# An Empirical Study of OSS-Fuzz Bugs

Zhen Yu Ding* and Claire Le Goues†
*Motional (work done at Carnegie Mellon University), Pittsburgh, USA
zhd23@pitt.edu
†School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
clegoues@cs.cmu.edu

***Abstract*—Continuous fuzzing is an increasingly popular technique for automated quality and security assurance. Google maintains OSS-Fuzz: a continuous fuzzing service for open source software. We conduct the first empirical study of OSS-Fuzz, analyzing 23,907 bugs found in 316 projects. We examine the characteristics of fuzzer-found faults, the lifecycles of such faults, and the evolution of fuzzing campaigns over time. We find that OSS-Fuzz is often effective at quickly finding bugs, and developers are often quick to patch them. However, flaky bugs, timeouts, and out of memory errors are problematic, people rarely file CVEs for security vulnerabilities, and fuzzing campaigns often exhibit punctuated equilibria, where developers might be surprised by large spikes in bugs found. Our findings have implications on future fuzzing research and practice.**

***Index Terms*—fuzzing, continuous fuzzing, OSS-Fuzz**

## I. Introduction

Fuzz testing is effective at finding bugs and security vulnerabilities such as crashes, memory violations, and undefined behavior. Continuous fuzzing — using fuzz tests as part of a continuous testing strategy — is increasingly popular in both industry [1]–[5] and open-source software engineering [1], [3], [6], [7]. To improve the quality and security of open-source, Google maintains OSS-Fuzz [6], a continuous fuzzing service supporting over 300 open-source projects.

We present the first empirical study of OSS-Fuzz, examining over 4 years of data and 23,907 fuzzer-discovered bugs found in 316 software projects. To our best knowledge, this is the largest study of continuous fuzzing at the time of writing. We expand the body of empirical research on fuzzing, which the fuzzing community expressed a need for [8]. Our main contributions are:

- We present the first empirical study of OSS-Fuzz and the largest study of continuous fuzzing at the time of writing, analyzing 23,907 bugs in 316 projects.
- We analyze the characteristics of fuzzer-found bugs. We consider fault types, flakiness, fuzz blockers, unfixed bugs, CVE entries, and relationships among these features. We find that many fuzzer-found bugs harm availability without posing direct threats to confidentiality or integrity, timeouts and out of memory errors are unusually flaky, flaky bugs are mostly unfixed, and few bugs, mostly memory corruption bugs, receive CVE entries.
- We probe OSS-Fuzz bugs' lifecycles. We find that most fuzzer-found bugs are detected and fixed quickly, albeit lifecycles vary across fault types, flaky bugs are slower to detect and fix, and fuzz blockers are slower to fix.
- We study the longitudinal evolution of fuzzing campaigns. We find that bug discovery often show punctuated equilibria, with occasional spikes in the number of bugs found interspersed among relatively slow bug hunting.

**Algorithm 1** Coverage-guided fuzzing.

```
1: procedure FUZZ(program $p$, set of seed inputs $I_0$)
2:     $Inputs \leftarrow I_0$
3:     $TotalCoverage \leftarrow$ coverage of $p$ on $Inputs$
4:     while within time budget do
5:         $i \leftarrow$ pick from $Inputs$
6:         $i' \leftarrow$ mutate $i$
7:         $coverage, error \leftarrow$ execute $p$ on $i'$
8:         if $\exists$ $error$ then
9:             report $error$ and faulty input $i'$
10:            optionally, exit
11:        else if $coverage \not\subseteq TotalCoverage$ then
12:            add $i'$ to $Inputs$
13:            $TotalCoverage \leftarrow TotalCoverage \cup coverage$
```

Section II provides background to contextualize our work. Section III overviews the bug reports under study. We analyze fault characteristics, fault lifecycles, and longitudinal evolution in Sections IV, V, and VI respectively. Section VII discusses our findings' implications for research and practice, Section VIII overviews related work, and Section IX concludes.

## II. Background

To explain how OSS-Fuzz found the bugs under analysis, we provide background on coverage-guided fuzzing (Section II-A) and OSS-Fuzz (Section II-B). We also discuss the CIA triad of information security (Section II-C), which we use to analyze the security impact of bugs in our analysis.

### A. Coverage-guided fuzzing

Coverage-guided fuzzing (CGF), implemented by tools such as AFL [9], libFuzzer [10], and honggfuzz [11], is a popular bug detection method. CGF uses genetic search to find inputs that maximize code coverage. Algorithm 1 describes CGF at a high level. The algorithm maintains a pool of *Inputs* and the *TotalCoverage* of program $p$ on *Inputs*. The user provides seed inputs $I_0$ to instantiate *Inputs*. The fuzzer repeatedly picks an input $i$ from the pool of *Inputs* and applies a mutation (e.g., increment, bit flip, or user-defined mutations) to produce $i'$. The fuzzer then executes program $p$ on mutated input $i'$ to gather the *coverage* of program $p$ on $i'$ and detect any *error*, such as crashes, assertion violations, timeouts,

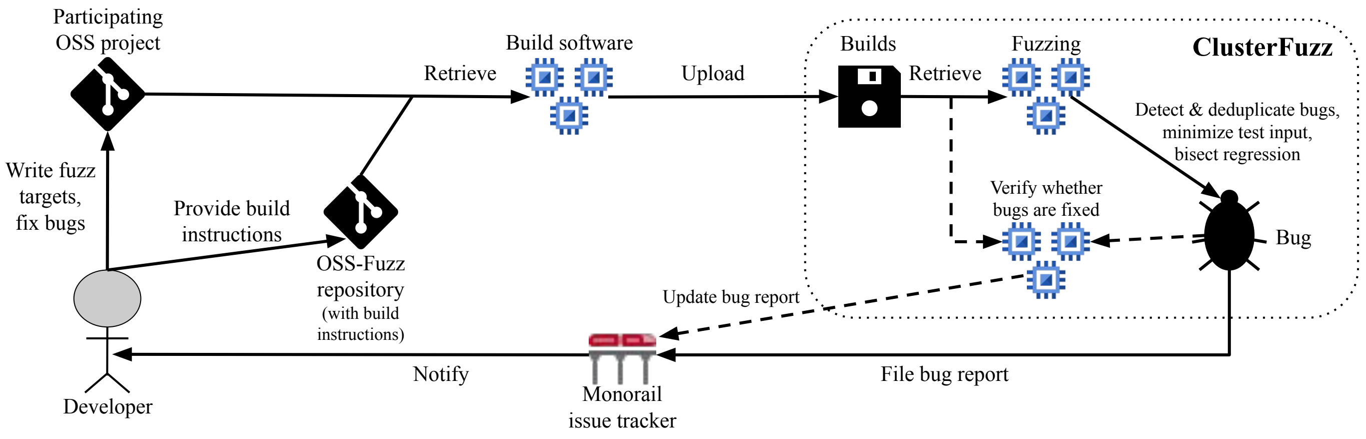

Participating OSS project
Retrieve
Build software
Upload
Builds
Retrieve
Fuzzing
ClusterFuzz
Detect & deduplicate bugs, minimize test input, bisect regression
Write fuzz targets, fix bugs
Provide build instructions
OSS-Fuzz repository (with build instructions)
Verify whether bugs are fixed
Bug
Update bug report
Notify
File bug report
Developer
Monorail issue tracker

Fig. 1: OSS-Fuzz's workflow

memory leaks or access violations (with ASan [12]), undefined behavior (with UBSan [13]), uninitialized memory use (with MSan [14]), or data races (with TSan [15]). If $i'$ does not trigger an *error* and discovers new *coverage* that is not previously seen in *TotalCoverage*, then add $i'$ to *Inputs* and update *TotalCoverage*. By finding inputs that cover new code, CGF aims to test as much of the program as possible.

Fuzzers need an entrypoint into the program to provide test inputs; such an entrypoint is often called a *fuzz target*. libFuzzer-style fuzz targets — which AFL and honggfuzz also support and OSS-Fuzz uses — are functions that take in fuzzer-generated arbitrary bytestream input, transform the input to program-usable input data if needed, and execute the program under test with the input.

## B. Continuous Fuzzing and OSS-Fuzz

Continuous fuzzing uses fuzzing as part of a continuous testing strategy to find regressions as software evolves. Several organizations incorporate fuzzing as part of their quality assurance strategy [1], [3], [16], [17] or offer tools that provide continuous fuzzing as a service [2], [4]–[6].

OSS-Fuzz [6] is Google's continuous fuzzing service for open source software (OSS) projects that are widely used or critical to global IT infrastructure. OSS-Fuzz uses ClusterFuzz [1], Google's continuous fuzzing framework. Figure 1 illustrates OSS-Fuzz's workflow. Developers in a participating OSS project write fuzz targets and provide instructions for building the software. OSS-Fuzz continuously builds the software and uploads it to ClusterFuzz. ClusterFuzz finds fuzz targets and uses the coverage-guided fuzzers AFL [9], libFuzzer [10], and honggfuzz [11] to fuzz the software. Upon detecting a bug, ClusterFuzz checks whether the bug is a duplicate of any previously found bugs, minimizes the bug-inducing input, and bisects the range of commits in which the regression occurred. If the bug is not a duplicate, then ClusterFuzz files a bug report on Monorail, an issue tracker. ClusterFuzz periodically verifies whether any previously found bugs are fixed; if so, OSS-Fuzz updates fixed bugs' report.

Bug reports are initially available only to project members. OSS-Fuzz uses Google's standard 90-day public disclosure policy [18], [19] for all found bugs. If a bug is patched, then the disclosure date moves up to either 30 days post-patch or stays at 90 days post-discovery, whichever is earlier. Bug disclosure deadlines are a standard practice in industry; deadlines encourage prompt repair, while the delay in public disclosure gives developers time to write and discreetly distribute patches.

## C. CIA Triad

Since fuzzing is often used as part of a security testing strategy, we analyze the potential security impacts of bugs using the CIA triad [20]. CIA stands for confidentiality, integrity, and availability; each is a desired security property.

Confidentiality entails that only authorized users can access a resource. Data breaches are instances of violated confidentiality. Dangerous memory reads, such as buffer overflow reads or use of uninitialized values, can leak confidential data residing in memory. For example, Heartbleed was a buffer overflow read vulnerability in OpenSSL that jeopardized the confidentiality of server data, such as private keys [21].

Integrity entails that only authorized users can modify resources in an allowable manner. Unauthorized deletion or tampering of resources are instances of violated integrity. Tampering of memory — which may result from improper memory management or dangerous functions that allow for buffer overflow writes or unsafe heap operations — can corrupt data or facilitate arbitrary code execution.

Availability entails that resources remain available to users. Denial of service attacks harm availability. Attackers can leverage resource exhaustion bugs such as timeouts, out of memory errors, or memory leaks to reduce system performance. Bugs that result in abnormal process termination such as null dereferences, stack overflows, or operating system signals can deny service to anyone else using the same terminated process.

## III. OSS-Fuzz Bug Reports

We extract data from OSS-Fuzz bug reports on Monorail. Figure 2 shows OSS-Fuzz Issue #20000 as an example. ClusterFuzz generates these reports in a standardized format. The report indicates which software "Project" is affected, the "Fuzzing Engine" and "Fuzz Target" that found the bug, the

| | |
|---|---|
| **Issue 20000** | Project: binutils |
| Reported by ClusterFuzz | Fuzzing Engine: libFuzzer |
| on Fri, Jan 10, 2020 | Fuzz Target: fuzz_disassemble |
| 7:15 AM EST | Platform Id: linux |
| Status: Verified (Closed) | Crash Type: Unsigned-integer-overflow |
| Modified: Feb 10, 2020 | Sanitizer: undefined (UBSAN) |
| | Regressed: oss-fuzz.com/revisions?job=*omitted* |
| Labels: | &range=201912170318:201912190318 |
| Reproducible | Reproducer Testcase: oss-fuzz.com/download |
| *7 others…* | ?testcase_id=*omitted* |

**Comment 1** by ClusterFuzz on Jan 11, 2020, 10:24 AM EST
ClusterFuzz testcase…is verified as fixed in *link to fix code commit range.*

**Comment 2** by sheriffbot on Feb 10, 2020, 1:10 PM EST
This bug has been fixed for 30 days. It has been opened to the public.

Fig. 2: An OSS-Fuzz bug report. Some details are omitted or edited for brevity. Original report at https://bit.ly/3oaLhCp

"Platform" used, and the "Crash Type" of the bug. To aid bug reproduction, the report indicates which "Sanitizer" was used, the range and time window of commits where the software "Regressed," and a "Reproducer Testcase" to trigger the bug. After ClusterFuzz verifies that the bug is fixed, it posts a comment indicating the commit range where the software was fixed. Sheriffbot tracks disclosure deadlines and posts comments to warn about approaching deadlines (if a bug is still unfixed) or notify that a bug passed a disclosure deadline.

We use Selenium [22], a browser automation tool, with Google Chrome to scrape OSS-Fuzz bug reports on Monorail. We ethically scrape data in accordance with Monorail's robots.txt file [23]. We extract data fields from the bug reports' text via pattern matching. We scrape 23,907 bug reports from 316 projects, with report dates spanning from May 2016 to October 2020[1].

## IV. Fault Characteristics

To gain a better sense of the landscape of OSS-Fuzz bugs, we begin by examining the following fault characteristics:

**Fault type** A categorization of faults; e.g., timeout, out of memory, null dereference.
**Flakiness** Whether a bug is reliably reproducible.
**Fuzz blocker** Whether a fuzzer encounters the same bug very often, which blocks further fuzzing of downstream code.
**Unfixedness** Whether a bug is unfixed.
**CVE** Whether a bug has an associated record in the Common Vulnerabilities and Exposures (CVE) system [24].

The remainder of the section motivates, describes, and analyzes these characteristics individually, and examines relationships between them.

### *A. Fault types*

Fuzzing is often discussed in the context of software security as an effective tool for uncovering security vulnerabilities [2], [6], [25]–[32], particularly in finding buffer and numeric overflows — two of the most common software

[1]due to OSS-Fuzz's disclosure policy, not all bug reports from July–October 2020 were publicly available at the time of data collection.

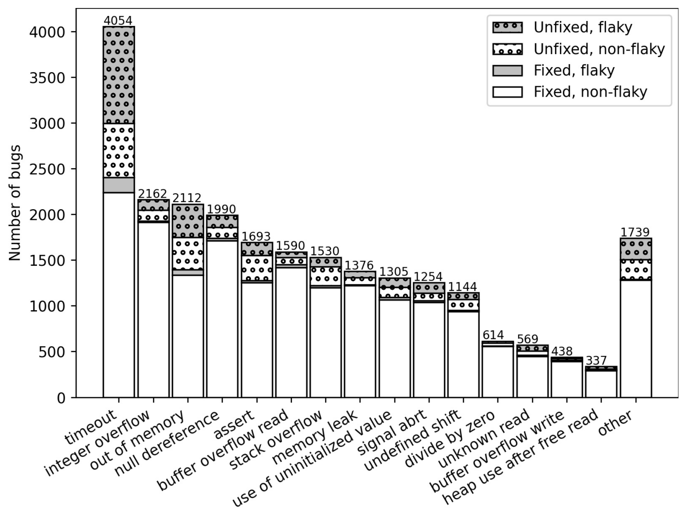


Fig. 3: Numbers of bugs among the top 15 fault types.

security vulnerabilities [33]–[36]. Prior advances in fuzzing research targeted specific fault types, such as timeouts [37]–[39], out of memory errors [40], [41], integer overflows [42], or buffer overflows [43]–[46]. The attention on fuzzing as a security testing technique and prior work on targeting specific fault types motivates the following question:

**RQ-FT** Which fault types do OSS-Fuzz's coverage-guided fuzzers frequently find?

*a) Methodology:* To determine fault type, we use the "Crash Type" field in bug reports (e.g., in Figure 2, the "Crash Type" is "Unsigned-integer-overflow"). We standardize some text (e.g., group together "timeout" and "hang", or "null dereference" and "null reference"), and we consolidate heap, stack, and global overflows and underflows into buffer overflow. We group together "null dereference read" and "null dereference write," since reading or writing to a null address usually have similar consequences. For the opposite reasons, we distinguish between "buffer overflow read" and "buffer overflow write," since overreads primarily threaten confidentiality, while overwrites also threaten integrity and can lead to arbitrary code execution.

*b) Results:* Figure 3 shows the number of bugs among the top 15 fault types. Six of the most common fault types comprising 52% (*12316/23907*) of bugs — timeout, out of memory, null dereference, stack overflow, memory leak, and signal abrt — primarily harm availability by crashing. While such crashes might facilitate other attacks that compromise confidentiality or integrity by, for example, exposing potential vulnerabilities in the error-handling process, such crash-inducing faults are likely less severe in their own capacity.

The majority of fuzzer-found bugs primarily harm availability.

Four fault types comprising 23% (*5613/23907*) of bugs — integer overflow, assertion violation, undefined shift, and divide by zero — indicate unintended program logic. Such bugs can be exploited, for example, if an integer overflow affects a buffer index and thus can facilitate a buffer overflow. However, such unintended logic can also result in no more than abnormal termination or bad output.

Three fault types comprising 14% (*3232/23907*) of bugs — buffer overflow read, use of uninitialized value, and heap use after free read — do primarily jeopardize memory confidentiality. Buffer overflow writes (*2%, 438/23907*) jeopardize memory integrity and can lead to arbitrary code execution.

## B. Flakiness

Fuzzers can sometimes find flaky bugs, where a test input cannot reliably reproduce a bug. ClusterFuzz — the continuous fuzzing engine behind OSS-Fuzz — usually ignores unreproducible bugs; however, if a flaky bug appears very frequently, then ClusterFuzz will file a bug report [47]. Such flakiness motivates the following research questions:

**RQ-FLK** How prevalent are flaky bugs?
**RQ-FLK-FT** Which fault types are disproportionately flaky?

*a) Methodology:* To identify flaky bugs, we look for bugs which ClusterFuzz deems insufficiently reproducible, or where there appears a comment in the bug report including phrases suggesting irreproducibility (e.g., "unreproducible," "can't reproduce"). This produces a conservative count of flaky bugs, since ClusterFuzz only reports a flaky bug if it appears very often, and developers experiencing reproducibility problems may stay silent, complain outside of the bug report's comment section on Monorail (e.g., they may use their project's own issue tracker), or use different phrasing to suggest flakiness.

*b) Results:* Out of 23907 studied bugs, we identify 3139 (*13%*) as flaky. Figure 3 breaks down the composition of flaky and non-flaky among the top 15 fault types. We observe high rates of flakiness in timeouts (*30%, 1225/4054, $p < 10^{-272}$ via a $\chi^2$ test on the null hypothesis that timeouts and non-timeouts are equally flaky*) and out of memory errors (*20%, 424/2112, $p < 10^{-22}$ via a $\chi^2$ test on the null hypothesis that OOMs and non-OOMs are equally flaky*). Since both timeouts and OOMs are resource exhaustion bugs, the unpredictability of resource availability and usage may hamper reproducibility.

Timeouts and OOMs are disproportionately flaky.

## C. Fuzz blockers

If a bug occurs very frequently, it can block subsequent fuzzing, hurting fuzzer performance. ClusterFuzz reports these frequently crashing bugs as *fuzz blockers*, and advises maintainers that fixing such bugs would lead to better fuzzing. The fuzzing community expressed concern that while fuzz blockers are important to fix from a fuzzing practitioner's viewpoint, developers may ascribe low priority to fuzz blockers [48]. We thus ask the following research questions:

**RQ-BLK** How prevalent are fuzz blockers?
**RQ-BLK-FC** What relationships exist between fuzz blocker prevalence and other fault characteristics?

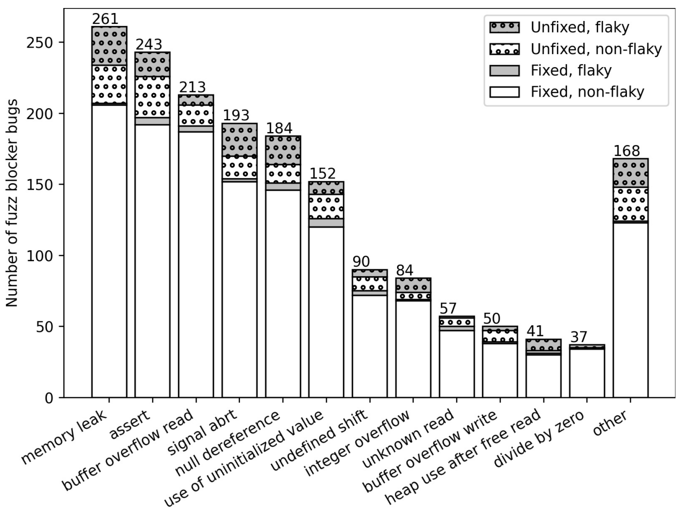


Fig. 4: Numbers of fuzz blockers among the top 12 fault types.

*a) Methodology:* If ClusterFuzz detects that a bug appears very frequently, then ClusterFuzz adds a "Fuzz-Blocker" label to the bug report. We use ClusterFuzz's labeling. However, based on ClusterFuzz's source code [49], we notice that ClusterFuzz does not report fuzz blocking timeouts, out of memory errors, or stack overflows. We speculate that ClusterFuzz's developers excluded these fault types since they block fuzzing so often that distinguishing blockers from non-blockers is uninteresting. Given the lack of data on the excluded fault types, we do not count them as fuzz blockers.

*b) Results:* Out of 16211 bugs that ClusterFuzz examined for fuzz blockage, 1773 (*11%*) are fuzz blockers, of which 10% (*185/1773*) are flaky. Figure 4 shows the 12 most prevalent fault types among ClusterFuzz-identified fuzz blockers. Memory leaks are the most common fault type among fuzz blockers (*15%, 261 of 1773 fuzz blockers*), and fuzz blocking memory leaks comprise 19% (*261/1376*) of all memory leaks. Leak detection is part of ASan [12], but can be disabled to ignore memory leaks and continue fuzzing code downstream of any leaks. Assertion violations are the second most common fault type among fuzz blockers (*14%, 243 of 1773 fuzz blockers*), and fuzz blocking assertion violations comprise 14% (*243/1693*) of all assertion violations. Recompiling code without the fuzz blocking assertion would similarly unblock fuzzing downstream of the assertion failure.

Memory leaks and assertion violations are the most common ClusterFuzz-identified fuzz blockers, both of which can be disabled.

Although the top two fault types are relatively low-impact, buffer overflow read, a high severity fault type, is the third most common fault type among ClusterFuzz-identified fuzz blockers (*12%, 213 of 1773 fuzz blockers*), and comprises 13% (*213/1590*) of all buffer overflow reads. Since fuzz blockers appear very often during fuzzing, an attacker may find a fuzz blocking buffer overflow quickly (we confirm the intuition that fuzz blockers are quickly found in Sec. V-A: Time-to-detect).

## D. Unfixed bugs

Developers may choose not to fix a bug for many reasons. A bug might be too hard to repair (e.g., if a bug is hard to replicate, or requires an expensive code overhaul). Maintainers might also judge that a reported bug is not actually a bug or is not important enough to warrant a fix (e.g., if developers think a fuzzer-found error case is low-impact, obscure, or will not manifest in practice). Fuzzers, in particular, can flood software projects with many seemingly low-impact bugs [48], [50]. Attackers, however, can use known bugs as a low-cost starting point to craft attacks with existing vulnerabilities, gather intelligence to eventually craft a more complex exploit, or gauge the agility of a software team's response to faults. We thus ask:

**RQ-NOFIX** How many fuzzing-identified bugs are unfixed?
**RQ-NOFIX-FC** Which bugs are often unfixed?

*a) Methodology and Results:* We examine publicly visible bugs without a verified fix at the time of data collection. Of the publicly visible bugs, 22% (*5148/23907*) are unfixed. OSS-Fuzz does not normally publicize unfixed bugs until after the 90-day disclosure deadline, except for 130 unfixed bugs that were publicized early by developers and whose data we collected prior to the default 90-day deadline. Bugs unfixed at the time of data collection were left unfixed for a median of 437 days, with 95% of bugs unfixed for 90–1275 days.

Figure 3 breaks down the composition of fixed and unfixed bugs according to flakiness and fault types. Flaky bugs, even if appearing frequently, are overwhelmingly not fixed (*86%, 2684/3139*). Non-flaky bugs are unfixed only 12% (*2464/20768*) of the time. We postulate that some flaky bugs, even though they appear frequently, are not actually faults in the software itself; for example, software might timeout due to a temporarily unavailable resource. Irreplicability also hinders attempts to diagnose and understand the bug. Although OSS-Fuzz encourages developers to write speculative patches for irreplicable bugs based on the information in the bug report, developers may hesitate to write speculative patches with limited information for a fault that may not actually exist.

Flaky bugs are overwhelmingly unfixed.

Because flaky bugs are often unfixed, we exclude them when comparing fix rates among different fault types to prevent flakiness from acting as a confounding variable. Three fault types have disproportionately many unfixed bugs compared to other fault types: timeout (*21%, 589 of 2829 non-flaky bugs are unfixed, $p < 10^{-55}$ with a $\chi^2$ test on the null hypothesis that timeouts and non-timeouts have the same frequency of unfixed bugs*), out of memory (*21%, 353/1688, $p < 10^{-32}$*), and assertion violation (*18%, 274/1530, $p < 10^{-13}$*).

However, as discussed in Section IV-A, timeouts and out of memory errors primarily hamper availability rather than confidentiality or integrity, and assertion violations do not necessarily pose an immediate threat to security or reliability. Considering the generally lower security impact of such bugs, the lower fix rate suggests greater developer apathy towards such bugs. Timeouts and out of memory errors may also be more annoying for developers to reproduce and fix, as reproducing such bugs consume substantial time and/or hardware resources, which slows the potentially repetitive process of analyzing the bug and testing patches. The lower impact and greater pain in fixing such bugs may explain the lower fix rate.

Timeouts, out of memory errors, and assertion violations are more frequently unfixed compared to other fault types, even if not flaky.

## E. CVEs

The Common Vulnerabilities and Exposures (CVE) list [24] is a public reference for security vulnerabilities. Organizations designated as CVE Numbering Authorities (e.g., MITRE, Debian, Microsoft, PHP Group) can issue CVEs for vulnerabilities discovered in-house or reported from third parties. Various security tools, such as threat intelligence dashboards and security scanning tools, use CVEs as a source of threat information. The security community expressed concern that very few of OSS-Fuzz's security vulnerabilities were issued CVEs [51], and thus most of OSS-Fuzz's discovered vulnerabilities are not visible to security tools that rely on CVEs. This prompts the following research questions:

**RQ-CVE** How many OSS-Fuzz bugs have CVE records?
**RQ-CVE-FT** Which fault types often receive CVEs?

*a) Methodology and Results:* A CVE record usually references bug reports or other documentation on a vulnerability. We mine CVE records [52] to find URL references to OSS-Fuzz bug reports. We produce a conservative list of OSS-Fuzz bugs with CVEs, since a CVE's list of references may be incomplete and omit a reference to an OSS-Fuzz bug report.

We find 98 OSS-Fuzz issues with CVEs, a small number relative to the over 20,000 bugs found by OSS-Fuzz. Table I presents the fault types of bugs with CVEs. Most bugs with CVEs are dangerous memory operations that threaten confidentiality or integrity, although people have also filed CVEs for bugs that primarily affect availability, such as timeouts, out of memory errors, and null dereferences.

Few OSS-Fuzz bugs result in CVEs, and most filed CVEs are for memory corruption bugs.

*b) Limitations:* Outsiders might independently co-discover and file CVEs for vulnerabilities found by OSS-Fuzz. Thus, the actual number of CVEs filed by project members in response to an OSS-Fuzz discovery is likely even lower.

[2]Buffer overflow write (40), heap use after free write (2), unknown write (2), stack use after return write (1), container overflow write (1).

[3]Buffer overflow read (16), unknown read (8), heap use after free read (6), use of uninitialized value (4).

[4]Divide by zero (1), signal abrt (1), negative size param (1), floating point exception (1), assert (1), memory leak (1).

| Fault type | OSS-Fuzz bugs with filed CVEs |
|---|---|
| dangerous memory write[2] | 46 |
| dangerous memory read[3] | 34 |
| null dereference | 4 |
| out of memory | 4 |
| heap double free | 2 |
| timeout | 2 |
| other[4] | 6 |

TABLE I: Number of CVEs filed per fault type. Most CVEs are filed for memory corruption bugs.

## V. Fault Lifecycle

Prior work [33] found that the median lifespan — the time between fault introduction and repair — of vulnerabilities was 438 days. Numeric errors and buffer overflows specifically had median lifespans of 659.5 and 781 days respectively. In that time, an attacker may find and exploit such vulnerabilities. Continuous fuzzing aims to shorten bug lifespans and help developers stay ahead of attackers. We study the following aspects of OSS-Fuzz bugs' fault lifecycles:

**Time-to-detect** The time from fault introduction to detection.
**Time-to-fix** The time from fault detection to repair.

The remainder of the section examines these aspects and their relationships to fault characteristics discussed in Sec. IV.

### A. *Time-to-detect*

By continuously probing software for faults, continuous fuzzing aims to shorten the time to detect regressions. Heartbleed, a buffer overflow read vulnerability in OpenSSL, was unfixed for two years [21], but is now a demonstrative bug for fuzzing [53]. Continuous testing techniques, such as continuous fuzzing, can shorten the lifespan of bugs and security vulnerabilities through rapid fault detection. Such lengthy bug lifespans, and the prospect of shortening such lifespans via continuous fuzzing, motivate the following research question:

**RQ-T2D** How much time elapses between fault introduction and detection (the *time-to-detect*)?

We also probe the relationship between fault characteristics and time-to-detect to reveal, for example, whether bugs of certain fault types take longer to detect.

**RQ-T2D-FC** What relationships exist between fault characteristics and time-to-detect?

*a) Methodology:* ClusterFuzz, after finding a bug, identifies the range of commits where the regression was introduced. We use the right time boundary of the regression range as the time of fault introduction. In the example presented in Figure 2, the time of fault introduction (2019-12-19-03:18 UTC) is the second datetime (201912190318) of the "range" value of URL in the "Regressed" field. We use the time of bug reporting as the time of bug detection; in Figure 2, the time of bug reporting is "Fri, Jan 10, 2020 7:15 AM EST." The time-to-detect is the time elapsed between these two times.

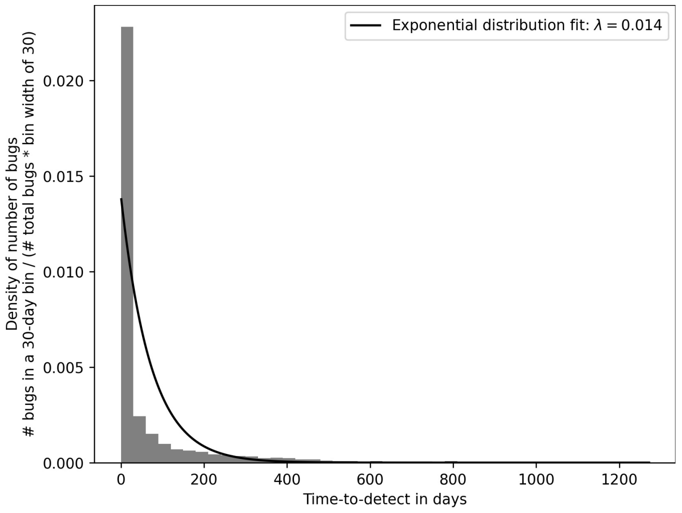
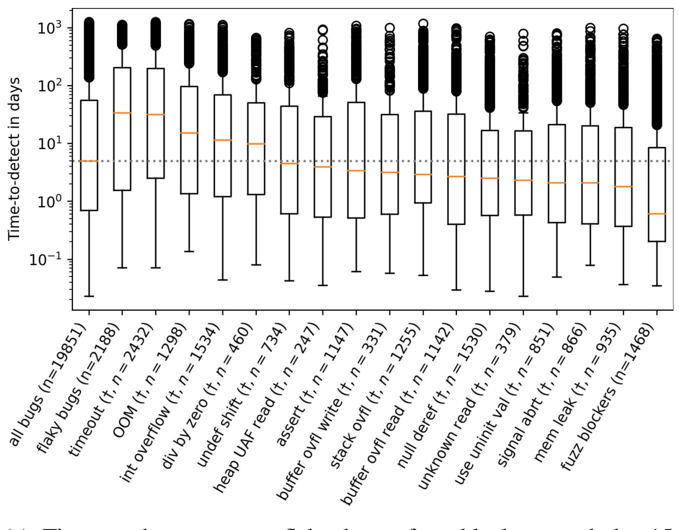


Fig. 5: Density of bugs with respect to time-to-detect. Bugs are often detected quickly, with the time intervals following an exponential distribution.

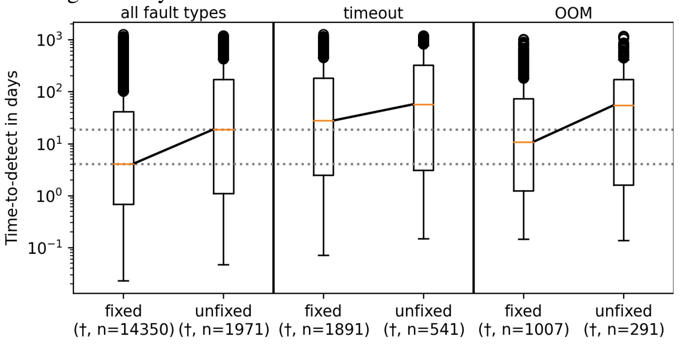
*b) Results:* Figure 5 shows the density of bugs over the time-to-detect axis. The density of bugs in a histogram bin is the number of bugs in the bin normalized by the width of the bin and the total number of bugs shown in the histogram. Such normalization makes the areas of the bars sum to one, akin to how the area under a probability density function sum to one (100% probability). The time-to-detect a bug is exponentially distributed ($p < 10^{-308}$ *via a Kolmogorov-Smirnov goodness of fit test*). OSS-Fuzz's fuzzers find most regressions quickly, with a median time-to-detect of 5 days. Our findings complement, with in-the-wild data, prior findings on the exponential cost of fuzzing [54]. Since coverage-guided fuzzing is a guided search through a combinatorially large search space of inputs, and fuzzer coverage plateaus over time [31], the exponential distribution is expected. The short time-to-detect provides further evidence that coverage-guided fuzzing is well-suited for continuous testing.

OSS-Fuzz detects the majority of identified regressions within a week.

Figure 6a compares the time-to-detect among various characteristics. Flaky bugs take much longer to detect than non-flaky bugs, with medians of 34 vs. 4 days to detect flaky vs. non-flaky bugs ($p < 10^{-109}$ *via a two-sided Mann-Whitney U test on the null hypothesis that flaky and non-flaky bugs have the same time-to-detect*). On the other hand, ClusterFuzz-identified fuzz blockers are found much earlier than non-blockers (*medians of 0.6 vs. 3.3 days,* $p < 10^{-82}$*, excluding the fault types that ClusterFuzz do not report fuzz blockers on*). Since fuzz blockers appear very frequently by definition, such bugs would appear more often in the search space, which increases the probability of discovering a fuzz blocker early.



(a) Times-to-detect among flaky bugs, fuzz blockers, and the 15 most common fault types. The dotted line is the median time over all analyzed bugs. Flaky bugs and multiple fault types take significantly longer to detect. Fuzz blockers and multiple fault types take significantly shorter to detect.



(b) Comparison of times-to-detect of bugs that developers (do not) fix. The lower (resp. upper) dotted line is the median time-to-detect over all fixed (unfixed) non-flaky/blocker bugs. Unfixed bugs generally have longer times-to-detect.

Fig. 6: Times-to-detect among bugs with various characteristics. A † means flaky bugs and fuzz blockers are excluded.

OSS-Fuzz finds flaky bugs late, and ClusterFuzz-identified fuzz blockers early.

The respectively long and short times-to-detect of flaky bugs and fuzz blockers prompt us to exclude these bugs when comparing fault types to prevent flakiness or fuzz blockers from acting as confounding variables. For example, flaky bugs, if not excluded, would skew timeout time-to-detect upwards due to the high prevalence of flaky timeouts. The median time-to-detect non-flaky, non-blocker bugs is 4.8 days.

Timeouts (*median time-to-detect of 32 days, $p < 10^{-158}$ via two-sided U test on the null hypothesis that timeouts and non-timeouts have the same time-to-detect*) and out of memory errors (*15 days, $p < 10^{-25}$*) take significantly longer than other bug types to detect. One possibility is that timeouts and OOMs can result from runaway looping or recursion. Fuzzers' coverage metrics often do not account for loop iterations or recursion depth, which reduces the benefit of coverage in guiding the input search towards inputs that consume a lot of resources. Specialized fuzzers such as SlowFuzz [37], which optimizes for path length, PerfFuzz [38], which optimizes for basic block execution counts, or `mem` (in FuzzFactory [40]), which optimizes for memory allocations, can guide the search for such bugs more effectively and reduce the time-to-detect.

Timeouts and out of memory errors take longer to detect.

Integer overflows (*12 days, $p < 10^{-13}$*) and divide by zeroes (*10 days, $p = 0.0007$*) also take longer to detect. Meanwhile, memory leaks (*1.8 days, $p < 10^{-20}$*), signal abrt (*2.1 days, $p < 10^{-14}$*), use of uninitialized values (*2.1 days, $p < 10^{-13}$*), unknown reads (*2.3 days, $p < 10^{-5}$*), and null dereferences (*2.5 days, $p < 10^{-22}$*) are faster to detect. Some of the variance between fault types may be the result of ClusterFuzz's prioritization of ASan, which detects memory access violations and leaks, and MSan, which detect use of uninitialized values, over UBSan, which detects undefined behavior such as integer overflow or divide by zero [55].

The three longest to detect fault types — timeout, OOM, and integer overflow — are by no means sparse; they are the three most prevalent fault types (Section IV-A). Despite the extra time taken to find such bugs, they are plentifully discoverable.

Slower-to-detect fault types are not sparse.

Figure 6b compares times-to-detect among fixed and unfixed bugs. We again exclude flaky bugs or fuzz blockers to avoid confoundment, especially since flaky bugs are largely unfixed. Unfixed bugs often take longer for fuzzers to detect (*$p < 10^{-55}$ via a two-sided U test on the null hypothesis that fixed and unfixed bugs have equal times-to-detect*). Perhaps the longer times-to-detect of unfixed bugs is indicative of fault complexity that makes the bugs both harder for fuzzers to find (hence the long time-to-detect) and harder for people to fix (hence the bug is unfixed). Perhaps developers are also more likely to forget about the details of an older code change and neglect to fix an older bug.

Unfixed bugs take longer to detect.

Since timeouts and out of memory errors are disproportionately unfixed, and unfixed bugs take longer to detect, we examine (un)fixed bugs among these two fault types to gauge the influence of unfixed bugs on the long times-to-detect of these two fault types. While unfixed bugs do take longer to detect among bugs of both fault types (*$p < 10^{-5}$ via a two-sided U test for timeouts, $p < 10^{-6}$ for OOMs*), fixed timeouts take longer to detect than fixed non-timeouts ($p < 10^{-125}$), and unfixed timeouts take longer to detect than unfixed non-timeouts ($p < 10^{-16}$). The same pattern appears in OOMs (*$p < 10^{-17}$ comparing fixed OOMs vs. non-OOMs, $p = 0.003$ comparing unfixed OOMs vs. non-OOMs*). Thus, timeouts and

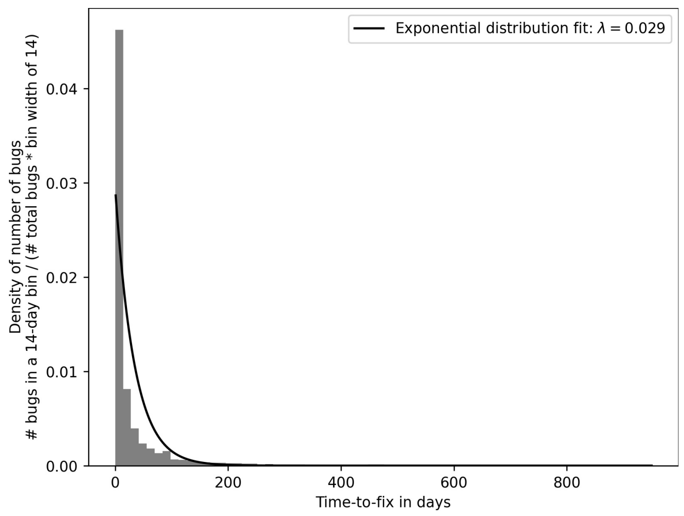


Fig. 7: Density of bugs with respect to time-to-fix. Bugs are often fixed quickly, with the time intervals following an exponential distribution.

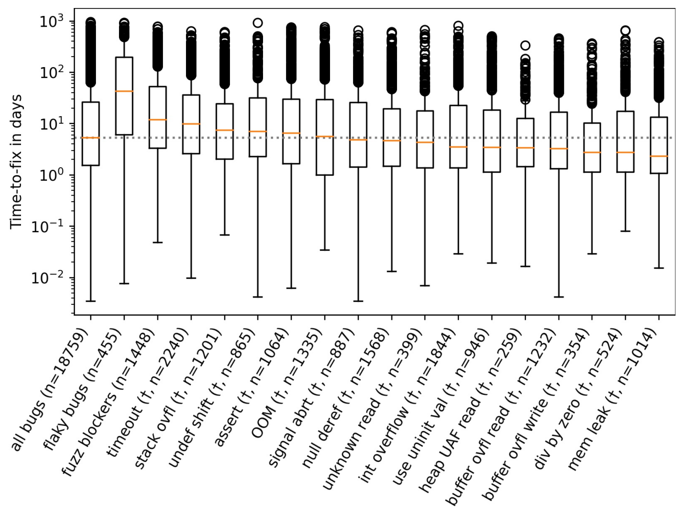


Fig. 8: Times-to-fix. A † means flaky bugs and fuzz blockers were excluded. The dotted line is the median time over all analyzed bugs.

OOMs' long times-to-detect are likely also attributable to factors other than those driving an increase in time-to-detect among unfixed bugs.

### B. *Time-to-fix*

Continuous fuzzing is most effective if developers promptly fix bugs. Otherwise, an accumulation of bugs can leave openings for attackers and hamper discovery of more bugs downstream of the unfixed faults. To promote prompt repair, OSS-Fuzz applies a 90-day disclosure policy. The severity, ease of repair, and developer attitudes toward bugs may affect the time to fix bugs. Prior work found that developers fix severe bugs almost twice as fast [56]. Moreover, fuzzing campaigns can generate an overload of low-priority bug reports [48], [50], suggesting a need to direct fuzzing efforts towards high-value bugs that developers eagerly fix. We thus ask:

**RQ-T2F** How much time elapses between fault detection and repair (the *time-to-fix*)?

**RQ-T2F-FC** What relationships exist between fault characteristics and time-to-fix?

*a) Methodology and Results:* We compute time-to-fix as the time from bug reporting to patch verification (Comment 1 in Figure 2). Figure 7 shows the density of bugs over the time-to-fix axis. The time-to-fix is exponentially distributed ($p < 10^{-308}$ *via a Kolmogorov-Smirnov goodness of fit test*). Out of all fixed bugs, 90% (*16952/18759*) are fixed within the 90-day disclosure deadline; the median time-to-fix is 5.3 days.

Most bugs are repaired well within the 90-day disclosure period, and over half are fixed within a week.

Figure 8 compares the time-to-fix among various bug categories. Flaky bugs, the vast majority of which are already unfixed, take an order of magnitude longer to repair, with a median time-to-fix of 43 days, versus 5.1 days for non-flaky bugs ($p < 10^{-54}$ *by a two-sided U test on the null hypothesis that flaky and non-flaky bugs have the same time-to-fix*). Given the difficulty of attempting to patch a hard-to-reproduce bug, the very long time-to-repair is unsurprising and supports prior findings [57] on bugs with reproducibility issues.

Flaky bugs, if fixed at all, are fixed very slowly.

Since fuzz blockers impede fuzzing performance by blocking program exploration downstream from the bug, quickly remedying blockers is important for a healthy fuzzing campaign. We find, however, that developers fix ClusterFuzz-identified fuzz blockers more slowly, with a median of 12 days as opposed to 4 for non-blockers ($p < 10^{-63}$ *via a two-sided U test, excluding the fault types that ClusterFuzz do not report fuzz blockers on*). Our finding confirms prior concerns [48] on developers' low prioritization of fuzz blockers, suggesting a need for greater awareness on the need to address blockers.

Fuzz blockers are fixed less, rather than more, urgently.

The long times-to-fix of flaky bugs and fuzz blockers prompt us to once again exclude these bugs when comparing fault types to prevent flakiness or fuzz blockers from acting as confounding variables. Timeouts (*median time-to-fix of 10 days,* $p < 10^{-45}$ *via two-sided U test on the null hypothesis that timeouts and non-timeouts have the same time-to-fix*), stack overflows (*7.5 days,* $p < 10^{-5}$), and undefined shifts (*7 days,* $p < 10^{-10}$) have longer times-to-fix, while memory leaks (*2.3 days,* $p < 10^{-18}$) and divide by zeroes (*2.8 days,* $p < 10^{-4}$) have shorter times-to-fix.

Although these aforementioned fault types can indicate faulty logic or impact availability via abnormal termination, they do not entail obvious consequences for confidentiality or integrity. Yet timeouts and memory leaks have the longest and shortest times-to-fix respectively among the top 15 fault types. Severity does not always correlate with the urgency of a patch.

Fault type severity does not always correlate with fix speed.

The short times-to-fix of memory leaks and divide by zeroes may suggest that such bugs are easier to fix, and thus fixed more quickly. Meanwhile, timeouts and stack overflows, even if not flaky, can be more annoying to repeatedly reproduce in the debugging process, and may be more difficult to localize and repair, resulting in longer times-to-fix.

Buffer overflow writes (*2.8 days,* $p < 10^{-6}$) also have shorter times-to-fix. Since such bugs are so severe — an attacker might execute arbitrary and malicious code — we are encouraged to see urgency in responding to such bugs.

Buffer overflow writes, which are very severe, are fixed more urgently.

*b) Limitations:* Some bugs have very short times-to-repair; 40 were fixed within one hour after the bug was reported. While some of these bugs may be repaired very quickly due to a rapid response to the bug report, we suspect that some bugs were discovered and patched before OSS-Fuzz had produced a bug report. We are not aware of an effective method to distinguish between the two cases, but the small number of such bugs poses only a marginal threat to validity.

## VI. Longitudinal Evolution

A motivation behind continuous fuzzing, as opposed to one-off fuzzing campaigns, is the benefit of continuously finding new bugs as software evolves. The continuous nature of OSS-Fuzz motivates the following research question:

**RQ-L:** How does the bug discovery rate change over time?

*a) Results:* Figure 9 show cumulative numbers of bugs found in projects with >200 discovered bugs. Many projects' cumulative distributions exhibit *punctuated equilibria*, with periods of slow growth punctuated by bursts of rapid growth.

Punctuated equilibria [58] appear in genetic algorithms [59] — which AFL, libFuzzer, and honggfuzz use as part of their search strategy — and in software evolution [60]–[62]. Fuzzers also exhibit punctuated equilibria in the long-term growth in coverage [31], [63]–[75] and number of bugs found [64], [67], [69], [70], [75]–[78]. We present evidence of punctuated equilibria in multi-year continuous fuzzing campaigns.

ClusterFuzz prioritizes hardware resources towards fuzz targets that are actively discovering new coverage [55]. Such a selection strategy increases selection pressure, which steepens peaks and flattens plateaus in the number of bugs found.

Bug discovery over time often exhibit punctuated equilibria, with short bursts of rapid bug discovery, rather than a consistent trickle of bugs.

## VII. Discussion and Implications

We are encouraged to see that OSS-Fuzz quickly finds regressions and developers quickly fix them. Our results provide real-world evidence of continuous fuzzing's effectiveness.

Flaky bugs, however, are problematic for developers, even if the bug appears often. The most flaky fault types, timeout and out of memory, may owe their flakiness from unpredictable resource availability or usage. Flakiness is also a symptom of a failure to control non-deterministic behavior during fuzzing, and flaky bugs may point developers to non-deterministic code that might require remediation for effective testing.

Timeouts and out of memory errors, respectively the first and third most common fault types, stand out as problematic. Even among non-flaky bugs, timeouts and OOMs are slower to detect, more often unfixed, and slower to fix. Specialized fuzzers [37], [38], [40] can find more such bugs in less time (countering the slow time-to-detect); however, existing widely-used fuzzers are already finding many such bugs. The lower severity of resource exhaustion bugs, combined with the potential human annoyance of reproducing such bugs, likely contributes to developers ascribing lower priority to such bugs.

Memory leaks and assertion violations, two other low-severity fault types, are common in fuzz blockers, which block fuzzing downstream from the bug. To mitigate the blockage, a continuous fuzzing system can temporarily disable leak detection or assertions to fuzz past the blockers. Developers fix fuzz blockers more slowly, and these automatic mitigations offer stopgap solutions until someone fixes the blockers.

On the other end of the severity scale, despite OSS-Fuzz finding thousands of severe bugs, such as memory corruption vulnerabilities, very few bugs received CVEs. This is a weakness in open-source security, as tools that rely on CVEs as a source of threat intelligence are not aware of potential threats in open-source software, which can percolate to computer systems and other software that depend on compromised open-source software. Developers without a security background might not be aware of the impact of CVEs, or might not wish to navigate through the process of requesting CVEs. More awareness and guidance may help to alleviate the issue.

Since fuzzing campaigns often exhibit punctuated equilibria with bursts of rapid bug discovery, developers may get an unpleasant surprise due to an avalanche of bugs in a short timeframe. A rapid dump of bugs can overwhelm developers and elicit defensive behavior [48]. Alerting developers of this phenomenon would mentally prepare them, avoiding surprises.

Fuzzing researchers expressed need for a fair time budget when evaluating fuzzers [8]. We suggest five days, OSS-Fuzz's median time-to-detect, as an option. Short time budgets (e.g., one hour) can introduce bias towards certain fault types.

*a) Limitations:* OSS-Fuzz is a continuous fuzzing service for open source software. Our findings might not extend to one-off fuzzing, commercial software, continuous fuzzing frameworks other than ClusterFuzz, or fuzzers other than the coverage-guided fuzzers AFL, libFuzzer, and honggfuzz.

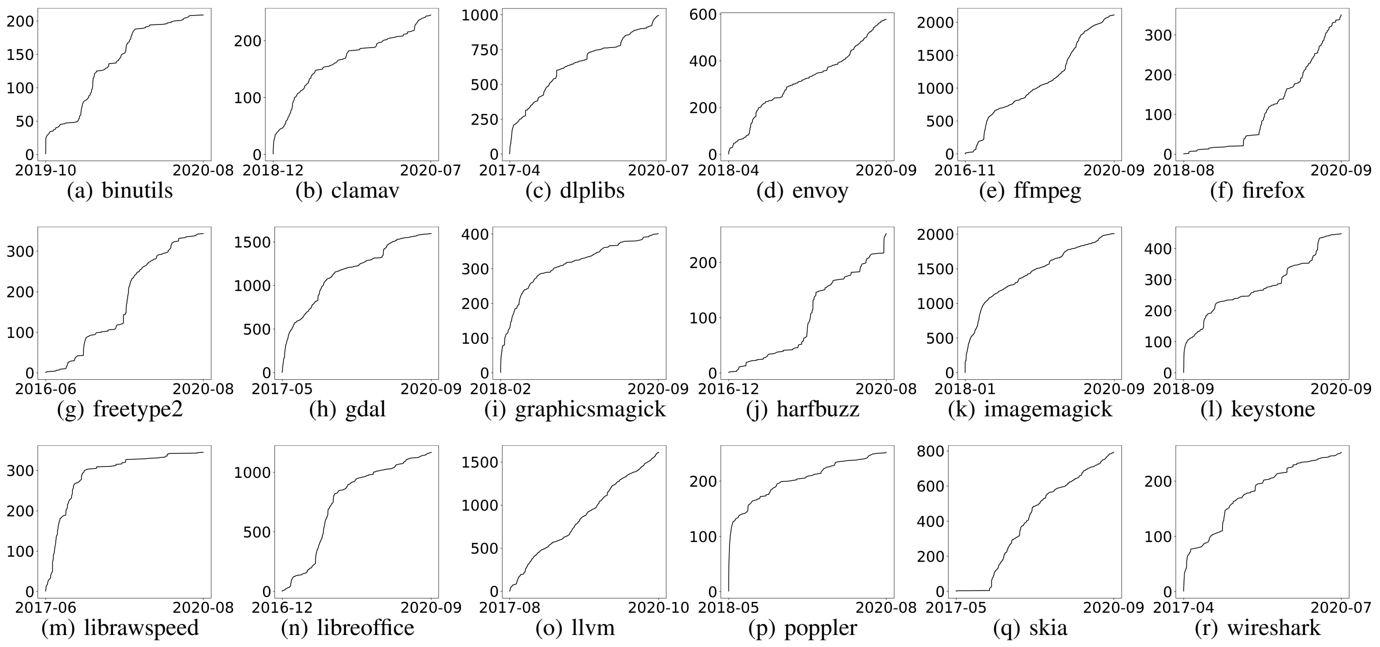


Fig. 9: Cumulative numbers of bugs found in projects with >200 bugs. The numbers of bugs found over time often exhibit punctuated equilibria, a common phenomenon in genetic algorithms such as those used in coverage-guided fuzzers.

## VIII. Related Work

Our work joins the corpus of empirical studies of software bugs, which have also examined fault types [33]–[35], [79]–[83], flakiness or irreproducibility [57], [81], [84]–[86], unfixed bugs [81], [87], [88], CVEs [33]–[35], [89], [90], and fault lifecycles [56], [81], [82], [85], [90]–[92]. In particular, Miller et al. [83], an empirical study of the reliability of Unix utilities, coined the term "fuzz" to denote a tool for randomly generating test inputs; they examined fault types of fuzzer-found bugs, similar to our empirical analysis of OSS-Fuzz.

MITRE's 2020 CWE Top 25 [93] lists the most impactful common software weaknesses observed in 2018–2020. Out-of-bounds write (CWE-787), out-of-bounds read (CWE-125), improper restrictions within memory buffer bounds (CWE-119), use after free (CWE-416), integer overflow (CWE-190), null dereference (CWE-476), and uncontrolled resource consumption (CWE-400) rank among the top 25. OSS-Fuzz found all of the above.

The 2019 Shonan Meeting on Fuzzing and Symbolic Execution [8] identified a need for more empirical analysis on fuzzing. They expressed interest in difficult or "deep" bugs, fair time budgets for evaluating fuzzers, and human-fuzzer interaction. Our empirical work sheds light on the current state of fuzzing in practice, illuminating the baseline to improve on.

We grow the literature on how practitioners interact with fuzzing. An industry report [94] found that writing fuzz targets required training, blockers are obstacles, and dirty hacks (e.g., disabling error reporting) can hide bugs from fuzzers.

Prior work examined continuous fuzzing for OS kernels. syzbot [7] is a continuous fuzzing system for kernels. A study [76] examined 2269 syzbot-found bugs in Linux, FreeBSD, NetBSD, and OpenBSD . The study analyzed the time to fix bugs (median of 38 days for Linux, <20 days for BSD) and fault types (debug checks, assertions, and use after free were among the most common). A report [95] on implementing continuous fuzzing for enterprise Linux kernels identified fuzz blockers as obstacles and found 132 bugs, 41 of which were reproducible. Lockups, deadlocks, and warnings were common fault types. We augment the existing literature on continuous fuzzing by analyzing a larger dataset of bugs in a diverse set of open source software.

## IX. Conclusion

We conduct the first empirical analysis of OSS-Fuzz bugs, evaluating 23,907 fuzzer-found bugs spanning over 4 years and 316 software projects. We fill a need for empirical evaluations of fuzzing, shine light on the state of the practice, and unveil insights for research and practice. While we examine bug reports, OSS-Fuzz and its participant projects contain other artifacts, such as fuzz targets, configurations, and code commits. We suggest an examination of these artifacts as part of a deeper study of continuous fuzzing. We provide an open-science package: https://doi.org/10.5281/zenodo.4625207

## Acknowledgements

We thank the ClusterFuzz and OSS-Fuzz teams, plus their OSS collaborators, for improving the quality of open source. We thank Rohan Padhye for his valuable feedback and insights. This research was partially funded by AFRL (#OSR-4066) and the NSF (#CCF-1750116). The authors are grateful for their support. Any opinions or findings expressed are those of the authors and do not necessarily reflect those of the US Government.

## REFERENCES

[1] "ClusterFuzz," google.github.io/clusterfuzz, accessed January, 2021.

[2] "Mayhem," www.forallsecure.com/mayhem, accessed January, 2021.

[3] J. Campbell and M. Walker, "Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale," https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/, 2020.

[4] "GitLab completes integration of fuzzing solutions to bolster DevSecOps capabilities," https://about.gitlab.com/press/releases/2020-11-19-gitlab-completes-integration-of-fuzzing-solutions.html, 2020.

[5] "Code Intelligence," https://www.code-intelligence.com/, accessed December, 2020.

[6] K. Serebryany, "OSS-Fuzz — Google's continuous fuzzing service for open source software," in *USENIX Security Symposium*, 2017.

[7] "syzbot," https://github.com/google/syzkaller/blob/fc7735a27949755327024847e12dcc1b868bcb99/docs/syzbot.md, 2020.

[8] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software (to appear)*, 2020.

[9] "american fuzzy lop," https://github.com/google/AFL, accessed January, 2021.

[10] "libFuzzer — a library for coverage-guided fuzz testing," https://www.llvm.org/docs/LibFuzzer.html, accessed January, 2021.

[11] "honggfuzz," https://honggfuzz.dev, accessed January, 2021.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, ser. USENIX ATC '12, 2012.

[13] "UndefinedBehaviorSanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, The Clang Team, accessed December, 2020.

[14] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *International Symposium on Code Generation and Optimization*, ser. CGO '15, 2015.

[15] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Workshop on Binary Instrumentation and Applications*, ser. WBIA '09, 2009.

[16] P. Shah, "Continuous fuzing with go at Dgraph," https://dgraph.io/blog/post/continuous-fuzzing-with-go/, 2020.

[17] T. Simonite, "This bot hunts software bugs for the Pentagon," *WIRED*, 2020.

[18] "Bug disclosure guidelines," https://google.github.io/oss-fuzz/getting-started/bug-disclosure-guidelines/#bug-disclosure-guidelines, accessed January, 2021.

[19] C. Evans, B. Hawkes, H. Adkins, M. Moore, M. Zalewski, and G. Eschelbeck, "Feedback and data-driven updates to Google's disclosure policy," https://googleprojectzero.blogspot.com/2015/02/feedback-and-data-driven-updates-to.html, 2015.

[20] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[21] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 63–67, 2014.

[22] "SeleniumHQ browser automation," https://www.selenium.dev/, accessed January, 2021.

[23] "robots.txt," https://bugs.chromium.org/robots.txt, accessed January, 2021.

[24] "Common vulnerabilities and exposures," https://cve.mitre.org/, MITRE, accessed January, 2021.

[25] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[26] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[27] K. Serebryany, D. Drysdale, C. Lopez, and M. Moroz, "Why fuzz?" https://github.com/google/fuzzing/blob/c4458cc2afa4c23f42190611f2172e0211171bbf/docs/why-fuzz.md, 2020.

[28] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[29] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *International Conference on Multimedia Information Networking and Security*, ser. ICMINS '12. IEEE, 2012.

[30] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[31] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Fuzzing: Breaking things with random inputs," in *The Fuzzing Book*. Saarland University, 2020.

[32] M. Sutton and A. Greene, "The art of file format fuzzing," in *Blackhat USA*, 2005.

[33] F. Li and V. Paxson, "A large scale empirical study of security patches," in *Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[34] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *International Conference on Software Engineering*, ser. ICSE '20, 2020.

[35] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *International Conference on Software Engineering*, ser. ICSE '12, 2012.

[36] S. Christey and R. A. Martin, "Vulnerability type distributions in CVE," https://cwe.mitre.org/documents/vuln-trends/vuln-trends.pdf, MITRE, Tech. Rep., 2007.

[37] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[38] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *International Symposium on Software Testing and Analysis*, ser. ISSTA '18, 2018.

[39] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," in *Network and Distributed Systems Security*, ser. NDSS '20, 2020.

[40] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[41] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *International Conference on Software Engineering*, ser. ICSE '20, 2020.

[42] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015.

[43] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser: A guided fuzzer to find buffer overflow vulnerabilities," in *USENIX Security Symposium*, 2013.

[44] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, "Smart fuzzing method for detecting stack-based buffer overflow in binary codes," *IET Software*, vol. 10, no. 4, pp. 96–107, 2016.

[45] A. Fioraldi, D. Daniele Cono, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *Secure Development Conference*, ser. SecDev '20, 2020.

[46] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *USENIX Security Symposium*, 2017.

[47] "Fixing a bug," https://google.github.io/clusterfuzz/using-clusterfuzz/workflows/fixing-a-bug/, accessed January, 2021.

[48] C. Holler, "The human component in bug finding," in *FuzzCon Europe*, 2020.

[49] https://github.com/google/clusterfuzz/blob/0b5c023eca9e3aac41faba17da8f341c0ca2ddc7/src/appengine/handlers/cron/cleanup.py, 2020.

[50] "Z3Prover/z3 issue no. 4461," https://github.com/Z3Prover/z3/issues/4461, accessed December, 2020.

[51] A. Gaynor, "Thousands of vulnerabilities, almost no CVEs: OSS-Fuzz," https://seclists.org/oss-sec/2019/q2/165, 2019.

[52] https://cve.mitre.org/data/downloads/allitems.csv, accessed October, 2020.

[53] M. Morehouse and M. Kaplan, "libFuzzer tutorial," https://github.com/google/fuzzing/blob/aeaafab3dd557ed050a0c1c659b7caa6899e89dc/tutorial/libFuzzerTutorial.md, 2020.

[54] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Joint Meeting of the European Software Engineering*

*Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, 2020.

[55] A. Arya and O. Chang, "ClusterFuzz: Fuzzing at Google scale," in *Blackhat Europe*, 2019.

[56] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," in *Working Conference on Reverse Engineering*, ser. WCRE '12, 2012.

[57] M. E. Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! Characterizing non-reproducible bug reports," in *Mining Software Repositories*, ser. MSR '14, 2014.

[58] N. E.-S. J. Gould and N. Eldredge, "Punctuated equilibria: An alternative to phyletic gradualism," *Essential readings in evolutionary biology*, pp. 82–115, 1972.

[59] M. D. Vose and G. E. Liepins, "Punctuated equilibria in genetic search," *Complex Systems*, vol. 5, no. 1, pp. 31–44, 1991.

[60] A. Gorshenev and Y. M. Pis'mak, "Punctuated equilibrium in software evolution," *Physical Review E*, vol. 70, no. 6, p. 067103, 2004.

[61] J. Wu, "Open source software evolution and its dynamics," Ph.D. dissertation, University of Waterloo, 2006.

[62] A. I. Antón and C. Potts, "Functional paleontology: System evolution as the user sees it," in *International Conference on Software Engineering.*, ser. ICSE '01, 2001.

[63] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: Fuzzing with a multi-task neural network," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, 2020.

[64] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Symposium on Security and Privacy*, ser. S&P '18, 2018.

[65] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," in *Symposium on Security and Privacy*, ser. S&P '19, 2019.

[66] Z. Sialveras and N. Naziridis, "Introducing Choronzon: An approach at knowledge-based evolutionary fuzzing," in *ZeroNights*, 2015.

[67] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Symposium on Security and Privacy*, ser. S&P '18, 2018.

[68] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Conference on Programming Language Design and Implementation*, ser. PLDI '17, 2017.

[69] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *USENIX Security Symposium*, 2017.

[70] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence." in *Network and Distributed Systems Security*, ser. NDSS '19, 2019.

[71] X. Zhu, X. Feng, X. Meng, S. Wen, S. Camtepe, Y. Xiang, and K. Ren, "CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation," *IEEE Transactions on Dependable and Secure Computing (to appear)*, 2020.

[72] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, 2020.

[73] S. Song, C. Song, Y. Jang, and B. Lee, "CrFuzz: Fuzzing multi-purpose programs through input validation," in *Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, 2020.

[74] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Conference on Computer and Communications Security*, ser. CCS '19, 2019.

[75] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *USENIX Security Symposium*, 2020.

[76] J. Ruohonen and K. Rindell, "Empirical notes on the interaction between continuous kernel fuzzing and development," in *International Workshop on Reliability and Security Data Analysis*, ser. RSDA '19, 2019.

[77] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[78] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security Symposium*, 2012.

[79] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? An empirical study of bug characteristics in modern open source software," in *Workshop on Architectural and System Support for Improving Software Dependability*, ser. ASID '06, 2006.

[80] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of C++ vulnerabilities in crowd-sourced code examples," *IEEE Transactions on Software Engineering (to appear)*, 2020.

[81] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Conference on Mining Software Repositories*, ser. MSR '12, 2012.

[82] ——, "Security versus performance bugs: A case study on Firefox," in *Conference on Mining Software Repositories*, ser. MSR '11, 2011.

[83] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[84] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014.

[85] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *International Conference on Software Engineering*, ser. ICSE '20, 2020.

[86] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *USENIX Security Symposium*, 2018.

[87] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows," in *International Conference on Software Engineering*, ser. ICSE '10, 2010.

[88] H. Wang and H. Kagdi, "A conceptual replication study on bugs that get fixed in open source software," in *International Conference on Software Maintenance and Evolution*, ser. ICSME '18, 2018.

[89] S. Neuhaus and T. Zimmermann, "Security trend analysis with CVE topic models," in *International Symposium on Software Reliability Engineering*, ser. ISSRE '10, 2010.

[90] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Workshop on Large-scale Attack Defense*, ser. LSAD '06, 2006.

[91] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *Conference on Software Maintenance, Reengineering, and Reverse Engineering*, ser. CSMR-WCRE '14, 2014.

[92] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *Mining Software Repositories*, ser. MSR '14, 2014.

[93] "2020 CWE top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2020, accessed January, 2021.

[94] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '18, 2018.

[95] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun, "Industry practice of coverage-guided enterprise Linux kernel fuzzing," in *Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '19, 2019.