is consequently checked, and `false` is returned at line 39. Note that in this failure case, the function does not modify the parameters `x` and `y`. In the success case, *i.e.,* when `poppler_document_get_page` returns a valid page corresponding to `pn`, which is used to get the size and the parameters `x` and `y` are updated (code omitted in the listing for brevity).

2.1.2 Execution flow. The function `pagesize` is called at line 12, and the above-mentioned failure results in the variables `tpagex` and `tpagey` being uninitialized. However, the return value of `pagesize` is not checked, and subsequently, these uninitialized variables are used to derive the size of a heap array `dat` at lines 17 and 19, resulting in an array of invalid size. This heap array `dat` is then passed as an argument to the function `setAnnot` at line 24. The passed argument is accessed (via parameter `buffer`) using index `p+1`, which is derived from `ac->mFile`. This index value `p+1` can be larger than the allocated size and thus results in heap buffer overwrite as indicated by 🐞.

2.2 Inadequacy of Existing Techniques

The bug in Listing 1 captures various aspects that make existing techniques inadequate.

2.2.1 API Misuse Detection. As will be discussed in Section 7, these techniques focus on identifying misuses of a given API function. Specifically, these techniques [6, 13, 42, 44, 45, 57, 78] analyze the local usage context of the target API function and check whether it is valid. APISan [78], a recent work, checks for semantic patterns, *i.e.,* return value of an API function is checked before use. However, these semantic patterns do not sufficiently capture API usage semantics. There are several cases where an API return value is checked, but the check is incorrect. APISan fails to find such

Table 1: Comparision of FuzzERR to other tools. For each of the features, we indicate whether the technique fully supports (✓), or does not support (✗) the feature

Tool Category	Tool	Generic	False Positives Filtering	Fine Grained Fault Injection	No manual specification	No need for valid uses
Systematic Static Approaches	CodeQL [30]	✓	✗	N/A	✗	✓
Anomaly-based Static Approaches	ApiSan [78]	✓	✗	N/A	✓	✗
	FICS [6]	✓	✗	N/A	✓	✗
	Arbitrar [44]	✓	✗	N/A	✓	✓
Dynamic Fault Injection Approaches	FairFuzz [43]	✓	✗	✗	✓	✓
	Fifuzz [38]	✓	✗	✗	✓	✓
	Lfi [53]	✓	✗	✓	✗	✓
	FuzzERR (Our Work)	✓	✓	✓	✓	✓

bugs. For our example in Listing 1, the target API is `poppler_document_get_page`. However, it is being used correctly by checking the return value at line 34. Consequently, these techniques fail to find the bug. In fact, executing APISan on `apvlv` resulted in 76 warnings, and the bug in Listing 1 was not detected. Furthermore, we selected the top 10 highly ranked warnings and found 9 of them to be completely false (*i.e.,* the return value was either handled correctly or the warnings were false given how the library actually works). The other warning, although true in the general case, was a false positive in the case of `apvlv`. The warning was about the return value of `g_signal_connect()` not being checked. The function `g_signal_connect()` returns a handler id, which is needed (or to be checked) only if the id will be used later to disconnect using `g_signal_handler_disconnect`. However, `apvlv` does not disconnect, so not checking the return values doesn't affect it. Similarly, anomaly-based detection techniques, such as FICS [6], also fail to detect because of missing anomalies, *i.e.,* the return value of `poppler_document_get_page` is always checked, whereas the return value of `ApvlvPDF::pagesize` is never checked.

2.2.2 Automated Testing or Fuzzing. We require that the call to `poppler_document_get_page` at line 33 to fail (*i.e.,* to return **`nullptr`**), which depends on its arguments `mDoc` and `pn`. The value of the first argument `mDoc` is a document object pointer (global variable), not a function parameter, and thus cannot be controlled by input. The value of the second argument `pn` can be controlled through external input. However, the corresponding value is checked at line 6 to be within a valid range. Consequently, the value that reaches the call site will be valid. This makes it hard for the call to `poppler_document_get_page` to fail while fuzzing the corresponding program. Consequently, as we show in Section 5.4, the bug was never found by existing fuzzing techniques despite the corresponding code being extensively covered during fuzzing runs.

2.2.3 Software Fault Injection (SFI). First, existing SFI techniques [22, 22, 27, 80] require explicit specification of fault profiles *i.e.,* how and where the faults should be introduced. For our example in Listing 1, a developer needs to explicitly specify that fault should be injected at line 33 by assigning **`nullptr`** only when called from line 12. Note that, *always injecting fault would terminate the program early without executing the vulnerable code.* This sort of context-sensitive fault injection is not possible with the existing techniques. A recent work, Fifuzz [38], tries to handle this by using context-sensitive fault injection. But, Fifuzz focuses on identifying bugs in error handling code, but not on API error handling bugs. In other words, Fifuzz can find if an API error is incorrectly handled, but *cannot find if an API error is never handled (Listing 2).* As expected and also shown by our experiments in Section 5.6.2, Fifuzz fails to find several bugs found by FuzzERR. Finally, Fifuzz suffers from false positives because of imprecision in identifying fault injection points. Consequently, a large number (7,973 (81%) (Identified - Realistic) from Table 4 of the paper [38]) of these are required to be manually filtered out.

3 BACKGROUND

The goal of FuzzERR is to find error handling bugs in a given *program p* of APIs present in a given *library l*. This section presents the necessary technical background and explains the notations used in the rest of the paper.

3.1 Coverage Guided Fuzzing (CGF)

This automated testing technique aims to generate inputs that can improve the code coverage of the program under test. As explained in Section 7, there are many ways to generate inputs. We use mutation-based generation, wherein new inputs are generated by applying various mutations to the provided seed inputs. We use AFL++ [26], an extensible coverage guided and mutation-based fuzzing tool, to guide our fault injection.

3.2 Terms and Notations

3.2.1 Fault Injectable Program Points (FIPs). These are locations in a library where faults can be injected by FuzzERR. Specifically, a FIP is a marker to a control flow instruction (*i.e.,* **`if`**, **`while`**, **`switch`**, **`case`**, etc.) that checks for certain invalid conditions and consequently control execution into an error handling path. Note that not all functions will have FIPs. Each FIP has a unique id (a sequential positive integer) and is represented by a *FIP record*, which is a tuple: <source file name, function name, line number, column number, [true or false]>. For the following code:

```
// poppler-image.cpp
poppler::image::data(...) {
  if (..) {
   ....
   return NULL;
  }
}
```

Listing 1 A real heap-overwrite found by FuzzERR in `apvlv` PDF reader because of incorrect handling of `poppler_document_get_page` API failure.

```
void
ApvlvDocCache::load (ApvlvDocCache *ac)
{
  // mPagenum is checked to avoid the failure of
  // poppler_document_get_page
  if (ac->mPagenum < 0 || ac->mPagenum >= c) {
   debug ("no this page: %d", ac->mPagenum);
   return;
 }

  double tpagex, tpagey;
  ❯1 ac->mFile->pagesize (ac->mPagenum,
                    gint (ac->mRotate),
                    &tpagex, &tpagey);
  // if pagesize fails
  // tpagex, tpagey will be uninitialized.
  ❯3 ac->mSize = tpagex * tpagex * 3;
  // dat allocated wrong size.
  ❯4 auto *dat = new guchar[2 * ac->mSize];
  ...
  for (...)
  {
    ...
    ❯5 ac->setAnnot (annot, dat, ac->mSize);
  }
}

bool
ApvlvPDF::pagesize (int pn, int rot,
                    double *x, double *y)
{
  PopplerPage *page =
     poppler_document_get_page (mDoc, pn);
  if (page != nullptr) {
    ...
    // initialize x and y according to pn
    ...
  }
  ❯2 return false;
}

void
ApvlvDocCache::setAnnot (...,unsigned char *buffer,
                         size_t buf_size) const {
  // p derived from ac->mPagenum
  // Heap Buffer overwrite
  ❯6 buffer[p + 1] = ...
}
```

FIP(45) = <`poppler-image.cpp`, `poppler::image::data`, 283, 9, true>, indicates the FIP with id 45, and it is present in `poppler-image.cpp` file and in function `poppler::image::data` at line number 283 and character number 9. This indicates that a fault can be injected at the `if` condition by forcing execution along the true branch. Given a library l, we use FIP_l to denote the set of all FIPs in it.

3.2.2 *Fault Injectable Library (FILib).*

We call a library (*i.e.,* module or shared object) in which faults can be injected at FIPs as the *Fault Injectable Library (FILib).* Given a library l, we denote the corresponding fault injectable variant as l_f.

3.2.3 *Reachable Faults List (RFList).*

This is the list of all FIPs reached during an execution of the program. Specifically, for a given program p using a fault injectable library l_f and an input i, RFList(p, l_f, i) indicates the sequence of FIPs in l_f that are executed (*i.e.,* reached) when p is run with i. Formally, RFList(p, l_f, i) = $< FIP^1, FIP^2, ..., FIP^n >$, where $\forall x \in [1, n] \mid FIP^x \in FIP_l$. For example, RFList(`apvlv`, `libpoppler`, `test2.pdf`)=<12,45,45,45,9> indicate FIP ids of `libpoppler` that are reached (in that order) when `apvlv` is executed with `test2.pdf`. The repeated id 45 indicates that the corresponding library function is called in a loop or from multiple call sites. For simplicity, we assume deterministic execution, *i.e.,* executing the same program p with the same i and l_f results in the same RFList. Formally,

$$(p = p_1 \wedge i = i_1 \wedge l_f == l_f^1) \implies RFList(p, i, l_f) = RFList(p_1, i_1, l_f^1)$$

3.2.4 *Fault Injection List (FIList).*

For a given RFList of length k, an FIList is a sequence of k bits that indicate which of the corresponding faults should be injected. For the previously mentioned RFList *i.e.,* <12,45,45,45,9>, a possible FIList is <0,0,1,0,1>, which indicates that fault should be injected at two FIPs *i.e.,* at 45 (only for the second time) and at FIP 9. As we explain in Section 4.2.2, we execute the program repeatedly with the same input, consequently having the same RFList. Providing a different FIList every time enables us to have fine-grained control over where the fault injection should happen during the execution of the program.

3.2.5 *False positive or Infeasible crash.*

This is a crash (*e.g.,* segmentation fault) induced by fault injection, which is impossible to occur during regular program execution. Consider the following `if` condition:

```
if (p != NULL) { *p = 0;  return NULL; }
```

Let's assume that FuzzERR injected a fault which made the execution reach `*p = 0` even when `p` is `NULL`. This results in a program crash because of NULL-ptr dereference. However, the crash is impossible in real program runs and hence is a false positive or infeasible crash.

4 FUZZERR

First, we will present an overview (Section 4.1) of different steps in FuzzERR and how each works on our example in Listing 1. Second, we describe each step in detail (Section 4.2). Finally, we will present the implementation details (Section 4.2.4). Our system requires the program under test (p) and a library l, which is used by p. As mentioned before, the goal of FuzzERR is to find error handling bugs in p related to APIs in l.

4.1 Overview

The Figure 1 shows the overview of FuzzERR, which contains the following two distinct phases.

4.1.1 *Generating FILib of l (i.e., l_f).*

This phase is performed once for a given library. Given the source code of a library, we perform the following two steps.

FIP Identification (Section 4.2.1) We use source-level analysis and common error handling patterns to identify FIPs, which, as mentioned before (Section 3.2.1), are the conditional statements guarding execution into error handling paths. This results in the set of FIPs in l *i.e.,* FIP_l. For `libpoppler` used in Listing 1, the FIP_l contained 139 entries.

Library Instrumentation (Section 4.2.1) We instrument conditions represented by each FIP ($\in FIP_l$) so that execution can be forced along the error path guarded by the corresponding condition. An example of our instrumentation is shown below:

```
if ( has_fault(12) || !has_space()) {
    *ptr = NULL; return -1; }
```

Here, `12` is the ID of the corresponding FIP. This instrumentation enables us to force the execution inside the `if` condition based on the return value of the newly inserted call `has_fault()`. Consequently, injecting fault into the corresponding API.

The function `has_fault()` will be implemented by our *helper library* that also includes the necessary logic to inject faults in a fully-context sensitive manner. Finally, we link the instrumented library and our helper library to get FILib of the given library l, *i.e.,* l_f. Our instrumentation also adds a *record-only* mode in l_f that stores (without any fault injection) the list of FIPs reached during a run, which is needed to get RFList.

4.1.2 Program Testing (Section 4.2.2). This phase will be performed for each program that uses our target library l.

Generating RFList. For a given program p and an input i, we get the RFList by executing p with i and using l_f (instead of l) in record mode. As explained in Section 3.2.3, RFList gives the list of FIPs in l_f reached during the execution of p with i.

Coverage Guided Fault Injection. Next, we iteratively execute p with i and l_f. In each iteration, we provide a FIList corresponding to the RFList to l_f. The FIList precisely configures fault injection at various FIPs resulting in failure of APIs executed by p. We generate FIList using various mutation techniques, starting with a no-op FIList (*i.e.,* all bits set to 0). Although we execute p with the same input i, failures of different API calls (caused by FIList) can change the execution path in each iteration, resulting in additional code coverage in p. We use this as feedback to our mutation engine, and mutations will be directed toward improving the code coverage.

Handling Crashes. We consider all crashes in the program code as true crashes. Given a crash, we minimize the crash causing FIList by identifying the minimal faults that also lead to the same crash. Finally, we perform a lightweight root cause analysis to identify the API errors at the corresponding locations in p that resulted in the crash.

Crash Filtering. As we mention in Section 3.2.5, crashes in the library could be false positives because fault injection could violate certain program constraints. We develop a filtering technique that discards false positive crashes by performing a lightweight analysis on execution traces.

For our example in Listing 1, we first run `apvlv` with a simple PDF file (`test.pdf`) and `libpoppler`$_f$ (*i.e.,* FILib) in record-only mode. This initial run gave us RFList with 54 entries. Next, we repeatedly executed `apvlv` with `test.pdf` and `libpoppler`$_f$ and every time we provided a different FIList with 54 bits (same size as RFList), which we generate through coverage guided mutations. Our crash filtering was able to filter out 25.68K false positive crashes resulting in 1,099 true crashes, including the heap overwrite in Listing 1. Finally, our crash minimization and root cause identification found that the crash is because of a fault injected into `poppler_document_get_page` at line 33.

4.2 Design

4.2.1 Creating Fault Injectable Library (FILib). This is the first phase that works on a given library's source code. Here the goal is to generate a fault injectable version of the given library l.

a) Identifying FIPs. Based on existing works [39, 49, 51, 58, 70], we identified a set of source idioms listed in Table 2 that indicate that the corresponding program path encountered an invalid condition. We call these *error markers*. Given the source code of the target library, first, we get Abstract Syntax Trees (ASTs) of all the functions defined in it. Second, we go through the list of error markers in Table 2 to see if any of them is present in a function's AST. Third, given a matched error marker, we identify the immediate control dependency conditional statement [25] and consider that as an FIP. We also note whether the error marker is in the true or false branch of the conditional statement. In the case of `switch` statements, we note the value of the corresponding `case` clause. This immediate control dependent conditional statement is considered as an FIP, and we create a corresponding FIP record (Section 3.2.1). We call the path from a FIP to the corresponding error marker an *error handling path.*

The Figure 2 shows examples demonstrating our FIP identification technique. The red blocks show Basic Blocks (BBs) in Control Flow Graph (CFG) of various functions containing an error marker. The light-colored blocks show the error markers' control dependency blocks, and we only *select the immediate control dependency as a FIP.* An example of error marker (★) and corresponding control dependent (⊗) conditional statements and the FIP in `libpng` is shown below:

```
static store_palette_entry *
store_current_palette(png_store *ps, int *npalette)
{
 ...
 ⊗if (...) {
   ⊗if (ps->current == NULL) <- FIP
   {
     ...
     store_log(...);
     ★return NULL;
 ...
}
```

In the above example, we only consider the inner `if` statement (*i.e.,* immediate control dependency) as a FIP. We do not consider the non-immediate control dependent conditional statements (*e.g.,* outer `if`) as FIPs because they do not exclusively control the entry to the error marker and consequently might be related to the program functionality. In the end, we collect all FIP records in a `json` file (*i.e.,* FIP$_l$), which becomes an input to the next instrumentation step.

b) Instrumenting FIPs. We instrument each FIP as shown in Table 3. For switch-based FIP, our instrumentation will force the execution along the case containing the error marker. Our instrumentation enables us to have *a common fault injection mechanism irrespective of the type of FIP.* Specifically, we can inject fault by making the call to `has_fault` return `1`. This will cause the execution to reach an error marker, irrespective of where it is (*i.e.,* true or false branch or case label). For instance, if the error marker is in the false branch, our instrumentation modifies `original_condition` as `!has_fault(<FID>) && original_condition`. Here, if `has_fault` returns `1`, execution will be forced along the false branch irrespective of `original_condition` and eventually reaches the corresponding error marker. The same reasoning applies to all other types of FIPs.

The `has_fault` function is part of our helper library (*ferrlib*), which handles several aspects of our fault injection. This additional level of indirection through *ferrlib* enables us to modify our fault injection logic without re-instrumenting l. We also have a *record-only*

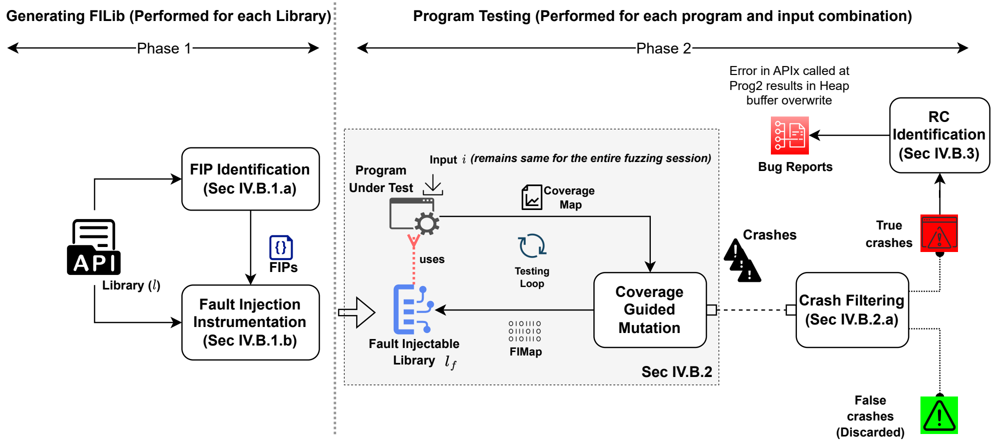
Generating FILib (Performed for each Library)
Phase 1
Program Testing (Performed for each program and input combination)
Phase 2
API
Library (l)
FIP Identification (Sec IV.B.1.a)
FIPs
Fault Injection Instrumentation (Sec IV.B.1.b)
Program Under Test
Input i (remains same for the entire fuzzing session)
Coverage Map
uses
Testing Loop
Fault Injectable Library l_f
FIMap
Coverage Guided Mutation
Sec IV.B.2
Crashes
Crash Filtering (Sec IV.B.2.a)
True crashes
False crashes (Discarded)
RC Identification (Sec IV.B.3)
Bug Reports
Error in APIx called at Prog2 results in Heap buffer overwrite

Figure 1: Overview of FuzzERR.

Table 2: Description of various code idioms used as error markers in functions of corresponding type (Section 4.2.1).

Function Spec	Marker	Description

Function return type is pointer	`return NULL;`
	`return 0;`
		The function returns a NULL pointer, an invalid address.

Function return type is integer	`return <negative_number>;`	The function is returning a negative number. Which are commonly used to indicate the error status.

Function return type is void	{
	`return;`
	}
	large number of statements
		The function is returning abruptly while the other
		mutually exclusive branch contains most of the function's code.

All functions	`goto <error_label>;`	The execution is abruptly directed to an arbitrary location. A common pattern used to handle error conditions in system's code [64].
	`exit(..),abort(..)`	These functions are used to terminate an execution. Common pattern used to handle unrecoverable errors.
	`throw <exception>`	The function is throwing an exception, a commonly used paradigm to communicate error conditions.

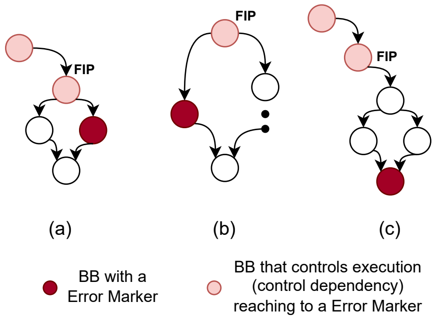
FIP
FIP
FIP
(a)
(b)
(c)
BB with a Error Marker
BB that controls execution (control dependency) reaching to a Error Marker

Figure 2: Examples demonstrating FIP identification technique.

Table 3: Instrumentation of FIPs based on the edge leading to an error marker. Here, `FID` is the id of the corresponding FIP.

Error Marker Edge	Original Code	Instrumentation
true	`<original_cond>`	`has_fault(<FID>) || <original_cond>`
false	`<original_cond>`	`!has_fault(<FID>) && <original_cond>`

val (switch case)	`switch(var) {`
	`case val:`
	`}`
		`if (has_fault(<FID>)){`
		`var = val;`
		
		`}`
		`switch(var) {`
		`case val:`
		`}`

mode in *ferrlib*, which can be enabled or disabled through an environment variable. In this mode, no faults will be injected *i.e.*, `has_fault` will always return 0, and in addition, all calls to `has_fault` will be logged.

4.2.2 Program Testing Through Fault Injectable Library. This is the second phase of FuzzERR, and it involves testing programs that use the target library l. For a given program, we first collect a set of valid inputs from pre-existing test suites and augment them with automated test-generation techniques. Next, we configure the program to load l_f (*i.e.*, our instrumented version) instead of l. We achieve this by modifying the `RPATH` [68] in the header of the program executable to include the file path containing l_f. For each valid input i to the program, we perform the following three: (i) First, we execute the program with i by enabling record-only mode in l_f. This gives us RFList, *i.e.*, the ordered list of FIPs reached during execution. (ii) Next, we create an initial no-op FIList as a file that contains |RFList| number of 0 bits. (iii) Finally, we continuously run the program with the same i for a predefined time. In each run, we provide a new FIList generated through various mutation strategies starting from the initial no-op FIList. In every run, we also keep track of any additional code covered in the program and pick mutations that are likely to improve code coverage and trigger bugs. When a crash occurs, we use our crash filtering mechanism (as will be explained in Section 4.2.2) to check whether it is true or false and provide positive or negative feedback to our FIList generation mechanism.

a) Filtering Infeasible Crashes. Our crash filtering mechanism tries to filter out crashes caused because fault injection violated certain program invariants. An example of a false positive crash is shown in Section 3.2.5. However, precisely identifying whether a fault violated certain program invariants requires analyzing interprocedural data dependencies, which is a known hard problem [81]. Furthermore, we want the filtering mechanism to be fast because it will be used in-line during testing iterations. We propose a lightweight mechanism based on program traces. As shown in Figure 3, there are three possible crash scenarios because of fault injection.

- *Program Crashes (Scenario 1):* In this scenario, the crash occurs in the program code. We expect the program to handle all possible API error cases. However, a crash in program code because of a fault injected in the library indicates that the program failed to handle certain error cases of an API. Hence we consider *these crashes as true* and resulting from a potential improper error handling bug.
- *Library Crashes (Scenario 2 and 3):* In this case, the crash occurs in the library code. As shown in Figure 3, there are two possible scenarios (2 and 3) on how such crashes can occur. In Scenario 2, the most recent fault was injected in the same library context as the crash. This likely indicates that the fault violated certain invariants, and hence *we consider these as false positives.* In Scenario 3, the most recent fault injection is in a different context than the crash. Here, the execution flows through the program, indicating that an API error is propagating through the program. This most likely indicates that the program is passing certain error data from an API call to another. Hence, *we consider these as true crashes.* For instance, consider that we injected a fault in `fopen` call, which returned `NULL`. Next, the program calls `fwrite` using the return value without checking whether it is `NULL`. However, `fwrite` expects the file pointer argument to be non-NULL. This results in a crash (`SIGSEGV`) in `fwrite` (*i.e.,* library), which is a true crash, revealing an API (*i.e.,* `fopen`) error handling bug in the program.

We use runtime stack trace as the context for fault injection and crashes. We install a signal handler as part of our helper library, *i.e., ferrlib.* When a crash occurs, our signal handler gets triggered, which uses the stack trace to check if the crash occurred in the program or library code. If the crash occurs in library code, we compare the stack trace of the most recent fault injection with the crash's stack trace. If they are the same, we consider the crash as a false positive and discard it. The crash is considered true in all other cases, and corresponding FIList will be stored for further processing. While the crash filtering approach can miss out on certain kinds of API misuses and hence is unsound. However, as we show in Section 5.4, the intuition described above works well for a large number of API misuse bugs.

4.2.3 FIList minimization and Root Cause Identification.

The goal here is to find the minimal combination of faults (*i.e.,* a combination of 1 bits) in FIList that cause the crash and represent a possible root cause of it. However, finding the minimal combination of 1 bits in a given FIList is a combinatorial problem.

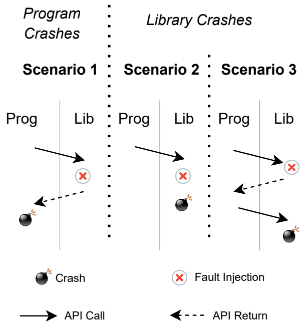

Figure 3: Summary of different crash scenarios.

To handle this, we propose a delta debugging [7] technique based on simulated annealing [11] to reduce FIList. The goal here is to minimize (*i.e.,*reduce the number of 1 bits) a crash causing FIList, which still causes the crash. Our technique is iterative, and it works as follows:

In every iteration, given the current $FIList$ of length n bits, we create a reduced $FIList_{new}$ by setting each bit $i \in [1, n]$ based on the following equation:

$$FIList_{new}(i) = \begin{cases} 1 & \text{if } FIList(i) = 1 \wedge rand(0,1) \geq C \\ 0 & \text{otherwise} \end{cases}$$

Here, $rand(0, 1)$ generates a random floating point value in $[0, 1]$ range, and C represents the cooling factor and is also a floating point value within the same range. The generated $FIList_{new}$ will be of the same length n as the previous $FIList$. However, the number of 1 bits in $FIList_{new}$ will be less than or equal to those in $FIList$. The cooling factor C controls the reduction rate – a higher value indicates a lesser reduction rate.

After every iteration, we check if the newly reduced $FIList_{new}$ still causes the crash. If yes, then $FIList_{new}$ is the newly reduced $FIList$ and will be used in subsequent iterations. We will also decrease the value of C by s (*i.e.,* cooling schedule) and consequently increase the reduction rate for subsequent iterations.

If $FIList_{new}$ does not cause the crash, we will retain the previous $FIList$ for subsequent iterations. We will also increase the value of C by s and reduce the reduction rate for subsequent iterations.

This process continues for τ iterations, after which the latest $FIList$ will be considered as the final reduced $FIList$.

RC Identification. Given the minimized FIList, we use the stack trace of fault injection points to identify the source location of corresponding API calls in the program. These source locations will be used to form our error report. For instance, "*The errors in APIx called at Line 23, and APIy called at Line 43 results in a buffer overflow at Line 145.*". We wrote an additional helper script that uses the stack trace and the kind of crash to identify the unique bugs amongst the true crashes.

4.2.4 Implementation Details.

We used Clang/LLVM compiler framework version 12 to implement our analysis components. Our library instrumentation is implemented as a LLVM pass. We get the overall bitcode file of the target library by using wllvm [71], and run our instrumentation pass on the generated bitcode file. Our mutation and testing techniques are implemented by modifying AFL++ using its post-processor support. Our helper library, *ferrlib*, is implemented in C and provides various knobs to control different aspects of fault injection. The crash filtering/root-cause identification technique is implemented as a python module, which our modified AFL++ will use during every crash. In total, our implementation involves 6.1K lines of C++ code and 5.3K lines of python code.

5 EVALUATION

We pose the following research questions to guide our evaluation of FuzzERR:

- *RQ0: Effectivess of FIP Identification.* How effectively does our technique identify FIPs?
- *RQ1: Effectiveness of* FuzzERR. How effective is FuzzERR in finding API error handling bugs? What is the contribution of each of our techniques?
- *RQ2: Impact of Code Coverage.* Does coverage guidance improve the effectiveness of FuzzERR? and does FuzzERR help increase code coverage?
- *RQ3: Comparison against the state-of-the-art.* How does FuzzERR perform in comparison with the state-of-the-art techniques?

5.1 Dataset

Our goal is to collect a representative real-world dataset that enables us to effectively evaluate different components of FuzzERR. Our current implementation is based on Clang and consequently has the following restrictions on the dataset. (i) The library must be compilable using Clang. (ii) As with all dynamic techniques, these programs should be easy to set up and run, *i.e.,* not network servers or other programs that require complex setup. We scrapped the Debian package repository [28] and randomly picked 12 libraries that satisfy our requirements. For each of the libraries, using reverse search of `apt-utils` [37], we selected 1-3 programs that use API functions in the corresponding library, which were easy to fuzz. In total, we selected 20 programs. The first part of Table 4 shows the list of libraries and corresponding programs along with source-level statistics.

We also used APIMU4C [36], an existing API misuse bug dataset that contains various synthetic bugs in 3 programs. However, the dataset is targeted towards static tools and uses programs that are hard to fuzz, especially network servers such as `httpd`. Out of the three programs, only `openssl` was readily testable. The last row of Table 4 shows the details of the program.

5.1.1 Collecting Programs' Testcases.

5.1.1 Collecting Programs' Testcases. As explained in Section 4.2.2, FuzzERR executes a program repeatedly with a fixed test case and injects various faults in each run. We collect test cases for each program by using the corresponding programs' test suites. We further augment these by running AFL++ for 24 hours on each program with the initial inputs as seeds. The column **Num.T** of Table 4 shows the total number of test cases collected for each of the corresponding programs.

Table 4: Evaluation Dataset.

Libraries					Programs			
ID	**Name**	**Size (Loc)**	**APIs**	**FIPs**	**ID**	**Name**	**Size (Loc)**	**Num. T**
1	libelf	19.2K	25	151	1	eu-objdump	890	24
2	libpng	99K	55	97	2	contextfree	78.3K	24
					3	optipng	5.7K	24
3	libxml2	332K	162	2,982	4	xmllint	3.9K	24
					5	xgettext	84.4K	24
4	libzstd	112.5K	7	284	6	curl	178.8K	1
					7	plocate	5.6K	1
5	libpoppler	143.7K	30	139	8	pdftotext	11.5K	24
					9	apvlv	11.9K	24
6	libjpeg	85.7K	23	18	10	jpegoptim	2.5K	31
					11	jp2a	2.7K	36
					12	jpegqs	5K	19
7	libsqlite3	502K	56	381	13	lnav	178.9K	10
8	libavcodec	662.5K	10	6,778	14	shotdetect	2.1K	24
9	libavformat	225.4K	13	4,212	15	unpaper	4.6K	24
10	libavutil	71.7K	13	449	16	loudgain	2.6K	21
11	libcairo	234K	31	912	17	fntsample	1.1K	24
					18	duc	14.3K	9
12	libfreetype	182.2K	19	534	19	logstalgia	18.8K	20
					20	dvisvgm	234.9K	24
Total		2.669M	444	16,937	**Total**		848.4K	412
APIMU4C								
ID	**Name**	**Size (Loc)**	**APIs**	**FIPs**	**ID**	**Name**	**Size (Loc)**	**Num. T**
13	libcrypto	182K	1065	3688	21	openssl	295k	3

5.2 Experimental Setup

We ran all our experiments on a AMD EPYC 7543P CPU machine with 64 cores and 64GB memory. We run FuzzERR for 15 minutes on 30 cores in parallel mode, with instances sharing coverage. Specifically, for a given program and a test case combination, we run FuzzERR for 15 minutes on 30 cores. While it is recommended that experiments related to fuzzing be run for 24 hours, we did a preliminary experiment with more time on a subset of programs but noticed that coverage did not improve after 15 minutes. This is expected because we are only injecting faults on a fixed input. Hence, we fuzz each *(program, test case)* combination for 15 minutes. We run our minimization technique with initial cooling factor $C = 0.1$, reduction rate $s = 0.05$, and run for $\tau = 50$ iterations. These values provide the best reduction rate, as shown in Section 5.4.1.

5.3 Effectiveness of FIPs Identification

For a given library, the first step in FuzzERR is FIPs identification. As mentioned in Section 3.2.1, FIPs represent conditional statements that control execution into error handling code. The **FIPs** column in Table 4 shows the number of FIPs found in each of the corresponding libraries. On average, our technique found 1,411 per library with a total of 16,937.

To evaluate the accuracy of our technique, ideally, we need to manually verify all the FIPs found by it. However, given the large number of FIPs, we performed a random sampling. First, we randomly picked 400 FIPs across all libraries and manually checked whether each of these is a true FIP or not. Second, we randomly picked 200 functions and manually identified all FIPs in it, and then we checked if all of these FIPs were also found by our technique. Table 5 shows the results of our evaluation. Our technique is able to identify FIPs with a very high accuracy. This is expected because

Table 5: Accuracy of FIPs identification.

	Identified FIPs	
	True Positives	**False Positives**
Randomly Selected 400 FIPs	95%	5%
	FIPs Identified	**FIPs not Identified**
Randomly Selected 200 functions	91%	9%

our FIP identification is precise as it is based on a strict set of error markers (Section 4.2.1), which are mostly used in error-handling code. However, there are few false positives (*i.e.,* 5%). This is mainly because few APIs use error markers to indicate functional cases. For instance, return `NULL` to indicate the given array is already sorted. Although an error marker, this does not indicate an error but a valid return value and hence a false positive. However, as shown by the small percentage, such cases are minimal.

As shown by the false negative column, our technique has false negatives, *i.e.,* we missed identifying certain FIPs. The main reasons for this are: (i) Conditional compilation: Few FIPs were guarded by certain pre-processor directives that were not enabled in the default build configuration of the corresponding programs. Consequently, the pre-processor skipped these FIPs, although visible in the source code. (ii) Value dependencies: In a few cases, identifying error markers require value flow analysis [69]. For instance, in the snippet `if ((r = foo()) < 0) return r;`, although, according to our definition (Table 2), `return r` is an error marker as it is returning a negative number. However, identifying this requires value flow analysis, which we consider out of scope for our technique.

Missing FIPs does not greatly affect FuzzERR as it just reduces fault injection points and does not necessarily eliminate fault injection capability – as faults in an API can be injected through multiple FIPs.

5.4 RQ1: Effectiveness of FuzzERR

In this section, we evaluate the overall effectiveness of FuzzERR. The first group of columns in Table 7 shows the overall performance of FuzzERR on all the programs in the dataset. We have omitted programs (*e.g.,* `curl`) on which FuzzERR did not find any bugs.

The column **Filtered Crashes** shows the number of false positive crashes automatically filtered out by our lightweight filtering technique (Section 4.2.2). All programs have a large number of filtered crashes, highlighting one of the important problems with software fault injection. This also emphasizes the importance of having an automated filtering mechanism. The large number of filtered crashes is also because many mutated FILists will trigger the same false positive crash. We plan to improve this as part of our future work (Section 6).

Next, column **Final Crashes** shows crashes that were collected at the end of testing. Most of the programs have relatively few total crashes except for `xgettext` (ID: 5), `logstalgia` (ID: 19), and `dvisvgm` (ID: 20), which contain an unusually large number of crashes. However, the number of bugs corresponding to these crashes is relatively small. This large number of crashes is because a few API error bugs in these programs result in using uninitialized variables. An example of such a bug is shown in Listing 1. Depending on the value of these uninitialized variables, the program crashes at different locations, leading to a huge number of crashes for relatively few unique bugs. This large number of filtered crashes highlights one of the important problems with software fault injection and emphasizes *the importance of having an automated filtering mechanism.*

It is interesting to see that there are false positives in final crashes. This is because certain invariants *across API* calls are violated by fault injection. However, our crash filtering mechanism treats these as true crashes (Scenario 3 in Figure 3). But, these cases are relatively small, *i.e.,* ∼ 0.7% (42). Finally, our crash minimization and root cause identification technique mapped these true crashes to 31 bugs across various programs. These bugs represent unique locations in the corresponding program where an API error was wrongly handled.

As shown by the last row, out of the five bugs in API4MU dataset, FuzzERR found only three bugs. The main reason for missing the other two bugs is missing error markers (Table 2). Specifically, a few APIs (*with integer return type*) use value 0 to indicate an error (*i.e.,* `return 0`). However, we do not consider this an error marker and fail to identify FIPs and miss fault injections. We plan to address this in our future work (Section 6).

5.4.1 Sensitivity Study of FIList Minimization. We performed a sensitivity study to understand the performance of our technique better. We vary each of our parameters, *i.e.,* C, s, and τ, and measure how the minimization effectiveness varies.

The Figure 4 shows the results of this experiment. Each line represents the percentage of reduction when the corresponding parameter value is varied, as shown by the legend. Ideally, we want our technique to stabilize quickly (*i.e.,* should take less number of iterations to achieve high reduction). As shown by the red line, increasing s (*i.e.,* cooling schedule) helps in quick convergence and improves the reduction rate, but higher values rapidly modify the cooling factor and consequently decrease the reduction rate. As shown by the green line, similar behavior is exhibited by varying C (*i.e.,* cooling factor). However, as the blue line shows, increasing the number of iterations (τ) will increase the reduction rate. But increasing the number of iterations also increases the time. As shown by the topmost point of the green line, the parameter values that provide the highest reduction rate are $C = 0.1$, $s = 0.05$, and $\tau = 50$.

5.4.2 Types of Bugs found by FuzzERR. The Table 6 shows the categorization of bugs. Although most bugs (20, 64%) are NULL-ptr dereferences, the rest (11, 36%) are severe security vulnerabilities that can lead to arbitrary code execution. The Table 10 in Appendix shows the complete list.

Examples. The Listing 2 and 3 shows examples of the bugs found by FuzzERR. In Listing 2, the fault of API call to `jpeg_read_scanlines` at line 4 is not handled (*i.e.,* the return value is not checked). The failure of this API call does not initialize `jpg`, which is eventually used to compute `y` (line 16). A negative value of `y` results in an infinite loop (line 20) and, consequently, a buffer overwrite at line 23.

The Listing 3 shows a bug resulting from the Scenario 3 category crash (Figure 3). Here, similar to the previous example, the fault of API call to `avformat_write_header` at line 6 is not handled (*i.e.,*

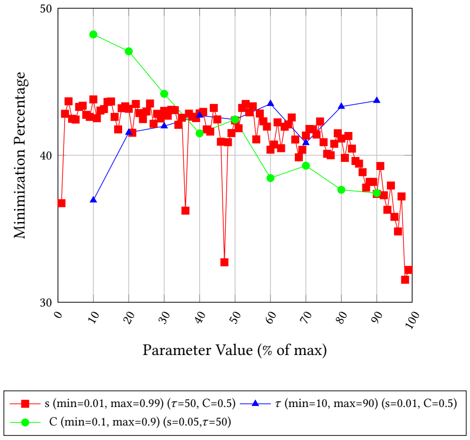

Figure 4: The impact of our minimization parameters on the FIList minimization averaged across 150 crashes.

Table 6: Categorization of bugs found by FuzzERR.

Bug Type	Num. of Bugs
Heap Buffer Overread	2
NULL-ptr dereference	20
Overallocation due to integer overflow	2
Floating Point Exception	1
Segmentation Fault	1
Use After Free	5
Total	31

the return value is not checked). This will cause `out_ctx` to be uninitialized. Subsequently resulting in NULL deference at line 21.

5.4.3 Responsible Disclosure. We reported these to the maintainers of the corresponding applications along with patches for 28 bugs, out of which 20 are already accepted. We are still working on the patches for the rest (3) of the bugs, as they require understanding application logic to gracefully handle API failures.

5.5 RQ2: Impact of Fault Injection on Code Coverage

To study the effect of coverage guidance in fault injection, we configured FuzzERR for random fault injection, *i.e.,* the FIList will be generated randomly in every iteration ($FuzzERR_{rand}$). We tested on $FuzzERR_{rand}$ on four programs of varying sizes and repeated the experiment multiple times. These programs were selected because they used the libraries covering characteristic representative functions (pdf, audio, images). $FuzzERR_{rand}$ could identify only 2 out of 7 API Misuse bugs identified by FuzzERR. This shows that coverage-guided considerably contributes to the effectiveness of FuzzERR.

As mentioned in Section 4.2.2, we run FuzzERR for each program and a test case combination. We repeatedly run the program with the same test case, but different faults will be injected into the target library (through FIList). For each test case, we measured the amount of additional code covered because of our fault injection.

Table 7: Performance of FuzzERR **in comparison with** AFL++ (*FL*) **and** FairFuzz (*FF*)**.**

Prog Id	FuzzERR Performance					Bugs also found by	
	Filtered Crashes	Final Crashes			Unique Bugs	FL	FF
		False	True	Total			
5	25.68K	10	1,099	1,109	2	0	0
9	11.04K	0	106	106	1	0	0
10	4.18K	3	20	23	1	1	0
11	58.27K	0	2	2	1	0	0
12	0	0	2	2	1	0	0
14	55.65K	4	35	39	11	1	0
15	8.55K	3	27	30	3	0	0
16	215.68K	3	42	45	3	0	0
19	7.61K	5	1,955	1,960	1	1	0
20	11.59K	7	2,334	2,341	2	1	1
21	7.5K	7	3	10	3/5	0	0
Total	673.79K	54	5,838	5,892	34	3	1

The Figure 5 displays a box plot of the additional code covered by various test cases for programs that FuzzERR has detected bugs.

Except for a couple of programs (*i.e.,* ID: 10 and 11), the additional code covered because of fault injection is relatively less (≤ 6%). But still FuzzERR was able to find bugs in these programs. This shows that *fault injection in libraries does not necessarily improve program code coverage but helps in exploring interesting program states*, as demonstrated by the several bugs found by FuzzERR. An example of this can be seen in Listing 3. Here, fault injection did not cover additional code but explored an interesting program state by not initializing `out_ctx`.

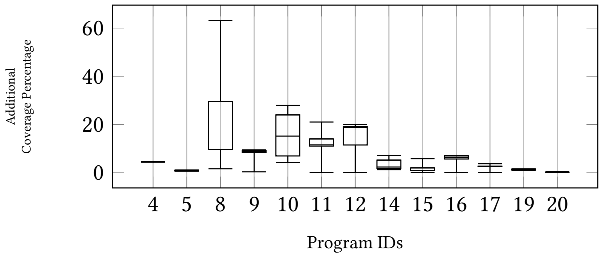

Figure 5: Additional code covered because of fault injection by FuzzERR**.**

5.6 RQ3: Comparative Evaluation

We selected the following state-of-the-art techniques for our comparative evaluation.

- AFL++ [26] (*FL*): This is a re-engineered fork of the popular coverage-guided fuzzer AFL [79]. Furthermore, AFL++ incorporates techniques from several other fuzzing tools, such as REDQUEEN [8], AFLSmart [62], and MOPT [50].
- FairFuzz [43] (*FF*): This is a fuzzing tool that uses novel mutation techniques to generate inputs effective at exploring error handling code.
- FiFuzz [38]: This is a fuzzing tool that uses a context-sensitive SFI approach, in order to cover error handling code in different

Listing 2 A real bug found by FuzzERR in `jp2a` because of incorrect handling of `jpeg_read_scanlines` API return value.

```
// file: jp2a/src/image.c
// ----------------------
  // A fault in this API does not initialize jpg.
  ⛓ jpeg_read_scanlines(&jpg, buffer, 1);
  // this uninitialized jpg is passed as argument
  process_scanline_jpeg(&jpg, buffer[0], &image);

void
process_scanline_jpeg(
  const struct jpeg_decompress_struct *jpg,
  const JSAMPLE* scanline,
  Image* i
){
  ...
  // y becomes negative since jpg->output_scanline was 0
  const int y = ROUND(i->resize_y *
      (float) (jpg->output_scanline-1) );
  ...
  // loops infinitely since y is negative
  while (lasty <= y) {
    ...
   // Buffer overwrite of pixel.
   🐞 pixel[x] += adds>1 ? v / (float) adds : v;
   ...
  }
  ...
}
```

Listing 3 A NULL pointer dereference bug found by FuzzERR in `unpaper` because of incorrect handling of `avformat_write_header` API return value.

```
// file: unpaper/file.c
// --------------------
  AVFormatContext *out_ctx;
  ...
  // A fault in this API does not initialize out_ctx.
  ⛓ avformat_write_header(out_ctx, NULL);
  ...
  // flow continues with the out_ctx uninitialized
  av_write_frame(out_ctx, &pkt);
  // this function internally calls other functions,
  // which finally call compute_muxer_pkt_fields
    ....> compute_muxer_pkt_fields(out_ctx, st, &pkt)
// file: ffmpeg/libavformat/mux.c
// ------------------------------
static int
compute_muxer_pkt_fields(
  AVFormatContext *s, AVStream *st, AVPacket *pkt){
  ...
  // st->internal->priv_pts is derived from s and
  // it will be NULL.
  🐞 pkt->dts = st->internal->priv_pts->val;
  ...
}
```

contexts, with the aim of finding bugs in error handling code with complicated contexts.

These tools represent the state-of-the-art in two categories – first, tools that aim to increase code coverage by exploring rare branches (AFL++ and FairFuzz) and second, tools to increase code coverage by directing execution towards error handling code (Fifuzz). As suggested by the recent work [41], we tested each program for 24 hours to normalize the effects of randomness.

5.6.1 Comparison with AFL++ *and* FairFuzz. The last two columns of Table 7 show this experiment's results on programs that FuzzERR has detected bugs. AFL++ found 3 bugs of those found by FuzzERR, whereas FairFuzz found only 1 bug. On APIMU4C dataset, neither AFL++, nor FairFuzz found bugs. This is expected because the above tools are coverage guided and focus on improving the code coverage. However, the bugs found by FuzzERR require a thorough state exploration rather than covering additional code (as shown in Figure 5).

Table 8: Performance of FuzzERR in comparison with FIFUZZ.

Prog Id	Bugs Found by				FIFUZZ Crashes		
	FuzzERR		FIFUZZ				
	Exclusive	Total	Exclusive	Total	False	True	Total
10	1	1	0	0	9	0	9
11	1	1	0	0	0	0	0
12	1	2	3	4	0	4	4
15	3	3	1	1	0	10	10
16	1	3	0	2	20	5	25
21	0	3	0	3	159	9	168
Total	7	13	4	10	188	28	216

5.6.2 Comparison with Fifuzz. We also compared FuzzERR with Fifuzz [38], a recent tool that also uses SFI to test the error handling code of applications. We want to emphasize that Fifuzz is not designed to find API misuse bugs but to improve code coverage by forcefully executing error-handling code. As a consequence of this, Fifuzz might find API error-handling bugs. Unfortunately, the tool binary or its source code is not available, and we also failed to get any response from the authors. We re-implemented Fifuzz by following instructions from the paper. As explained in the original paper [38], Fifuzz error point detection is specialized for C programs. To ensure a fair comparison, we ran Fifuzz only on C programs (as described in Sec 8 of the original paper [38]) and ran Fifuzz for 24 hours on each application (as suggested in 5.3 of the original paper [38]). The Table 8 shows the results of comparing FuzzERR with Fifuzz on only C programs. In total, FuzzERR found thirteen bugs, three more than Fifuzz, and interestingly, FuzzERR exclusively found seven out of ten bugs. The Listing 3 shows a bug exclusively detected by FuzzERR but not Fifuzz. However, Fifuzz also exclusively found four bugs that were missed by FuzzERR. We found that *none of these bugs are because of improper API error handling but rather in the clean-up code.* Furthermore, on Prog 16, Fifuzz resulted in 20 (80%) false positive crashes, which required considerable manual effort to filter them out. On APIMU4C dataset (Prog Id 21), Fifuzz was able to find all three found by FuzzERR. However, Fifuzz resulted in 159 (94%) false positive crashes. Table 9 shows the coverage obtained by FuzzERR and Fifuzz in these programs. The coverage for Fifuzz is over a run of 24 hours, whereas the coverage for FuzzERR is over a run of 6 hours (as mentioned in Section 5.2, increasing the time does not affect coverage). Despite the lower code coverage obtained by FuzzERR, it could identify more bugs than Fifuzz, further confirming our observations in Section 5.5. This shows that FuzzERR is more effective at exploring error handling code and triggering API misuse bugs.

In summary, FuzzERR is more effective than the state-of-the-art tools in finding API error-handling bugs by exploring interesting program states through fault injection.

6 LIMITATIONS AND FUTURE WORK

We present the limitations of the current implementation of FuzzERR, along with our future plans to address them:

Table 9: Comparision of coverage obtained by FIFUZZ and FuzzERR

Prog Id	FIFUZZ coverage (24 hrs)	FuzzERR coverage (6 hours)
10	28.76%	26.71%
11	23.22%	20.72%
12	11.13%	7.69%
15	7.04%	34.98%
16	11.79%	5.65%

- *Compiler Dependency.* We require the target library to be compilable with CLANG as our implementation (Section 4.2.4) is based on it. Few libraries, such as `glibc`, require considerable effort to be compilable with CLANG [54]. Consequently, FUZZERR cannot be used to find the error handling bugs of corresponding APIs.
- *Dynamic Analysis and Testcase Requirements.* FUZZERR benefits from the availability of an exhaustive set of test cases. Generating effective test cases is becoming relatively easy with the advances in automated testing techniques. We envision that FUZZERR will be used as an additional step on top of existing automated testing techniques to effectively explore API error handling code.
- *False Negatives.* The current design choices of FUZZERR are focused on precision and can potentially miss certain bugs, resulting in false negatives. There are two cases where this can happen. (I) We use a pre-defined set of pre-defined error markers to identify FIPs. These markers may be missing in certain libraries. (ii) Our filtering mechanism uses stack traces to check that the fault injection point is different from the crash point for library crashes and discards them (if same) (Section 4.2.2). However, there can be cases when the stack trace is the same, even when both are different (*e.g.,* loops). As part of our future work, we will improve our FIP identification using learning techniques and extend our filtering mechanism to perform additional post-processing.

7 RELATED WORK

Software Fault Injection (SFI). SFI techniques [22, 27, 80] are shown to be very effective in testing error-handling code. Bai *et al.,* [9, 10] developed techniques to inject faults into device handling functions while testing device drivers to detect bugs in error handling code. LFI [52] is one of the first works that try to inject faults in library functions and see how the program behaves. However, LFI mainly focuses on returning the failure return value without capturing the entire error semantics of library functions. *e.g.,* clearing or freeing the pointers passed through arguments. This can lead to false positives or infeasible crashes. FUZZERR avoids this by forcing execution to error blocks, thereby leveraging existing error semantics.

False positives or reaching infeasible states [40, 55] is one of the known problems with SFI. Another problem with existing SFI techniques is location-based injection rather than execution based. Recent work, FIFUZZ [38] tries to avoid this by using context-sensitive fault injection, wherein they use the combination of error location information along with the execution context as a fault injection point. FUZZERR is more precise, and we use a filtering mechanism to avoid any resulting false positive crashes. Furthermore, as shown in Table 8, FUZZERR is more effective at finding API error-handling bugs than FIFUZZ. Software faults can also be mimicked by directly mutating the program, commonly called mutation analysis [4] or mutation testing. There exist techniques, such as SlowCoach [20], to generate mutations [59]. FUZZERR is similar in principle to mutation testing, where targeted mutations are performed in libraries.

API Misuse Detection. Most of the work [36] on API misuse bug detection is based on static approaches. Some of these techniques [17, 31, 35] check for violations of API usage rules. But techniques to automatically generate API usage rules [3, 13, 42, 45, 47, 57, 60] are hard to scale and require a large corpus of valid API uses. There are other techniques based on Anomaly Detection (AD) [16, 24]. The recent work APISAN [78] encodes common patterns as semantic beliefs and looks for violations of these beliefs. There are also machine learning-based techniques [14, 46, 48, 61, 72, 74, 75], such as the recent work FICS [6], that cluster API usage patterns and consider minority clusters as API misuses. These static automated techniques have a lot of false positives. For instance, ARBITRAR [44], APISAN [78] and FICS [6] have false positive rates of 51.5%, 87.9% and 88%, respectively. These techniques are impractical for real use [12, 21].

Fuzzing. Generational fuzzers [19, 33, 73, 76] require a specification, and inputs will be generated based on the specification. These techniques are helpful for testing programs that expect well-structure input, such as compilers [19], interpreters [73] and device drivers [23]. Mutational fuzzers [8, 15, 26, 29, 34, 43, 63, 77] generate inputs by performing mutations on a given set of seed inputs. There are other techniques, such as Driller [67] and Angora [18], that use hybrid approaches of combining mutations with symbolic execution.

Most of the existing fuzzing techniques use code coverage [66] as the feedback. In our work, we use an existing coverage-guided mutational fuzzer, *i.e.,* AFL++ [26], as our base and use it to direct fault injection.

8 CONCLUSION

We present FUZZERR, a generic dynamic analysis approach to detect API error handling bugs based on coverage-guided software fault injection. We inject faults into APIs by forcing execution along error paths and then observe how the program under test handles these failures. We also filter out false positives by using a lightweight technique based on program traces. Our evaluation shows that FUZZERR found 31 new and previously unknown bugs resulting from incorrect handling of API errors, significantly outperforming the state-of-the-art.

ACKNOWLEDGMENTS

This research was supported by in part by the National Science Foundation (NSF) under Grants CNS-2247686, Amazon Research Award (ARA) on "Security Verification and Hardening of CI Workflows" and by Defense Advanced Research Projects Agency (DARPA) under contract number N6600120C4031 and N660012224037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, Amazon or the United States Government

REFERENCES

[1] 2023. AWS IOT Device SDK: always false condition. https://github.com/aws/aws-iot-device-sdk-embedded-C/pull/1861/commits/f312ea3eaaec04d09cfbe4fda0daf89e995da454

[2] 2023. Nvidia open-gpu kernel modules: always false condition. https://github.com/NVIDIA/open-gpu-kernel-modules/pull/493/commits/527cb9d50c42a3fa91946e67eaa085f8eeb71ce5

[3] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (FSE)*. 25–34.

[4] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1979. *Mutation Analysis.* Technical Report. Georgia Inst of Tech Atlanta School of Information And Computer Science.

[5] Hussien Al-haj Ahmad, Yasser Sedaghat, and Mahin Moradiyan. 2019. LDSFI: a Lightweight Dynamic Software-based Fault Injection. In *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*. 207–213. https://doi.org/10.1109/ICCKE48569.2019.8964875

[6] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. 2021. Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In *30th USENIX Security Symposium (USENIX Security 21)*. 2025–2040.

[7] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[9] Jia-Ju Bai, Yu-Ping Wang, Hu-Qiu Liu, and Shi-Min Hu. 2016. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology* 73 (2016), 122–133.

[10] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. 2016. Testing error handling code in device drivers using characteristic fault injection. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 635–647.

[11] Dimitris Bertsimas and John Tsitsiklis. 1993. Simulated annealing. *Statist. Sci.* 8, 1 (1993), 10–15.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[13] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422.

[14] Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai. 2018. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering* 45, 10 (2018), 984–1001.

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043.

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* 41, 3 (2009), 1–58.

[17] Hao Chen and David Wagner. 2002. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*. 235–244.

[18] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[19] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 197–208.

[20] Yiqun Chen, Oliver Schwahn, Roberto Natella, Matthew Bradbury, and Neeraj Suri. 2022. SlowCoach: Mutating Code to Simulate Performance Bugs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. 274–285. https://doi.org/10.1109/ISSRE55969.2022.00035

[21] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st ACM/IEEE international conference on Automated Software Engineering*. 332–343.

[22] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. 2015. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 361–372.

[23] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138.

[24] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.

[25] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[27] Chen Fu, Barbara G Ryder, Ana Milanova, and David Wonnacott. 2004. Testing of java web services for robustness. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. 23–34.

[28] José Angel Galindo, David Benavides, and Sergio Segura. 2010. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis.. In *ACoTA*. 29–34.

[29] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[30] GitHub. 2017. Discover vulnerabilities across a codebase with CodeQL. https://securitylab.github.com/tools/codeql.

[31] GitHub. 2017. Semmle - A code analysis platform for finding zero-days and automating variant analysis. https://semmle.com/.

[32] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[33] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 206–215.

[34] Google. 2022. Honggfuzz: A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer. https://github.com/google/honggfuzz.

[35] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, and Jiaguang Sun. 2019. Vetting api usages in c programs with imchecker. In *2019 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 91–94.

[36] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. 2019. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 11–20. https://doi.org/10.1109/COMPSAC.2019.00012

[37] Red Hat. 2010. Advanced Package Tool. http://apt-rpm.org/scripting.shtml.

[38] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *29th USENIX Security Symposium (USENIX Security 20)*. 2595–2612.

[39] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated inference of error specifications for C APIs. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*.

[40] Nobuo Kikuchi, Takeshi Yoshimura, Ryo Sakuma, and Kenji Kono. 2014. Do injected faults cause real failures? a case study of linux. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 174–179.

[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138.

[42] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 161–176.

[43] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[44] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1400–1415.

[45] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[46] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[47] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. *ACM Sigplan Notices* 44, 6 (2009), 75–86.

[48] Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 296–305.

[49] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Computer Security – ESORICS 2019*, Kazue Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer International Publishing, Cham, 3–25.

[50] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[51] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1562–1579.

[52] Paul D. Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 379–388.

[53] Paul D. Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 379–388.

[54] MaskRay. 2022. Challenges in Building glibc with clang. https://maskray.me/blog/2021-09-05-build-glibc-with-lld.

[55] Roberto Natella, Domenico Cotroneo, Joao Duraes, and Henrique Madeira. 2010. Representativeness analysis of injected software faults in complex software. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 437–446.

[56] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. 2016. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–55.

[57] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 166–177.

[58] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1203–1218.

[59] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[60] Jian Pei, Haixun Wang, Jian Liu, Ke Wang, Jianyong Wang, and Philip S Yu. 2006. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering* 18, 11 (2006), 1467–1481.

[61] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 426–437.

[62] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[63] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[64] John Regehr. 2013. Use of Goto in Systems Code. https://blog.regehr.org/archives/894

[65] Harold A Rosenberg and Kang G Shin. 1993. Software fault injection and its application in distributed systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 208–217.

[66] Christopher Salls, Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Exploring abstraction functions in fuzzing. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–9.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[68] Milan Stevanovic. 2014. Locating the Libraries. In *Advanced C and C++ Compiling*. Springer, 115–135.

[69] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[70] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*.

[71] travitch. 2015. Whole Program LLVM. https://github.com/travitch/whole-program-llvm.

[72] A Inkeri Verkamo. 1994. Efficient Algorithms for Discovering Association Rules. In *KDD Workshop*.

[73] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[74] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 797–812.

[75] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS)*. 499–510.

[76] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 283–294.

[77] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[78] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing {API} Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. 363–378.

[79] Michal Zalewski. 2018. AFL Technical Details. https://lcamtuf.coredump.cx/afl/technical_details.txt

[80] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 595–605.

[81] Feng Zhu and Jinpeng Wei. 2014. Static analysis based invariant detection for commodity operating systems. *Computers & Security* 43 (2014), 49–63.

[82] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

A LIST OF BUGS

Table 10: List of bugs identified by FuzzERR

S.No.	Library	Program	Type	Source File	Line Number	Remarks
1	libjpeg	jpegqs	Null ptr deref	quantsmooth.c	530	Return value of jpeg_read_coefficients() not checked. It can be NULL in certain conditions.
2	libxml2	xgettext	Null ptr deref	locating-rule.c	92	Return value of xmlGetProp() not checked. It can be NULL in certain conditions.
3	libxml2	xgettext	Null ptr deref	locating-rule.c	313	Return value of xmlDocGetRootElement() not checked. It can be NULL in certain conditions.
4	libavutil	loudgain	Null ptr deref	scan.c	203	Return value of swr_alloc() not checked. It can be NULL in certain conditions.
5	libavutil	loudgain	Null ptr deref	scan.c	442	Return value of av_malloc() not checked. It can be NULL in certain conditions.
6	libavutil	loudgain	overallocation due to integer overflow	scan.c	438	Return value of av_samples_get_buffer_size() not checked (negative error code in case of failure). This is passed as an argument to av_malloc(). This value can be negative in case of error, which would lead to overallocation.
7	libfreetype	dvisvgm	use-after-free	FontEngine.cpp	224	Return value of FT_Load_Glyph() not checked, leading to use-after-free bugs in certain conditions.
8	libfreetype	dvisvgm	use-after-free	FontEngine.cpp	233	
9	libfreetype	dvisvgm	use-after-free	FontEngine.cpp	244	
10	libfreetype	dvisvgm	use-after-free	FontEngine.cpp	253	
11	libfreetype	dvisvgm	use-after-free	FontEngine.cpp	262	
12	libfreetype	dvisvgm	Null ptr deref	Font.cpp	197	Return value of FontEngine::setFont() not checked (false on failure). In case its unsuccessful, it can lead to a null-pointer-dereference later in the code.
13	libfreetype	dvisvgm	Null ptr deref	Font.cpp	239	
14	libfreetype	logstalgia	Null ptr deref	fxfont.cpp	81	Return value of FT_Glyph_To_Bitmap() is not checked (non-zero return on error), which can lead to null-pointer-dereference later.
15	libavcodec	shotdetect	Null ptr deref	main.cc	216	Retrun value of film::process() is not checked (negative return on error). A failure in film::process() can lead to a null-pointer-dereference later.
16	libavcodec	shotdetect	Null ptr deref	graph.cpp	300	data.size() can be 0 in certain conditions, which would leade to integer underflow in the loop condition. This would eventually lead to a null-ptr-dereference.
17	libavcodec	shotdetect	Null ptr deref	film.cpp	295	Return value of av_frame_alloc() not checked, which will lead to null-ptr-dereference later.
18	libavcodec	shotdetect	Null ptr deref	film.cpp	296	Return value of av_frame_alloc() not checked, which will lead to null-ptr-dereference later.
19	libavcodec	shotdetect	Null ptr deref	film.cpp	297	Return value of av_frame_alloc() not checked, which will lead to null-ptr-dereference later.
20	libavcodec	shotdetect	overallocation due to integer overflow	film.cpp	302	Return value of avpicture_get_size() not checked (non-zero return on error). This can lead to integer underflow in the argument passed to malloc() later.
21	libavcodec	shotdetect	Null ptr deref	film.cpp	304	Return value of malloc() is not checked, which can lead to null-ptr-dereference later.
22	libavcodec	shotdetect	Null ptr deref	film.cpp	305	
23	libavcodec	shotdetect	Null ptr deref	film.cpp	310	Return value of avpicture_fill() not checked (negative return on error). Failure in avpicture_fill() can later lead to null-ptr-dereference.
24	libavcodec	shotdetect	Null ptr deref	film.cpp	312	
25	libavcodec	shotdetect	Null ptr Deref	film.cpp	346	Return value of avcodec_decode_video2() not checked (negative return on error). Failure in avpicture_fill() can later lead to null-ptr-dereference.
26	libjpeg	jp2a	Heap overflow	image.c	705	Return value of jpeg_read_scanlines() not checked. In certain conditions, this can lead to an integer overflow, which eventually leads to a heap overflow (in process_scanline_jpeg()).
27	libavformat	unpaper	Null ptr deref	file.c	228*	Return value of avformat_write_header() is not checked and context is passed onto av_write_frame(). This can lead to a null-ptr-dereference in compute_muxer_pkt_fields() inside libavformat itself, at libavformat/mux.c:580.
28	libavformat	unpaper	FPE	file.c	228*	Return value of avformat_write_header() is not checked and context is passed onto av_write_frame(). The context can be in a state that is not properly intitialized. This can lead to a FPE in frac_add() in libavformat itself, at libavformat/mux.c:84.
29	libavformat	unpaper	SEGV	file.c	43	Return value of avformat_find_stream_info() not checked. This can lead to a segmentation fault later.
30	libpoppler	apvlv	Heap overflow	ApvlvDoc.cc	2203	Return value of library API function is checked. However the return value of application's own function that wraps the call to the api is not checked. Under certain conditions, this can lead to heap buffer overflow.
31	libjpeg	jpegoptim	Null ptr deref	jpegoptim.c	711	Return value of jpeg_read_cofficients() not checked. This can lead to null-ptr-dereference later.

* Depending on the location and context at the time of fault, the same API misuse causes different faults in the library.