The Journal of Systems and Software 216 (2024) 112119

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

BugOss: A benchmark of real-world regression bugs for empirical investigation of regression fuzzing techniques☆

Jeewoong Kim, Shin Hong *

Chungbuk National University (CBNU), Cheongju, Chungbuk, 28644, Republic of Korea

A B S T R A C T

This paper presents the design and the constitution of BugOss, a real-world regression bug benchmark for empirical study of regression fuzzing techniques. To reproduce the actual project context where a regression bug was introduced, each bug case of BugOss pinpoints the exact bug-inducing commit and provides a specific test oracle considering the presence of other co-existing bugs. BugOss currently comprises 20 real-world bug cases from 20 open-source C/C++ projects, which had been reported by the OSS-Fuzz projects and confirmed by the project maintainers. The empirical investigation with two regression fuzzing techniques show that, with the bug cases in BugOss, the regression fuzzing techniques perform differently depending on the given project context. In addition, the experiments imply that BugOss encompasses various cases of regression bugs in real-world, thus the bug cases would be useful for empirically investigating regression fuzzing techniques.

1. Introduction

Continuous fuzzing is a testing practice to periodically run fuzzers to generate new test inputs to guard the project against regression bugs along a series of program changes (Serebryany, 2016). OSS-Fuzz (Serebryany, 2017) has demonstrated the effectiveness of continuous fuzzing by detecting more than 49340 bugs from 859 and more open-source projects in first 5 years of its service. Continuous fuzzing is becoming increasingly popular as more projects are hosting fully automated continuous integration pipelines on cloud, and greybox fuzzers are advancing to better support this trend (Zhu and Böhme, 2021; Yoo et al., 2021; Klooster et al., 2023).

Regression fuzzing techniques utilize the information on program changes to quickly discover recently introduced failures. Zhu and Böhme (2021) proposes a change-aware power scheduling scheme to put more fuzzing effort to a code region if it is more recently changed or more frequently updated. Yoo et al. (2021) presents a technique to reuse the seed corpus of a previous version to fuzzing subsequent versions based on program change information. By selectively exploring changed behaviors, regression fuzzing techniques aim to quickly discover regression bugs, and reduce computation effort to re-test the unchanged behaviors.

Regression fuzzing is a promising direction for enhancing continuous fuzzing performance, and more investigations are expected to follow to improve regression fuzzing techniques. However, the existing fuzzing benchmarks (Zhu and Böhme, 2021; Hazimeh et al., 2021; Dolan-Gavitt et al., 2016; Zhang et al., 2022; Bundt et al., 2021) are not suitable for evaluating regression fuzzing techniques because their bug cases are artificially constructed based on bug-fix information (i.e., patches) while not reproducing actual commits that introduce bugs to the target projects.

This paper presents BugOss, a collection of real-world regression bugs with the package of information for experimenting regression fuzzing techniques. To help researchers reproduce realistic project context where continuous fuzzing performs against a bug-inducing commit, each artifact of BugOss indicates the exact commit where a regression bug was introduced in the version history, and provide a ground-truth condition to check whether a failure is induced by the regression bug, or other co-existing faults. We systematically extracted these pieces of information from the OSS-Fuzz issue tracker and the target project repositories to avoid uncertainty (Section 3).

Currently, 20 bug cases from 20 C/C++ programs are registered for the BugOss benchmark. To understand their characteristics, we conducted experiments with two general-purpose fuzzers and four regression fuzzing techniques (Section 5). From the experiment results, we found that BugOss encompasses various cases of real-world regression bugs which indicate the remaining challenges of regression fuzzing techniques. We believe that BugOss offers researchers a useful basis of empirical investigation of regression fuzzing techniques, as it is publicly available at the following site:

https://github.com/sdevlab/BugOss

The main contribution of this paper are as follows:

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: jeewoong@chungbuk.ac.kr (J. Kim), hongshin@chungbuk.ac.kr (S. Hong).

https://doi.org/10.1016/j.jss.2024.112119
Received 13 August 2023; Received in revised form 30 May 2024; Accepted 1 June 2024
Available online 15 June 2024

- To the best of the authors' knowledge, BugOss is the first benchmark with actual bug-inducing commits of real-world C/C++ projects under continuous fuzzing for empirical study of regression fuzzing techniques. All artifacts of the benchmark and the experiment results are publicly open for future research.
- We present a systematic procedure to construct regression bug benchmarks based on code repository and issue tracker information. We propose a new method to collect failure cases and derive a bug-specific test oracles.
- We provide the experiment results with two general-purpose and four regression fuzzing techniques that explore the characteristics of the bug cases in BugOss. The experiment results also imply that BugOss is useful to evaluate regression fuzzing techniques and identify their limitations.

The remaining sections of this paper are as follows. Section 2 explains the background and the related work. Section 3 describes the structure of the bug cases in the BugOss benchmark and the detailed procedure of the bug case artifact construction. Section 4 presents the current constitution of the BugOss benchmarks and provides the detailed information about the 20 bug cases. Section 5 describes the empirical studies of BugOss with six fuzzing techniques. Section 6 discusses our observations on BugOss, and Section 7 concludes this paper.

2. Background and related work

2.1. Continuous fuzzing and regrssion fuzzing techniques

Continuous fuzzing is an automated verification practice that periodically (e.g., more than once every 24 hours, Serebryany, 2017) conducts fuzzing throughout the project's evolution to swiftly detect and eliminate adverse program changes. Unlike continuous integration, which synchronously runs static bug checkers against each commit, continuous fuzzing operates asynchronously with continuous integration. This is because greybox fuzzers require a sufficiently long running time (e.g., 24 hours, Klees et al., 2018) to explore different target program paths by generating numerous inputs stochastically.

One limitation of continuous fuzzing is that a significant portion of fuzzing efforts is expended on repeatedly testing unchanged program paths over the project's history (Klooster et al., 2023). Even though only a small portion of the production code changes during a fuzzing duration, the current practice of continuous fuzzing utilizes conventional fuzzers that aim to explore various program paths regardless of how the target program has changed.

Regression fuzzing techniques have been proposed to consider the project context in fuzzing to alleviate this inefficiency in continuous fuzzing practice. AFLChurn (Zhu and Böhme, 2021) is a directed greybox fuzzer that refers to code change history to identify which regions of the target program have been recently introduced and more frequently updated. It then guides a fuzzing run towards the identified critical regions of the program. The key idea of AFLChurn is a power scheduling algorithm that assigns more power to a seed if the program path toured by the seed contains recently changed regions or frequently updated regions.

Another approach proposed by Yoo et al. (2021) is change-aware seed reuse for continuous fuzzing. This approach suggests reusing the seed corpus generated in an earlier version to direct fuzzing of the next version towards changed code regions. When a new version is subjected to continuous fuzzing, the suggested technique first identifies the regions of the target program that changed after the previous fuzzing run. It then collects the previously generated test inputs that explore the corresponding regions in the previous fuzzing run and initiates a new fuzzing run with the collected test input as the initial seed corpus. This approach is based on the assumptions that if a test input reaches a region in the previous version of the target program, the test input will likely reach the same region after the code changes. Additionally, a fuzzing run will likely explore changed program behaviors if test inputs reaching the changed code regions are used as initial seeds.

2.2. OSS-Fuzz service

OSS-Fuzz (Serebryany, 2017) is a service operated by Google offering continuous fuzzing to well-known open-source projects to promptly detect security vulnerabilities and bugs as they are induced. Each of the hosted open-source projects registers one or multiple fuzzing drivers (i.e., fuzzing targets and fuzzing configurations) that OSS-Fuzz runs fuzzers (e.g., libfuzzer and AFL) with them on a regular basis. Using the registered fuzzing drivers, the recent version of the target projects at GitHub undergo automated testing using the initial seed corpus maintained by OSS-Fuzz.

Once a failure is detected, OSS-Fuzz generates a bug report and privately shares it to the corresponding maintainers. Only after few months, OSS-Fuzz made a bug report public at a separate issue tracker,[1] since it may expose the latest weakness or the bug information to potential attackers. Upon receiving a bug report from OSS-Fuzz, a target project maintainer may change the program to resolve the concerned issues. Since OSS-Fuzz does not post a bug report to the target project's issue tracker or sends a pull request, the traceability link between the bug report and the corresponding program change is unclear (Keller et al., 2023) unless the maintainer explicitly references the OSS issue number or mentions the specific failure identified by OSS-Fuzz in the commit message, self-registered issue, or pull request.

An issue report automatically generated by OSS-Fuzz provides details about the observed failures in a specific format. An issue report outlines the failure symptom including crash type, crash state and the involved sanitizer if exists. Additionally, an issue report includes a failure-inducing test input, enabling the developers to reproduce the failure by running the attached test input on the respective fuzzing driver.

Note that, as a fuzzing driver is integrated with the target project, we can check the test results with the same input on a series of program versions, thus we can pinpoint the exact bug-inducing commit if the bug was introduced after the fuzzing driver had been added to the project. To facilitate the identification of the bug-inducing commit, OSS-Fuzz conducts bisection and provides a suspected range of the bug-inducing commit. Furthermore, for a reported issue, OSS-Fuzz routinely checks if the failure is reproduced in a recent version. If the failure is not reproduced, OSS-Fuzz adds a note to the issue report with a commit number for which it witnesses the failure is not reproduced, because the bug causing the failure would be fixed before the commit. We can pinpoint the exact bug-fix commit by testing all preceding program versions rigorously.

2.3. Fuzzing benchmark

The fuzzing research community has developed benchmarks for sound empirical assessments of a variety of research attempts scientifically and to guide research efforts towards improving the techniques. For example, FuzzBench (Metzman et al., 2021) provides open-source fuzzer benchmarking service using massive computation resources to periodically evaluate how the state-of-the-art fuzzers perform with 24 target programs as they evolve.

A key challenge in benchmark construction is to achieve diversity of bug cases, encompassing various bug types and locations, while preserving authenticity of the bugs to ensure they share characteristics with real bugs. The synthetic approach addresses this challenge by generating a large number of buggy versions controlling the diversity of bug cases (Dolan-Gavitt et al., 2016; Lee et al., 2023; DARPA Cyber Grand Challenge Repositories, 2017). For example, LAVA (Dolan-Gavitt et al., 2016) provides a benchmark by synthesizing many buggy versions of eight open-source programs by injecting different types of bugs over different code locations, where security bugs are injected to

[1] https://bugs.chromium.org/p/oss-fuzz/

Table 1
Comparisons of fuzzing benchmarks.

Name	# Proj.	LoC	# Ver.	# Bugs	Bug origin	Bug-specific test oracle	Co-existing bug info.	Seed corpus
LAVA-M (Dolan-Gavitt et al., 2016)	4	2K–4K	4	2265	Synthesized	No	No	Arbitrary
Rode0day (Bundt et al., 2021)	11	1K–160K	11	978	Synthesized	No	No	Arbitrary
FuzzBench (Metzman et al., 2021)[a]	24	3K–1422K	28	28	BIC	No	No	BIC
UniFuzz (Li et al., 2021)[b]	20	4K–1126K	20	20	BIC	No	No	Arbitrary
Magma (Hazimeh et al., 2021)[c]	9	294K–1830K	9	138	Migrated	Yes	Yes	Latest
AFLChurn (Zhu and Böhme, 2021)	15	16K–601K	15	15	Migrated	No	No	Latest
BugOss	20	18K–1680K	20	20	BIC	Yes	Yes	BIC

[a] commit b6d7a9c (2024-03-07) at URL https://github.com/google/fuzzbench
[b] commit 834f297 (2021-03-17) at URL https://github.com/unifuzz/unibench
[c] commit 75d1ae7 (2022-12-08) at URL https://github.com/HexHive/magma

evaluate fuzzing techniques with diverse bug cases. Fuzzle (Lee et al., 2023) proposes to synthesize ground-truth buggy programs modeling realistic path constraints to evaluate fuzzing techniques.

While these synthetic approach effectively manages bug case diversity, there remains uncertainty about whether synthetic bugs accurately represent real bugs introduced in practical scenarios. An empirical study (Bundt et al., 2021) utilizing two synthetic benchmarks, LAVA-M (Dolan-Gavitt et al., 2016) and Rode0day (Fasano et al., 2019; Bundt et al., 2021), discovered that fuzzers often exhibit different performance in failure detection when dealing with synthetic bugs compared to real bugs collected from actual bug-inducing commits.

When assessing regression fuzzing techniques, synthetic benchmarks would encounter significant limitations as they do not provide real project context such as version histories and previous fuzzing results. Our observations indicate that the performance of regression fuzzing techniques depends on the provided project contexts (see Sections 5.2.3 and 5.2.4). Consequently, utilizing synthetic benchmarks introduces further validity threats in the evaluation of regression fuzzing techniques.

FuzzBench extends Google Fuzzer Test Suite (a.k.a. FTS) (Google fuzzer-test-suite (FTS) benchmark, 2016) to incorporate 24 real-world open-source programs for assessing the performance of various greybox and whitebox fuzzing tools (Metzman et al., 2021). UniFuzz (Li et al., 2021) presents a benchmark of 20 real bugcases to evaluate various fuzzing techniques. These benchmarks are free from the validity threats of employing artificial bugs, because only real-world bug-inducing commits found in the real-world projects are used. One technical challenge of these real bug benchmarks is managing potential interference of identified or unidentified co-existing bugs (Böhme et al., 2022; An et al., 2021).

BugOss takes a similar approach by exclusively leveraging bug-inducing commits found in the real-world. We present a systematic process to construct bug-specific test oracles and other failure's test oracles to determine whether a failure found by a fuzzer is caused by the target bug (i.e., the bug newly induced by the bug-inducing commit under fuzzing). We observed that these test oracles are effective in the experiments (see Sections 5.2.2 and 6.2). In addition, considering the possible gap between OSS-Fuzz issues and project commit history, we propose a systematic procedure for constructing benchmark bug cases that maintain traceability.

Magma (Hazimeh et al., 2021) constructs a benchmark using the forward-porting approach. For each of the seven open-source projects, they identified multiple security-relevant bugs, and then re-inserted the identified bugs to a single version while adding bug-specific assertions to identify whether a test input reaches the bug location and the infection occurs. As the benchmark employs real bugs and real-world projects, Magma attains both diversity and authenticity of bug cases at the same time. And, potential interference of multiple bugs can be managed by having bug-specific assertions. However, in evaluating regression fuzzing techniques, Magma shares a limitation with the synthetic approaches that it lacks real project context information since multiple bugs are injected at earlier versions.

Table 1 contrasts BugOss with the existing fuzzing benchmarks at a glance. The columns labeled "# proj.", "# ver." and "# bugs" present the number of projects, versions and bugs included in each benchmark, while "LoC" indicates the range of the program sizes in terms of Lines of Code (LoC). The "bug origin" column specifies whether the target bugs are artificially synthesized, migrated from bug-fixing commits (i.e., reversing actual bug-fixes), or collected from actual bug-inducing changes as they are ("BIC"). The "bug-specific test oracle" column indicates whether a benchmark provides information or mechanism to determine whether a failure is caused by the target bug. Similarly, the "co-existing bug info." column indicates whether the information about the co-existing bugs or their failures is provided, if they exist. Finally, the "seed corpus" column represents the source of the initial seed corpus: "arbitrary" means that the given initial corpus is constructed regardless of the target project, "latest" maens the given corpus is obtained from the latest version of the project, and `BIC` means the given corpus is one that used in the project when the BIC occurred.

Table 1 shows that, unlike the other benchmarks, BugOss provides comprehensive information for reconstructing real project context where a regression bug is newly introduced, and a fuzzing run is conducted against it. Reconstructing the bug-inducing context is important for empirically studying how fuzzers perform in detecting regression errors (see Sections 5.2.3 and 5.2.4). Additionally, when a target version is not latest one, but a previous one (i.e., right after the bug-inducing change), the effects of co-existing bugs should be systematically managed in experiments (see Section 5.2.2). For these reasons, we believe that BugOss complements the existing fuzzing benchmarks with its specific design for facilitating empirical study on regression fuzzing.

3. Benchmark design and construction

3.1. Artifact structure

Each artifact of a bug case in BugOss is purposed to study how the fuzzer performs in generating a test input that reveals a specific target bug (i.e., an input that triggers a target bug to induces a failure) when the bug was actually introduced in a real-world project. To capture real-world bug-inducing situations, we initially gathered OSS-Fuzz issues that led the project maintainers to recognize and fix previously unidentified bugs in their projects. For each gathered issue, we systematically reviewed the target project's commit history and the related OSS-Fuzz issues to identify the bug-inducing commit and also the bug-fixing commit, and collected the information to determine the symptom of the target bug (i.e., bug-specific test oracle). Specifically, the artifact of a bug case in BugOss consists of the following elements:

- **bug-revealing input**: an input for each fuzzing target, that induces a failure. This input is given as the failure-reproducing input in the OSS-Fuzz issue.

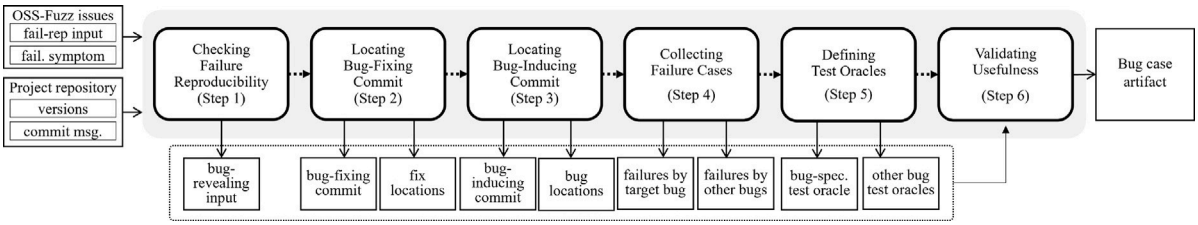

Fig. 1. Bug artifact construction process.

- **bug-inducing commit**: the program change that newly adds the target bug to the target program. For simplicity, we call the program versions before and after the bug-inducing commit as the *bug-free* version and the *first buggy* version, respectively.
- **bug-fixing commit**: the program change that repairs the target bug (i.e., fix-inducing commit). We call the immediately following version of the bug-fixing commit as the *fixed* version and the immediate preceding version as *last buggy* version (or *pre-fix* version), respectively.
- **bug locations**: a subset of the changed lines in the bug-inducing commit, that are suspected to result in a failure when the bug-revealing input is given.
- **fix locations**: a subset of the changed lines in the bug-fixing commits, that are related to resolving failures identified by the bug-revealing input. Note that fix locations and bug locations are originated from independent sources, thus they may or may not be overlapped.
- **bug-specific test oracle**: conditions to determine whether a failure is induced by the target bug, or by the other bugs based on failure symptom (e.g., crash message, stack trace, execution time). These conditions are provided executable forms (e.g., sanitizer setting, script), so that these can be used in automated processes.

Note that, even without bug-specific test oracles, it is possible to determine if the input reveals a target bug by executing the bug-free version, the first buggy version, the last buggy version and the fixed version with the same test input, and the bug-free version with the same test input (i.e., an input reveals the target bug if the first buggy version and the last buggy version fail with the input while the bug-free version and the fixed version do not fail). However, employing such a determination approach demand to much testing effort to be integrated into a continuous fuzzing process because multiple versions of the same project must be used. BugOss offers bug-specific test oracles as conditions over failure symptoms to efficiently determine whether a fuzzer reveals a target bug.

3.2. Construction process

We constructed BugOss by tracking the information about the target bugs from the failure information reported by the OSS-Fuzz issues. Although a OSS-Fuzz issue gives clues about when the target bug was induced and fixed (e.g., bijection and disclosure), manual inspection is required to identify exact information about the target bug, and reject unsuitable cases.[2] To systematically construct benchmark artifacts, we had taken the following six steps for each bug case (see Fig. 1):

1. **Checking failure reproducibility.** For a OSS-Fuzz issue, we retrieved the latest version before the issue report time, and identified the failure-reproducing input attached with the issue as the bug-revealing input. Note that we used the issues automatically generated by OSS-Fuzz only, thus these issues provide full details including failure symptoms (e.g., crash type, crash stack trace) and failure-reproducing inputs. Subsequently, we checked if a failure occurs as expected, when fuzzers run the fuzzing driver (i.e., fuzzing target) with the failure-reproducing input attached in the OSS-Fuzz issue. We rejected the issues where failure reproduction was not successful with one of the fuzzers (or both). Also, we rejected the issues if the failure symptoms are different with the issue reports.
2. **Locating bug-fixing commit.** We iterated over the versions after the issue report time, until the bug-revealing input does not induce the expected failure. Once we spotted the first version where the failure does not occur, we looked for the evidence that the commit is to resolve the issue. If we found that the commit message written by the maintainer or the related documentation mentions OSS-Fuzz or the particular failure symptom explicitly, we confirm that the commit is the bug-fixing commit intended by the developers. We rejected the issues if we failed to find any clear evidence, or if the fix is not yet made, because the last buggy version and the fixed version may be used to examine an unidentified failure symptom (see Section 3.3).
3. **Locating bug-inducing commit.** From the issue report time, we checked the preceding versions in a backward manner to pinpoint the version where the expected failure occurs for the first time. As we are able to test multiple versions against the failure-reproducing inputs, we could identify a bug-inducing commit as the exact commit where the failure is first reproduced, instead of using the estimation methods based on the code change information (Siliwerski et al., 2005), test coverage (An et al., 2023) or IR-based techniques (Wen et al., 2016). For reproducing the failures, we used the same fuzzing drivers which are used in Step 1. In case where the fuzzing driver has changed over versions, we transplanted the fuzzing driver to earlier versions. Note that, as the same fuzzing driver is used for all examined versions, we confirmed that the failure violates the property that had been satisfied before the bug-inducing commit (i.e., regression failure).
   Once the first failing version is located, we proceeded to confirm that the corresponding commit is suspected to induce the target bug. We manually checked that the suspected commit and the bug-fixing commit (Step 2) concern the same code lines, the same data structure, or they are associated indirectly. If we could not confirm that they are related, we excluded the issues from the benchmark construction to avoid possible uncertainty. We also rejected an issue if the located commit changes only build scripts, configuration files, or fuzzing drivers; we suspect such a change affects the reachability of existing bugs, rather than inducing a new bug.
   Once the bug-inducing commit is confirmed, we retrieved the seed corpus at the bug-inducing time based on the OSS-Fuzz configuration. In addition, we determined the bug locations and the fix locations by comparing the changed lines of the bug-inducing commit and the bug-fixing commit.
4. **Collecting failure cases.** Since a program version is likely to contain multiple bugs (An et al., 2021), each bug case of BugOss provides a bug-specific test oracle which discriminates the failures by the target bug (i.e., "target failures") from the failures by the other bugs (i.e., "other failures").

[2] The manual inspection (for Steps 2 and 3) involved the participation of five graduate students and one professor, including the authors. We verified the evidence when a consensus was reached among all participants.

To collect the references for defining a bug-specific test oracle, we collected the two kinds of failure cases by the following process. First, we ran the general-purpose fuzzers with the bug-free version for 48 hours to collect the failures induced by pre-existing bugs before the bug-inducing commit. If a failure is observed with the bug-free version, we considered such failure cases as the failures by the other bugs.

Second, we gathered failure-reproducing inputs of the OSS-Fuzz issues registered between the bug-inducing commit and the bug-fixing commit, which fail the first buggy versions. For each gathered failure-inducing input, we identified the survival range in the version history that the input actually causes the failure. If the range of a failure-reproducing input is identical to that of the bug-revealing input, we considered that the failure-inducing input is redundant to the bug-revealing input, and accepted its failure as one case of the failure by the target bug. Otherwise, we regarded the cases as the failures by other bugs (see more details at Section 3.3).

5. **Defining bug-specific test oracles and other-failures test oracles.** Given two kinds of failure cases of a bug case artifact, we defined a bug-specific test oracle, a condition that discriminates the failures by the target bug and the failures by the other bugs. A bug-specific test oracle is defined over the following failure symptoms: specific failure types, stack trace patterns, coverage of specific branch condition, or their combinations (see Section 3.3). Once a failure discriminating condition is identified, we encoded the condition as an executable form (e.g., sanitizer setting or script for checking error messages) such that the bug-specific test oracle can be automatically checked in a fuzzing process. If we could not find a condition of the failure symptoms that differentiates target failures and other failures, we rejected the cases to limit the threats by inaccurate test oracles.

   In addition, we also define a test oracle for each of the other failures, following the same procedure as for the bug-specific test oracle (see Section 3.3). Note that bug-specific and other-failure test oracles are inferred test oracles, thus they may result in false positives or false negatives (see more discussion on this point at Section 6.2).
6. **Rejecting useless cases.** Lastly, we rejected the case if, for the first buggy version, two conventional fuzzers, AFL++ and libfuzzer, always generate a bug-revealing input within first 3 minutes of a fuzzing run. We believe that such bug cases are not useful for empirically investigating the performance of fuzzers. Also, we rejected the bug cases if the bug-inducing commit changes more than 300 lines of code, because such a large code modification may involve major functionality changes and not suitable subject of regression testing.

Currently, BugOss encompasses 20 bug cases acquired from 20 open-source C/C++ projects hosted by OSS-Fuzz. In order to construct these artifacts, we had reviewed total 2074 OSS-Fuzz issues from 65 open-source projects. We rejected 2054 issues in the middle of the construction process. In the first step, 53 issues were rejected because we failed on our testing environment to reproduce the described failures with bug-reproducing test inputs offered by OSS-Fuzz. We conjecture that the reproduction was not possible due to unspecified dependencies.

In the second step, we excluded 1296 issues for which we cannot locate bug-fixing commits with clear traceability links to the corresponding bug-inducing commits (e.g., a mention in the commit message). We excluded these issues because there could be different reasons causing the failures to disappear (see Section 6.3). During the third step, 622 issues were dropped out because the failures are reproduced even at the earliest versions that we can conduct fuzzing. Finally, in the sixth step, we discarded 83 issues. Among the 83 issues, we determined 17 are not useful because the corresponding failures occur promptly after fuzzing starts. In addition, we concluded that 66 are not useful because their bug-inducing commits involve significant code changes which seem to require comprehensive fuzzing for retesting all components, instead of regression fuzzing.

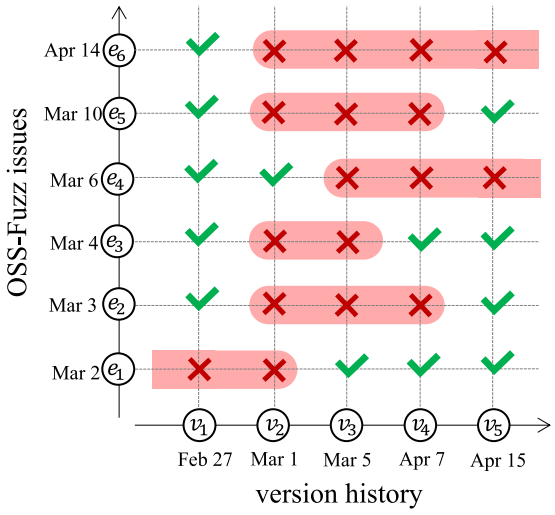

Fig. 2. Examples of target failures and other failures.

3.3. Test oracle construction

A bug-specific test oracle is a predicate over the failing execution that accurately determines whether the failure is caused by the target bug or by the other bugs. A bug-specific test oracle is essential for regression fuzzing experiments to assess whether a fuzzer effectively reveals a target regression bug in the presence of other co-existing bugs.

To construct bug-specific test oracles of a bug case, we explored the possibilities of co-existing bugs by testing the bug-free version and checking OSS-Fuzz issues. And, we systematically collected the failures by the target bugs and the failures by the other co-existing bugs (Step 4 in Section 3.2). Fig. 2 describes the failure case collection with examples. The x-axis lists a series of the target program versions, v_1 to v_5 whereas the y-axis lists a series of OSS-Fuzz issues with failure-reproducing inputs, e_1 to e_6. Each date label declares the time when the corresponding version or issue is registered. A mark given to v_i and e_j indicates whether the test with e_j on v_i is failing (marked with red cross), or passing (marked with green check).

Suppose that we are constructing a bug case artifact with bug-revealing input e_5 whose survival range is between v_2 and v_4. Given bug-revealing input, we examined the failure-reproducing inputs reported in Mar 1st to Apr 15th, which are e_2 to e_6. We consider that e_3, e_4, and e_6 are resulted by the other bugs because they are not introduced at v_2 (e_4), or not resolved at v_5 (e_3, e_4, e_6). Meanwhile, e_2 is considered as another bug-revealing input, thus the failure with e_2 is classified as the failure by the target bug. Last, e_1 shows a failure-reproducing input related to a pre-existing bug. These failure cases can be collected by fuzzing the bug-free version.

For a given target failure and other failure cases, we systematically derived a bug-specific test oracle based on the following criteria:

- if we are given only target failures (i.e., no other failures), we defined the bug-specific test oracle as the same as the general test oracle which accepts all crashes and timeout identified by a fuzzer as failures by the target bug.
- if the target failures have different failure types (e.g., heap-buffer overflow and null-pointer dereference) with all other failures, we define the bug-specific test oracle to accept a failure if and only if it has the same failure type with a target failure.
- if target failures and other failures have the same failure type, we first determine the first frame in their stack traces (from the top frame) where the target failures have different functions with the other failures. And then, we used the stack trace condition in the bug-specific test oracle.

Table 2
Bug cases in the BugOss/ benchmark.

Name	Project description	Project size (LoC)	Failure type	Seeds	Changed lines		BIC→	Failure cases		Test oracle type
					BIC	BFC	BFC	Target	Others	
arrow-40653	Columnar data analyzer	889K	Abort	10	22	86	230	2	4	Failure type
aspell-18462	Text spell checker	54K	Buf-overrun	2	5	18	87	1	1	Stack-1
curl-8000	Network data transfer tool	184K	Buf-overrun	4202	51	2	1	49	0	All
exiv2–50315	Image metadata editing tool	386K	Int-overflow	454	45	3	18	1	13	Failure type
file-30222	File type checker	18K	Null-deref	34	21	11	0	1	7	Failure type
gdal-47716	Geospatial data translator	1680K	Buf-overrun	1615	10	4	8	1	0	All
grok-28418	Image compression tool	179K	Mem-leak	178	101	55	3	1	23	Stack-2
harfbuzz-55779	Text-to-font rendering library	135K	Assert violat.	1003	105	10	13	1	0	All
leptonica-25212	Image processing library	201K	Null-deref	10	25	26	5	1	1	Stack-1
libarchive-44843	Multi-format archive library	168K	Null-deref	13	46	13	7	1	6	Failure type
libhtp-17198	HTTP protocol parser	40K	Buf-overrun	97	26	29	3	3	0	All
ndpi-49057	Deep packet inspector	93K	Int-overflow	351	51	10	5	1	16	Failure type
openh264–26220	H.264 codec library	140K	Buf-overrun	174	7	5	19	10	0	All
openssl-17715	SSL/TLS cryptographic tool	592K	Buf-overrun	2240	91	47	267	2	0	All
pcap++-23592	Packet processing library	59K	Buf-overrun	615	32	13	130	1	11	Stack-8 & Branch
poppler-35789	PDF file rendering library	198K	Null-deref	476	3	20	4	5	0	All
readstat-13262	Statistics data converter	31K	Buf-overrun	94	5	10	44	1	5	Stack-5 & Branch
usrsctp-18080	SCTP protocol library	85K	Use-aft-free	156	6	8	9	1	2	Stack-1
yara-38952	Malware detector	63K	Buf-overrun	9	277	17	0	1	0	All
zstd-21970	Data compression tool	104K	Null-deref	12462	280	247	53	1	0	All

- if a target failure and one other failure have the same failure type and stack trace, we checked if there exists any branch condition in the bug locations and the fix locations that are satisfied only with the target failures. If such branch condition exists, we used the coverage of the branch condition in a bug-specific test oracle.

To reduce false negatives, we defined the bug-specific test oracle as general (i.e., weak) as it discriminates the given target failures from the given other failures.

A fuzzer may discover a failure caused by a previously unknown bug in the target projects. To identify such unknown cases, BugOss also provides a test oracle for each of the other failures. Constructions of these other bug test oracles follow the criteria as the bug-specific test oracle, while only one failure is used for constructing one test oracle. When a fuzzer detects a failure, the bug-specific test oracle is examined. If the test oracle shows negativity, each test oracle of the other failures are examined. If none of these oracle gives a positive result, the failure is identified as unknown.

For example, for a target bug b, if we found failures f_1 and f_2 caused by other co-existing bugs, BugOss offers test oracles c_b, c_{f_1} and c_{f_2}. If a failure does not satisfy c_b, the user can check if it satisfies either c_{f_1} or c_{f_2}; If none of the test oracle is satisfied, this fact signals the user that the failure is different from all previously known cases. Then, the user may run the failure-inducing input on the different versions related to the target bug (i.e., bug-free version, first buggy version, last buggy version, and fixed version) to conclude whether the failure is a symptom of the target bug.

A unknown failure may result from an unidentified bug or an unreported symptom of an existing bug. Despite the limitation as test oracles, as BugOss provides the bug-inducing commit (i.e., bug-free version and first buggy version) and the bug-fixing commit (i.e., the last buggy version and the fixed version), it is possible to determine whether or not the unknown failure is caused by the target bug; we observed such cases in the experiments (see Section 6.2).

4. BugOss benchmark

We have registered 20 bug cases for BugOss, as shown in Table 2. These artifacts are built based on 20 real bugs that OSS-Fuzz had detected from 20 well-known open-source C/C++ projects. In Table 2, the first column names each bug case by combining the project name and the OSS-Fuzz issue number. The second column shortly describes the functionalities of the projects, and the third column gives the total sizes of the project source code. The fourth column describes the failure types of the target bugs. The fifth column shows the number of seed inputs at the bug-inducing commits.

The columns with 'changed line' shows the number of changed lines in the bug-inducing commit ('BIC') and the bug-fixing commit ('BFC'). The 'BIC→BFC' represents the number of commits between the bug-inducing commit and the bug-fixing commit. The columns with 'failure cases' give the number of the failures by the target bug (i.e., target failures), and the number of the failures by the other bugs (i.e., other failures) used for defining the bug-specific test oracles.

Last column marked as 'test oracle type' describes which failure symptoms are used for defining the bug-specific test oracles. For the bug cases with 'all', the test oracle is the same as the general test oracle. 'failure type' means that the bug-specific test oracles uses the failure type information in the error message to determine whether a failure is caused by the target bug or the other co-existing bugs. 'stack-N' represents that the bug-specific test oracle uses the top-N frames of the stack trace. The bug cases with 'stack-N & branch' (i.e., pcap++-23592 and readstat-13262) have the test oracles that check the satisfaction of the specific branch conditions in addition the top-N stack trace information.

Table 2 shows that the bug cases in BugOss has a diversity of projects and bug characteristics. The benchmark encompasses a variety kinds of projects. The scale of the project spans from 18KLoC to 1680KLoC. The number of lines changed by bug-inducing commit is small (less than 10) for five, moderate (10 to 50) for other eight, and large (more than 50) for the other seven bug cases. The numbers in the 'BIC→BFC' column estimate the time-to-fix aspects (Ding and Le Goues, 2021). Among the 20 bug cases, 11 have less than short time-to-fix durations (less than 10 commits), while the other nine have moderate to long time-to-fix durations. The numbers of the failure cases imply that nine bug cases are of the single-bug case, while each of the other 11 bug cases involve multiple bugs in its first buggy version.

One notable point in Table 2 is that, in most artifacts, the bug-inducing commits change more lines of code than the corresponding bug-fixing commits. This result implies that bug-fixing commits indicates the bug locations more specifically than real bug-inducing commits. Another difference between bug-inducing commits and bug-fixing commits is the seed corpus. We found that, for eight bug cases, the seed corpora of BIC and BFC are different. For example of *aspell-18462*, the seed corpus at BIC consists of two inputs, while the seed corpus at BFC has 60 inputs. By indicating actual bug-inducing commits, BugOss effectively limits possible threats and provides realistic experiment setup for evaluating regression techniques that utilize code change information.

5. Empirical investigation

5.1. Experiment design

5.1.1. Research questions

We conducted experiments with different fuzzing techniques to understand the characteristics of the existing BugOss bug cases, and to explore how the choice of code change information and test oracles influences fuzzing performances. We devised the following four research questions:

- RQ 1. How effectively and efficiently does a fuzzer generate a test input that reveals a target bug?
- RQ 2. To what extent the failure detection performance of a fuzzer changes if a bug-specific test oracle is not used?
- RQ 3. To what extent the failure detection performance changes if a regression fuzzer is given with the code change information of the bug-fixing commit, instead of the bug-inducing commit?
- RQ 4. To what extent the failure detection performance changes if a regression fuzzer runs on the last buggy version, instead of the first buggy version?

RQ 1 is to check if BugOss encompasses various bug cases to compare regression fuzzing techniques. RQ 2 is assessing the effect of using bug-specific test oracles in BugOss. RQ 3 and RQ 4 are devised to study how the experiment results change if bug cases are not obtained from actual bug-inducing commits.

To answer RQ 1, we ran the following five fuzzing configurations based on three fuzzers and two fuzzing techniques against the 20 bug cases to find whether the target bug is detected within 36 core-hours (= 6 hours × 6 cores) and how much time is taken for generating the first bug-revealing test input:

- libfuzzer (Serebryany, 2016) of llvm-14.0.0 as a general-purpose fuzzer
- AFL++ version 4.05c (Fioraldi et al., 2020) as a general-purpose fuzzer
- AFLChurn (Zhu and Böhme, 2021) of commit 194e18c. AFLChurn has change-aware power scheduling for regression fuzzing. We carefully instructed AFLChurn to target all changed lines of the bug-inducing commits while not targeting the changed lines of the other commits.
- CSR-libfuzzer as an in-house implementation of the change-aware seed reuse technique (Yoo et al., 2021) upon llvm-14.0.0. We configured libfuzzer to run fuzzing with the bug-free version for 36 core-hours and then re-use all generated inputs that cover a changed function for fuzzing the first buggy version.
- CSR-AFL++ as an in-house implementation of the change-aware seed reuse technique (Yoo et al., 2021) upon AFL++ version 4.05c (Fioraldi et al., 2020). As similar for CSR-libfuzzer, we used AFL++ to run fuzzing with the bug-free version for 36 core-hours and then re-use all generated inputs that cover a changed function for fuzzing the first buggy version.

We run AFL++, AFLChurn and CSR-AFL++ with the distributed mode using one core for master (`-M`) and five for workers (`-S`) and sharing the seed corpus through a fuzzing campaign. We run libfuzzer and CSR-libfuzzer with the fork mode using six cores.

To answer RQ 2, we compared the failure detection results of the studied fuzzers with the bug-specific test oracles and without the bug-specific test oracles. The fuzzing results of RQ 1 represent the fuzzing performance with the bug-specific test oracle. To obtain the results without using the bug-specific test oracles, we determined that a fuzzer generates the bug-revealing input of a target bug only if a failure occurs regardless of its failure type or another condition (i.e., general test oracle).

To answer RQ 3, we ran fuzzing of the studied regression fuzzing techniques (i.e., AFLChurn, CSR-libfuzzer and CSR-AFL++) with the changed functions determined with the bug-fixing commits (not with the bug-inducing commits as RQ 1). For re-using seeds, CSR-libfuzzer and CSR-AFL++ are given with the inputs generated by libfuzzer and AFL++, respectively, for the bug-free version, as the same with RQ 1.

To answer RQ 4, we ran fuzzing of the studied regression fuzzing techniques on the last buggy version instead of the first buggy version (the immediately preceding version of the bug-fixing commits) with the code change information with the bug-fixing commits. Similar to RQ 3, CSR-libfuzzer and CSR-AFL++ use the seeds generated by libfuzzer and AFL++, respectively, for the bug-free version.

5.1.2. Experiment setup and measurements

A fuzzing run is configured to use six cores for six hours (i.e., 36 core-hours). Each fuzzer is set to fulfill this running time despite of failure detections. All experiments were performed with Intel i5-10600 3.30 GHz and 16 GB RAM running Ubuntu 20.04.3 LTS. For each setup, we repeated fuzzing runs for 10 times and then measured the average results.

For each fuzzing run, we measured the fault detection and the first time to fault detection. The fault detection is true if and only if the execution of at least one of the generated input satisfies the configured test oracle (the bug-specific test oracles, or the general test oracle for RQ 2). The first time to fault detection is measured as the time difference between the fuzzing starts and the first generation of a bug-revealing input. For computing the average time, we count 36 core-hours (i.e., total running time) if a fuzzing runs does not generate any bug-revealing input.

5.1.3. Threats to validity

External threats. An external threat is that the experiments involve only five fuzzing techniques, thus the results may be different with other fuzzing techniques. Although we could not use many general-purpose fuzzers, we employed two most widely-used fuzzers both in research and practice. Among many directed fuzzing techniques (Böhme et al., 2017; Lemieux et al., 2018), we used AFLChurn and change-aware seed reuse technique, and their combinations, as these two techniques specifically concern regression fuzzing techniques.

Another external validity threat is that the number of the target projects and bugs are limited. While the number of bugs in BugOss (i.e., 20) is mostly small than the ones with synthesized or migrated bugs (see Table 1), it is comparable to the existing benchmarks leveraging actual bug-inducing changes (i.e., 24 for FuzzBench, and 20 for UniFuzz). Furthermore, compared to the most closely related work, AFLChurn, BugOss incorporates more projects and bug cases, thereby more effectively mitigating external threats.

Also, there is a potential threat that the selection may be biased due to the conservative procedure (see Section 3.2). Specifically, requiring the presence of a traceability link between BIC and BFC (at Step 2) could introduce a bias in selecting the target bugs. Through this conservative selection, we believe that we can effectively mitigate another potential threat posed by errors in BIC and BFC information.

The other external threat is that the experiments use limited seed corpuses. Although these seed corpuses may not be complete, these are still effective for reproducing the regression fuzzing context that OSS-Fuzz had at the bug-inducing commits, because these are obtained from the OSS-Fuzz configuration at the corresponding commits.

Internal threats. Our in-house implementation of the change-aware seed re-use techniques and the tooling for the experiments may have errors. To limit this threats, we put our best efforts to review all experiment scripts and the tool implementation. Also, we re-used the original configuration of OSS-Fuzz as much as possible.

Construction threats. We conducted each fuzzing run for 36 core-hours. This timeout setup gives more time to each fuzzing run than the original experiment of AFLChurn (Zhu and Böhme, 2021) which gives 23 core-hours (i.e., using one core for 23 hours). The fuzzing results may change if the fuzzers run longer. However, in our preliminary

Table 3
Fuzzing results with the BugOss/ bug cases (RQ 1).

Name	libfuzzer		AFL++		AFLChurn		CSR-libfuzzer		CSR-AFL++	
	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time
arrow-40653	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
aspell-18462	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
curl-8000	1.0	0.15	1.0	0.06	1.0	0.05	0.7	0.55	**1.0**	**0.01**
exiv2–50315	0.0	1.00	0.2	0.92	0.3	0.84	0.0	1.00	**0.3**	**0.81**
file-30222	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
gdal-47716	0.5	0.60	1.0	0.07	1.0	0.17	**1.0**	**0.01**	**1.0**	**0.01**
grok-28418	0.0	1.00	0.3	0.85	**0.9**	**0.53**	0.0	1.00	0.5	0.72
harfbuzz-55779	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	**0.1**	**0.91**
leptonica-25212	0.0	1.00	0.0	1.00	0.2	0.87	0.1	0.93	**0.3**	**0.70**
libarchive-44843	0.0	1.00	0.0	1.00	0.0	1.00	–	–	–	–
libhtp-17198	0.7	0.36	**1.0**	**0.01**	0.9	0.14	1.0	0.02	1.0	0.03
ndpi-49057	0.0	1.00	0.2	0.85	0.0	1.00	0.0	1.00	**0.3**	**0.85**
openh264–26220	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
openssl-17715	0.9	0.31	0.8	0.39	**1.0**	**0.09**	1.0	0.29	1.0	0.30
pcap++-23592	0.0	1.00	**0.2**	**0.89**	0.0	1.00	0.0	1.00	0.0	1.00
poppler-35789	0.0	1.00	0.8	0.72	0.7	0.64	–	–	**1.0**	**0.07**
readstat-13262	0.0	1.00	0.7	0.77	**0.7**	**0.39**	–	–	–	–
usrsctp-18080	0.0	1.00	**0.1**	**0.95**	0.0	1.00	0.0	1.00	0.0	1.00
yara-38952	0.6	0.56	**1.0**	**0.22**	0.9	0.29	0.1	0.93	0.5	0.60
zstd-21970	1.0	0.13	**1.0**	**0.03**	1.0	0.05	0.9	0.20	1.0	0.06

Table 4
Fuzzing results without using bug-specific test oracles (RQ 2).

Name	libfuzzer		AFL++		AFLChurn		CSR-libfuzzer		CSR-AFL++	
	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time
exiv2-50315	1.0 (0.0)	0.00 (1.00)	0.4 (0.2)	0.60 (0.92)	1.0 (0.3)	0.04 (0.84)	1.0 (0.0)	0.00 (1.00)	0.6 (0.3)	0.53 (0.81)
grok-28418	1.0 (0.0)	0.10 (1.00)	1.0 (0.3)	0.05 (0.85)	1.0 (0.9)	0.03 (0.53)	1.0 (0.0)	0.24 (1.00)	1.0 (0.5)	0.06 (0.72)
leptonica-25212	1.0 (0.0)	0.00 (1.00)	0.5 (0.0)	0.62 (1.00)	0.3 (0.2)	0.73 (0.87)	1.0 (0.1)	0.00 (0.93)	0.4 (0.0)	0.00 (1.00)
ndpi-49057	0.0 (0.0)	1.00 (1.00)	0.2 (0.2)	0.85 (0.85)	1.0 (0.0)	0.00 (1.00)	0.0 (0.0)	1.00 (1.00)	0.3 (0.3)	0.85 (0.85)
pcap++-23592	1.0 (0.0)	0.00 (1.00)	1.0 (0.2)	0.00 (0.89)	1.0 (0.0)	0.00 (1.00)	1.0 (0.0)	0.00 (1.00)	1.0 (0.0)	0.04 (1.00)
readstat-13262	1.0 (0.0)	0.13 (1.00)	0.8 (0.7)	0.41 (0.77)	0.9 (0.7)	0.22 (0.39)	– (–)	– (–)	– (–)	– (–)
usrsctp-18080	0.1 (0.0)	0.99 (1.00)	0.5 (0.1)	0.67 (0.95)	0.7 (0.0)	0.63 (1.00)	0.3 (0.0)	0.90 (1.00)	0.8 (0.0)	0.29 (1.00)
Average	0.73 (0.00)	0.32 (1.00)	0.63 (0.24)	0.46 (0.89)	0.84 (0.30)	0.24 (0.80)	0.72 (0.02)	0.36 (0.99)	0.68 (0.18)	0.30 (0.90)

studies, we found that the conclusions do not change if we run fuzzers for 72 core-hours. In addition, to support that the fuzzing efforts of 6 hours with 6 cores is no less than fuzzing efforts of 36 hours with a single core, we additionally conducted the experiments for RQ 1 by running three fuzzers (AFL++, libFuzzers and AFLChurn) for 36 hours using a single core and compared the results. From this pilot study, we found that, with 6 cores for 6 hours, the studied fuzzers mostly show better fault detection performance, thus fuzzing effort of using 6 cores for 6 hours would be considered no less than fuzzing efforts of using single core for 36 hours.

The fuzzing results may be unintentionally interfered by uncontrolled aspects of the machine environments. To limit this threat, we faithfully modeled the OSS-Fuzz test environments, and confirmed that the failures are properly reproducing based on the OSS-Fuzz issue information.

5.2. Results

5.2.1. RQ 1. Failure detection effectiveness and efficiency

Table 3 describes the fault detection performance of five fuzzing configurations. The columns with 'ratio' present the ratio of the fuzzing runs that detect the failure to total number of fuzzing conductions (i.e., 10). The 'time' columns show the average ratio of first failure detection time to the time limit (i.e., 36 core-hours). A cell marked with '–' signifies that the respective experiment is not viable since none of the changed functions was covered by the bug-free version fuzzing, thus the seed corpus for the first buggy version fuzzing is undefined. The best results for each bug case is marked with the bold face.

In overall, Table 3 show that BugOss exhibits various combinations of fuzzing performance among the studied techniques. There are five bug cases that no fuzzing configuration succeeded to detect any failures. For other nine bug cases, no more than three fuzzing configurations detected a failure. There are six bug cases where all fuzzing configurations detect a failure, yet these show a high variety in failure detection time results. Among the other 15 bug cases, CSR-AFL++ performed best (i.e., with a highest fault detection ratio or a lowest fault detection time) at seven, AFL++ performed best at five, AFLChurn at three, and CSR-libfuzzer at one.

There are many cases where the regression fuzzing techniques (i.e., AFLChurn and CSR) are not effective and often they draw negative effects. The three configurations with regression fuzzing techniques (i.e., AFLChurn, CSR-libfuzzer, CSR-AFL++) show better performance than a general-purpose greybox fuzzer AFL++ only with six, three and nine bug cases, respectively. The two fuzzing configurations with CSR (CSR-libfuzzer and CSR-AFL++) performed worse than their counterparts without CSR (libfuzzer and AFL++) with three and five bug cases, respectively.

> Answer to RQ 1: BugOss provides real-world bug cases that exhibit both advantages and the remaining challenges of regression fuzzing techniques.

5.2.2. RQ 2. Impact of using bug-specific test oracle

Table 4 compares the fuzzing results without using the bug-specific test oracles (i.e., using the general test oracles) and the fuzzing results with using the bug-specific test oracles. Table 4 excludes the nine bug cases whose bug-specific test oracles are the same as the general test oracles (i.e., test oracle type is marked as 'all' in Table 2) and the five bug cases where no fuzzer detected any failures. For the other seven bug cases, Table 4 shows the failure detection ratio and the average

Table 5
Fuzzing results with bug-inducing commit information and with bug-fixing commit information (RQ 3).

Name	Changed functions			AFLChurn				CSR-libfuzzer				CSR-AFL++			
	BIC	BFC	BIC∩ BFC	With BIC		With BFC		With BIC		With BFC		With BIC		With BFC	
				Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time
arrow-40653	2	5	0	0.0	1.00	0.0	1.00	0.0	1.00	–	–	0.0	1.00	–	–
aspell-18462	1	6	0	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
curl-8000	2	1	1	1.0	0.05	1.0	0.04	0.7	0.55	–	–	**1.0**	**0.01**	–	–
exiv2–50315	3	1	1	0.3	0.84	0.4	0.80	0.0	1.00	0.0	1.00	0.3	0.81	**0.9**	**0.36**
file-30222	2	1	0	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
gdal-47716	1	2	1	1.0	0.17	1.0	0.23	**1.0**	**0.01**	**1.0**	**0.01**	**1.0**	**0.01**	1.0	0.02
grok-28418	10	10	4	0.9	0.53	0.9	0.53	0.0	1.00	0.0	1.00	0.5	0.72	**1.0**	**0.27**
harfbuzz-55779	3	2	2	0.0	1.00	0.0	1.00	0.0	1.00	–	–	**0.1**	**0.91**	–	–
leptonica-25212	1	3	1	0.2	0.87	0.1	0.97	0.1	0.93	**0.7**	**0.48**	0.3	0.70	0.3	0.70
libarchive-44843	3	1	1	0.0	1.00	0.0	1.00	–	–	–	–	–	–	–	–
libhtp-17198	2	5	2	0.9	0.14	1.0	0.03	**1.0**	**0.02**	1.0	0.08	1.0	0.03	1.0	0.04
ndpi-49057	2	4	1	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	**0.3**	**0.85**	0.1	0.97
openh264–26220	1	1	1	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
openssl-17715	2	2	2	1.0	0.09	**1.0**	**0.01**	1.0	0.29	1.0	0.29	1.0	0.30	1.0	0.30
pcap++-23592	6	2	1	0.0	1.00	0.0	1.00	0.0	1.00	–	–	0.0	1.00	–	–
poppler-35789	2	2	1	0.7	0.64	0.4	0.90	–	–	–	–	1.0	0.07	**1.0**	**0.00**
readstat-13262	1	2	0	**0.7**	**0.39**	0.0	1.00	–	–	0.0	1.00	–	–	0.0	1.00
usrsctp-18080	1	1	0	0.0	1.00	**0.3**	**0.82**	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
yara-38952	1	–	0	0.9	0.29	**1.0**	**0.21**	0.1	0.93	–	–	0.5	0.60	–	–
zstd-21970	5	12	0	1.0	0.05	1.0	0.06	0.9	0.20	**1.0**	**0.03**	1.0	0.06	**1.0**	**0.03**

time of the first failure detection measured with the general test oracles on the "ratio" columns and the "time" columns, respectively. The numbers in the parenthesis are the corresponding results measured with the bug-specific test oracles (i.e., the results at Table 3). The last row presents the average over the seven bug case results for each column.

The failure detection results are significantly improved for all fuzzing techniques and bug cases except four cases (i.e., libfuzzer, AFL++, CSR-libfuzzer, and CSR-AFL++ for ndpi-49057). For example, libfuzzer achieves on average 73% failure detection, while it fails to detect any failure by the bug-specific test oracle. For example of AFLChurn, the failure detection ratio is increased by 180% on average, and the first time to failure detection is decreased by 70% on average when the general test oracle is used.

These results imply that the use of bug-specific test oracles significantly influences the fuzzing performance if a target program has multiple bugs. If other bugs may exist with a target bug, the bug-specific test oracle must be carefully constructed and applied for conducting fuzzing experiments. Otherwise, the experiment results may be easily invalid since the failure detection results are likely dominated by another bug for which fuzzers quickly generate failing inputs.

> Answer to RQ 2: The failure detection results change significantly depending on the use of the bug-specific test oracles when the target programs contain one or more co-existing bugs.

5.2.3. RQ 3. Impact of using bug-fixing change information

Table 5 compared the results of the regression fuzzing techniques with the changed functions defined by the bug-inducing commit (the "with BIC" columns) and the results by the bug-fixing commit (the "with BFC" columns). A cell marked with '–' signifies, as Table 3 does, that the respective experiment is not viable since none of the changed functions was covered by the bug-free version fuzzing. The best results for each bug case is marked with the bold face. In addition, Table 5 describes the number of the functions changed by the bug-inducing commit (the "BIC" column), the number of the functions changed by the bug-fixing commit (the "BFC" column), and their intersections (the "BIC∩BFC" column). For yara-38952, the functions changed by the BFC is marked as undefined ('–') because the code change is on the macro code which impacts on many functions over the projects. Among the 20 bug cases, two share identical changed functions for BIC and BFC, while seven have no overlap between the changed functions for BIC and BFC.

The result shows that, for six bug cases, all fuzzing configuration studied for RQ 3 failed in generating a bug-revealing input (arrow-40653, aspell-18462, file-30222, libarchive-44843, openh-26220 and pcap++-23592). With all the other 14 bug cases, the results show that, in at least one studied fuzzing configuration, the fuzzing performance changes depending on the given changed function information with 14 bug cases. When the BFC information is used, AFLChurn exhibits better fuzzing results (i.e., fault detection ratio increases or the time to first fault detection decreases by at least 5%p) with six bug cases (openssl-17715, exiv2-50315, gdal-47716, libhtp-17198, usrsctp-18080 and yara-38952), while it performs worse (i.e., fault detection ratio is reduced or the time to first fault detection increases by at least 5%p) with three bug cases (leptonica-25212, poppler-35789 and readstat-13262).

CSR-libfuzzer or CSR-AFL++ performs better with five bug cases when the changed functions are defined with the BFC information (exiv2-50315, leptonica-25212, grok-28418, poppler-35789 and zstd-21970). On the other hands, the CSR technique was not applicable to seven bug cases since none of the target functions were covered in the bug-free versing fuzzing phases. In addition, CSR-AFL++ exhibits worse performance with two bug cases (gdal-47716 and ndpi-49057).

One finding is that the BugOss bug cases demonstrates that, depending on whether the changed code is defined by BIC or BFC, the performance of the studied regression fuzzing techniques varies. This finding implies that the artificial bug cases with bug-fixing information may lead to inaccuracies in evaluating regression fuzzing techniques.

Another finding is that five bug cases in BugOss reveals counter-intuitive results, indicating that the performance of a regression fuzzing technique is rather reduced when the changed code information accurately points bug locations (leptonica-25212, poppler-35789 and readstat-13262 for AFLChurn; curl-8000, libhtp-17198 and yara-38952 for CSR). We anticipate that these bug cases indicate the limitations of the studied regression fuzzing techniques in directing fuzzing toward the target code locations.

Table 6
Fuzzing results on last buggy versions with bug-fixing commit information (RQ 4).

Name	AFLChurn				CSR-libfuzzer				CSR-AFL++			
	First buggy version		Last buggy version		First buggy version		Last buggy version		First buggy version		Last buggy version	
	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time
arrow-40653	0.0	1.00	0.0	1.00	0.0	1.00	–	–	0.0	1.00	–	–
aspell-18462	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
curl-8000	1.0	0.05	1.0	0.04	0.7	0.55	–	–	**1.0**	**0.01**	–	–
exiv2–50315	0.3	0.84	0.4	0.80	0.0	1.00	0.0	1.00	0.3	0.81	**0.6**	**0.76**
file-30222	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
gdal-47716	1.0	0.17	1.0	0.23	**1.0**	**0.01**	**1.0**	**0.01**	**1.0**	**0.01**	1.0	0.08
grok-28418	**0.9**	**0.53**	**0.9**	**0.53**	0.0	1.00	0.0	1.00	0.5	0.72	0.0	1.00
harfbuzz-55779	0.0	1.00	0.0	1.00	0.0	1.00	–	–	**0.1**	**0.91**	–	–
leptonica-25212	0.2	0.87	0.1	0.97	0.1	0.93	**0.6**	**0.58**	0.3	0.70	0.3	0.70
libarchive-44843	0.0	1.00	0.0	1.00	–	–	–	–	–	–	–	–
libhtp-17198	0.9	0.14	1.0	0.03	**1.0**	**0.02**	1.0	0.04	1.0	0.03	0.9	0.34
ndpi-49057	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.3	0.85	**0.4**	**0.71**
openh264–26220	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
openssl-17715	1.0	0.09	**1.0**	**0.01**	1.0	0.29	1.0	0.12	1.0	0.30	0.9	0.44
pcap++-23592	0.0	1.00	0.0	1.00	0.0	1.00	–	–	0.0	1.00	–	–
poppler-35789	0.7	0.64	0.4	0.90	–	–	–	–	1.0	0.07	**1.0**	**0.01**
readstat-13262	**0.7**	**0.39**	0.0	1.00	–	–	0.0	1.00	–	–	0.0	1.00
usrsctp-18080	0.0	1.00	**0.3**	**0.82**	0.0	1.00	0.0	1.00	0.0	1.00	0.0	1.00
yara-38952	0.9	0.29	**1.0**	**0.21**	0.1	0.93	–	–	0.5	0.60	–	–
zstd-21970	1.0	0.05	1.0	0.06	0.9	0.20	**1.0**	**0.03**	1.0	0.06	1.0	0.04

Answer to RQ 3: BugOss exhibits that the bug-inducing commits and the bug-fixing commits often indicates changed code locations differently, and the performance of the regression fuzzing techniques largely depend on given changed code information.

5.2.4. RQ 4. Impact of using last buggy version

In addition to RQ 3, RQ 4 explores how the studied fuzzing techniques perform differently when they test the last buggy version (i.e., the immediately preceding version of the bug-fixing commit) instead of the first buggy version. Table 6 compares the results with the last buggy version and the first buggy version for each studied fuzzing configuration. Note that the results with the first buggy versions are also presented in Table 3. A cell marked with '–' in this table indicates that the respective experiment is not viable since none of the changed functions was covered by the bug-free version fuzzing. The best results for each bug case is marked with the bold face.

Overall, with all 14 bug cases where one of the regression fuzzing techniques succeeded in generating a bug-revealing input, at least one of the studied fuzzing configurations exhibits varying performances (i.e., fault detection ratio is different or the difference in the time to first fault detection exceeds 5%p). For AFLChurn, the fuzzing performance are changed in eleven bug cases. For example of poppler-35789, the difference in the fault detection ratio is 0.3. For CSR-libfuzzer and CSR-AFL++, the performance change is observed with 12 bug cases. The most significant change is CSR-AFL++ with grok-28418 where the difference in fault detection ratio is 0.5.

Similar to the findings in RQ 3, the results for RQ 4 also show that the performance of the studied regression fuzzing techniques varies depending on whether fuzzing is applied to actual bug-inducing commits (i.e., first buggy version) or the bug-fixing commits (i.e., last buggy version) are utilized.

Answer to RQ 4: BugOss demonstrates that the performance of the regression fuzzing techniques change depending on whether fuzzing is applied to actual bug-inducing commits or the bug-fixing commits are utilized.

6. Discussion

6.1. Usefulness of BugOss bug cases

The results (Section 5) illustrate that the bug cases of BugOss uncovers the limitation of the studied fuzzing techniques. Specifically, all studied fuzzing configurations were unsuccessful in generating any bug-revealing inputs for five bug cases. For other five bug cases, a general-purpose fuzzer AFL++ outperforms regression fuzzing techniques, indicating that regression fuzzing techniques often have adverse effects on fuzzing performance. We envision that BugOss can serve as a comprehensive benchmark for identifying the research problems to facilitate improvements in regression fuzzing techniques and guide future research efforts in addressing the identified challenges.

To understand the current challenges, we further analyzed the cases of the five bugs (arrow-40653, aspell-18462, file-30222, openh264-26220 and libarchive-44843) for which all studied fuzzers failed to generate bug-revealing inputs. First, we checked whether the buggy code is executed. We found that, for one bug case (libarchive-44843), all fuzzers failed to execute any bug location within the given fuzzing time. Subsequently, we determined the infection condition by comparing the buggy code and the bug-fixing commit and inspecting the failing execution with the bug-revealing inputs, and checked whether the infection occurs when the bug locations are executed. As result, we concluded that, for two bug cases (file-30222 and openh264-26220), all fuzzing techniques could not induce infections. Meanwhile, for the other two bug cases (arrow-40653 and aspell-18462), we found that the infection conditions are once satisfied by generated inputs, yet the propagations never happened.

Through these analyses, we found that BugOss encompasses interesting bug cases to examine the fault detection ability of a fuzzing techniques. In addition, we confirmed that the bug case artifacts offer useful data for detailed analysis of the bug cases.

6.2. Usefulness of inferred test oracles

It is common that a bug-inducing commit adds a new bug to the target program where one or more other bugs already exist. Even when OSS-Fuzz identifies failures and reports concrete bug-revealing inputs, it frequently requires a decent amount of time for maintainers to fix the bugs. As a result, continuous fuzzing is often carried out in the presence of multiple bugs. BugOss represents such situations as 11 out

of the 20 bug cases are sourced from the bug-inducing commits with other co-existing bugs (see Table 2).

The study of RQ 2 (see Section 5.2.2) suggests that the evaluation of regression fuzzing techniques may imply invalid results if the effect of other co-existing bugs are not carefully accounted. To limit this threat in a cost-effective way, each bug case of BugOss provides the bug-specific test oracles together with the test oracles of the other failures (find more details at Section 3.3). The experiment results of RQ 2 demonstrates that the performance of a fuzzing technique varies significantly depending on whether or not these test oracles are used for determining if a failure reveals the target bug.

Since these are extracted from only limited failures cases, the test oracles in BugOss may fail to classify a failure correctly if the failure exhibits unidentified symptoms. To assess whether this limitation matters in our experiments, we studied all cases of unidentified failures (i.e., a failure accepted by none of the test oracles of the bug case). We found that total 5 occurrences of unidentified failures in whole experiments presented in Section 5. By running the last buggy version and the fixed buggy version with these failure-inducing inputs against, we confirmed that these unidentified failures are not caused by the target bugs. Furthermore, we found that, four out of the five unidentified failures are reproducible at the latest versions of the target projects at the moment of the experiments. Based on these results, we believe that the process outlined in Section 3 yields effective test oracles capable of identifying the failures by a specific bug of interest at fuzzing in the presence of co-existing bugs.

In addition, from the experiment results, we checked if all first failures found the fuzzers and accepted by the bug-specific test oracle are the actual bug-revealing inputs by running them against the different versions. As a result, we found that all of them are true positives (i.e., pass on the bug-free and fixed versions, fail on the first and last buggy versions). For these reasons, while the inferred test oracles may not be perfectly accurate, we believe that our approach is practically meaningful.

6.3. Challenges in benchmark construction

To obtain the 20 bug cases of BugOss, we had reviewed total 2074 OSS-Fuzz issues from 65 open-source projects. We rejected 2054 issues in the middle of the construction process (Section 3.2), mostly because we could not find clear evidence for supporting bug-fixing commits (Step 2). We intended BugOss to provide a full package of details for experimenting regression fuzzing techniques and studying their shortcomings by quantitative analyses on the bug cases. For this reason, we exclude a bug case if it is unclear which commit is intended to fix the bug under consideration. Although it seems quite conservative, we expect that this criteria prevents possible errors in BIC and BFC information (see Section 5.1.3).

Although OSS-Fuzz publicly open a large number of the bug reports that reveal real-world bugs in open-source projects, due to the lack of the traceability (Kagdi et al., 2007) between the OSS-Fuzz issues and the target project commit history, we found that only a small portion of these can be contributed to construct BugOss bug cases. It would be beneficial if there is an automatic way to accurately associate the OSS-Fuzz issues with the corresponding bug-inducing commits and bug-fixing commits based on the evidences in project context such as commit messages.

6.4. Using AFLChurn and change-aware seed reuse together

We constructed another fuzzing configuration that leverages both regression fuzzing techniques (AFLChurn and change-aware seed reuse) at the same time to explore a possible way to improve regression fuzzing. Table 7 shows the results of CSR-AFLChurn where AFLChurn runs on the first buggy versions where the initial seeds were generated on the bug-free versions and selected based on the code changes at the bug-inducing commits. The results of CSR-AFLChurn are compared with the results of AFLChurn for studying RQ 1 (Table 3).

Table 7
Fuzzing results of AFLChurn with Change-aware seed reuse.

Name	AFLChurn		CSR-AFLChurn	
	Ratio	Time	Ratio	Time
arrow-40653	0.0	1.00	0.0	1.00
aspell-18462	0.0	1.00	0.0	1.00
curl-8000	1.0	0.05	**1.0**	**0.00**
exiv2–50315	**0.3**	**0.84**	0.2	0.87
file-30222	0.0	1.00	0.0	1.00
gdal-47716	1.0	0.17	**1.0**	**0.01**
grok-28418	**0.9**	**0.53**	0.8	0.52
harfbuzz-55779	0.0	1.00	**0.1**	**0.94**
leptonica-25212	0.2	0.87	**0.3**	**0.70**
libarchive-44843	0.0	1.00	–	–
libhtp-17198	0.9	0.14	**1.0**	**0.04**
ndpi-49057	0.0	1.00	**0.1**	**0.93**
openh264–26220	0.0	1.00	0.0	1.00
openssl-17715	1.0	0.09	**1.0**	**0.01**
pcap++-23592	0.0	1.00	0.0	1.00
poppler-35789	0.7	0.64	**1.0**	**0.07**
readstat-13262	**0.7**	**0.39**	–	–
usrsctp-18080	0.0	1.00	0.0	1.00
yara-38952	**0.9**	**0.29**	0.7	0.48
zstd-21970	**1.0**	**0.05**	1.0	0.08

The combined use of the two techniques improves the fault detection results with eight bug cases compared to AFLChurn. Meanwhile, CSR-AFLChurn exhibits slightly reduced performances with five bug cases. When AFL++ and CSR-AFL++ are taken into account, CSR-AFLChurn outperformed only with openssl-17715. With the seven bug cases where no fuzzing configurations were effective, CSR-AFLChurn is also unsuccessful in generating a bug-revealing input. These findings suggest that further investigations are required to explore effective methods for integrating different regression fuzzing techniques to improve fault detection performance.

7. Conclusion

This paper presents a regression bug benchmark BugOss for evaluating regression fuzzing techniques for C/C++ programs with realistic continuous fuzzing situations. To mitigate possible threats in empirical evaluation, BugOss pinpoints bug-inducing commits of target bugs, and provide specific test oracles to discriminate the failures by the target bugs from failures by other co-existing bugs. We detailed the systematic procedures for constructing bug cases based on OSS-Fuzz issues. The empirical investigation with BugOss discloses that the project context information affects the regression fuzzing performance, and the 20 bug cases currently registered for BugOss covers various cases of regression bugs in real-world. The BugOss benchmark would be useful for researchers to evaluate regression fuzzing techniques and understand the remaining challenges.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Hanyoung Yoo, Suhyun Park, Sungbin Lim, and Kieun Kim for their efforts in reviewing the experiment results.

Funding

This work was supported by the National Research Foundation of Korea (grant numbers 2020R1C1C1013512 and 2021R1A5A1021944) and funded by the Korean government. In addition, this work was partly supported by Sabbatical Leave Grant at Handong Global University when the authors were affiliated with Handong Global University.

References

An, G., Hong, J., Kim, N., Yoo, S., 2023. Fonte: Finding bug inducing commits from failures. In: Proceedings of the International Conference on Software Engineering (ICSE).

An, G., Yoon, J., Yoo, S., 2021. Searching for multi-fault programs in Defects4J. In: LNCS, volume 12914: Search-Based Software Engineering (SSBSE).

Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A., 2017. Directed greybox fuzzing. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).

Böhme, M., Szekeres, L., Metzman, J., 2022. On the reliability of coverage-based fuzzer benchmarking. In: Proceedings of the International Conference on Software Engineering (ICSE).

Bundt, J., Fasano, A., Dolan-Gavitt, B., Robertson, W., Leek, T., 2021. Evaluating synthetic bugs. In: Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS).

2017. DARPA cyber grand challenge repositories. URL https://github.com/CyberGrandChallenge.

Ding, Z.Y., Le Goues, C., 2021. An empirical study of OSS-Fuzz bugs. In: Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. LAVA: Large-scale automated vulnerability addition. In: Proceedings of the IEEE Symposium on Security and Privacy (SP).

Fasano, A., Leek, T., Dolan-Gavitt, B., Bundt, J., 2019. The Rode0day to less-buggy programs. IEEE Security & Privacy 17 (6).

Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M., 2020. AFL++: Combining incremental steps of fuzzing research. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT).

2016. Google fuzzer-test-suite (FTS) benchmark. URL https://github.com/google/fuzzer-test-suite.

Hazimeh, A., Herrera, A., Payer, M., 2021. Magma: A ground-truth fuzzing benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems 4 (3).

Kagdi, H., Maletic, J.I., Sharif, B., 2007. Mining software repositories for traceability links. In: Proceedings of the IEEE International Conference on Program Comprehension (ICPC).

Keller, B.N., Meyers, B.S., Meneely, A., 2023. What happens when we fuzz? Investigating OSS-fuzz bug history. In: Proceedings ofthe IEEE/ACM 20th International Conference on Mining Software Repositories (MSR).

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).

Klooster, T., Turkmen, F., Broenink, G., Hove, R.t., Böhme, M., 2023. Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in CI/CD pipelines. In: Proceedings of the IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT).

Lee, H., Kim, S., Cha, S.K., 2023. Fuzzle: Making a puzzle for fuzzers. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).

Lemieux, C., Padhye, R., Sen, K., Song, D., 2018. PerfFuzz: Automatically generating pathological inputs. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).

Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., Lu, K., Wang, T., 2021. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In: Proceedings of the USENIX Security Symposium.

Metzman, J., Szekeres, L., Simon, L.M.R., Sprabery, R.T., Arya, A., 2021. FuzzBench: An open fuzzer benchmarking platform and service. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).

Serebryany, K., 2016. Continuous fuzzing with libFuzzer and AddressSanitizer. In: Proceedings of the IEEE Cybersecurity Development Conference (SecDev). http://dx.doi.org/10.1109/SecDev.2016.043.

Serebryany, K., 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In: Proceedings of the USENIX Security Symposium.

Siliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: Proceedings of the International Workshop on Mining Software Repositories (MSR).

Wen, M., Wu, R., Cheung, S.-C., 2016. Locus: Locating bugs from software changes. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).

Yoo, H., Hong, J., Bader, L., Hwang, D.W., Hong, S., 2021. Improving configurability of unit-level continuous fuzzing: An industrial case study with SAP HANA. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).

Zhang, Z., Patterson, Z., Hicks, M., Wei, S., 2022. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In: Proceedings of the USENIX Security Symposium.

Zhu, X., Böhme, M., 2021. Regression greybox fuzzing. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).