Poster: BugOss: A Regression Bug Benchmark for Empirical Study of Regression Fuzzing Techniques

Jeewoong Kim, Shin Hong
jeewoong@handong.ac.kr, hongshin@handong.edu
Handong Global University

***Abstract*—This paper presents BugOss, a benchmark of real-world regression bugs found in OSS-Fuzz for experimenting with regression fuzzing techniques. To reproduce the real project context where the bugs were introduced, each study artifact of BugOss indicates the exact bug-inducing commit, and provides the information about the target bug, together with the existing bugs in the same commit. The experiment results with five fuzzing techniques show that the 18 C/C++ artifacts currently registered for BugOss encompass various cases of regression bugs in real-world. We believe that BugOss offers a useful basis for empirically investigating regression fuzzing techniques.**

I. Introduction

Continuous fuzzing is running fuzzers periodically to generate new test inputs to guard the project against regression bugs along a series of program changes [1]. OSS-Fuzz [2] has demonstrated the effectiveness of continuous fuzzing by detecting more than 28000 bugs from 800 and more open-source projects. 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 [3]–[5].

Regression fuzzing techniques utilize the information on program changes to quickly discover recently introduced failures. Zhu and Böhme [3] 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.* [4] 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 can quickly discover regression bugs, and save computation effort from re-testing unchanged behaviors.

Regression fuzzing is a promising direction for enhancing continuous fuzzing efficacy, and more investigations are expected to follow to improve regression fuzzing techniques. However, the existing fuzzing benchmarks [3], [6], [7] are not suitable for evaluating regression fuzzing techniques because their study artifacts 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 II).

Currently, 18 artifacts from 18 C/C++ programs are registered for the BugOss benchmark. To understand their characteristics, we conducted experiments with two general-purpose fuzzers and two regression fuzzing techniques (Section IV). From the experiment results, we found that the 18 artifacts encompass various cases of real-world regression bugs. To the best of the this authors' knowledge, BugOss is the first benchmark with actual bug-inducing commits of real-world projects under continuous fuzzing for studying 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/arise-handong/BugOss`

II. Study Artifact Construction

A. Artifact structure

Each study artifact is purposed to study how the fuzzer performs to generate a bug-revealing input (i.e., an input that triggers a target bug to induces a failure) when a real bug is newly added to a project under continuous fuzzing. To capture bug-inducing situations in real-world, we first gathered OSS-Fuzz issues by which the developers recognized and fixed unknown bugs in open-source projects. For each gathered issue, we systematically reviewed the target project and the related OSS-Fuzz issues to identify the bug-inducing commit and collected various failure samples. Specifically, a study artifact in BugOss consists of the following attributes:

- **bug-revealing input**: an input for a fuzzing target, that induces a failure. This input is given as the failure-reproducing input in the OSS-Fuzz issue.
- **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 *clean* version and the *faulty* version, respectively.
- **fix-inducing commit**: the program change that repairs the target bug. We call the program versions after the fix-inducing commit as the *fixed* version, respectively.

- **bug locations**: a subset of the changed lines in the bug-inducing commit, that are suspected to result a failure when the bug-revealing input is given.
- **failure symptom**: a condition to determine whether a failure is induced by the target bug, or it is induced by the other bugs. A failure symptom is described in natural language, and encoded as an executable form (e.g., sanitizer setting, script).

B. 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-Fuzze issue gives clues about when the target bug was induced and fixed (e.g., bijection, disclosure), manual inspection is required to pinpoint exact information about the target bug, and reject unsuitable cases. We systematically conducted this process by taking the following steps:

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. After that, 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 that 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. **Finding fix-inducing 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 or the related documentation mentions OSS-Fuzz or the particular failure symptom explicitly, we confirm that the commit is the fix-inducing 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 later steps require the fixed version to classify failure symptoms.
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. Once such version is located, we manually checked if the corresponding commit is likely to induce the target bug.

   The located commit was confirmed as the bug-inducing commit if we found that the suspected commit and the fix-inducing commit (Step 2) concern the same code lines or the same data structure. Otherwise, we excluded the issues from the benchmark construction to avoid possible uncertainty. We also rejected the issues 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 from OSS-Fuzz. In addition, we determined the bug location by comparing the changed lines of the bug-inducing commit and the fix-inducing commit.
4. **Collecting failure samples** Since a program version is likely to contain multiple bugs [8], each study artifact of BugOss needs to provide a discriminating condition (calling it "fault symptom") between failures by the target bug and failures by other bugs.

   To build the references for writing a fault symptom, we gathered the two kinds of failure samples. First, we ran the general-purpose fuzzers with the clean version for 48 hours because there may be pre-existing bugs before the bug-inducing commit. If a failure is observed with the clean version, we considered all cases as the failures by the other faults. Second, we gathered failure-reproducing inputs of the OSS-Fuzz issues registered between the bug-inducing commit and the fix-inducing commit. For each gathered failure-reproducing input, we identified the survival range in the version history that the input actually results the failure. If the range of a failure-reproducing input is the identical to that of the bug-revealing input, we considered that the failure-reproducing 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 faults.

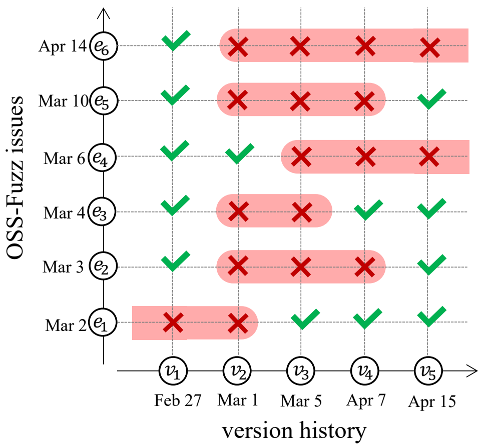

Fig. 1: Failure sampling

Figure 1 describes four different cases of failure samples. 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 study artifact with bug-revealing input e_5 whose the 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 concluded that e_3, e_4, and e_6 are resulted by

TABLE I: Study Artifacts

name	failure type	seeds	changed lines		B→F	#failures	
			BIC	Fix		T	F
aspell-18462	buf-overrun	2	5	18	87	1	1
curl-8000	buf-overrun	4202	51	2	1	1	0
exiv2-50315	int-overflow	454	45	3	18	1	13
file-30222	null-deref	34	21	11	0	1	7
gdal-47716	buf-overrun	1615	10	4	8	1	0
grok-28418	mem-leak	178	101	55	3	1	36
leptonica-25212	null-deref	10	25	26	5	1	1
libhtp-17198	buf-overrun	97	26	29	3	1	0
libxml2-17737	use-aft-free	1274	32	8	2	1	94
ndpi-49057	int-overflow	351	51	10	5	1	16
openh-26220	buf-overrun	174	7	5	19	10	0
openssl-17715	buf-overrun	2240	91	47	267	2	0
pcap++-23592	buf-overrun	615	32	13	130	1	3
poppler-35789	null-deref	476	3	20	4	1	0
readstat-13262	buf-overrun	94	5	10	44	1	5
usrsctp-18080	use-aft-free	156	6	8	9	1	3
yara-38952	buf-overrun	9	277	17	0	1	0
zstd-21970	null-deref	12462	280	247	53	1	0

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-exiting fault. We can collect such failures by fuzzing the clean version.

5. **Defining failure symptom.** Given two kinds of failure samples of a study artifact, we manually wrote a failure symptom, a condition that discriminates the failures by the target bug and the failures by the other bugs. A failure symptom is described as specific failure types, or patterns of stack traces, or crash messages. We encoded the failure symptom as an executable form (e.g., sanitizer setting, script) to be used in fuzzing process.

   If the failures by the target bug has the identical failure message and stack trace to certain failures by other bugs, we tried to resolve them by adding logging messages to notify that certain branches related to the bugs are covered. The failure symptoms of these artifacts check these logging messages in addition. Meanwhile, if we could not add logging for the differentiation, we rejected the cases to limit the threats by weak test oracles.
6. **Rejecting useless cases.** Lastly, we rejected the case if, for the faulty version, two conventional fuzzers, AFL++ and libfuzzer, always generate a bug-revealing input within 3 minutes. We believe that such cases may not be useful for comparing the performance of fuzzers. Also, we rejected the cases if the bug-inducing commit changes more than 300 lines of code, because such a large code modification is not a proper subject of regression fuzzing (i.e., may involve major functionality changes).

III. BugOss Benchmark

We have registered 18 study artifacts for BugOss, as shown in Table I. These artifacts are originated 18 real bugs detected by OSS-Fuzz from 18 well-known open-source projects. To these artifacts, we had reviewed total 819 OSS-Fuzz issues from 42 target projects, and rejected 801 issues in the middle of the construction process (Section II-B). Most of the rejection were made because we could not find explicit evidence for supporting fix-inducing commits (Step 2).

In Table I, the fist column names each artifact by combining the project name and the OSS-Fuzz issue number. The second column describes the failure type. The third column shows the number of seed inputs at the bug-inducing commit. The columns with 'changed line' shows the number of changed lines in the bug-inducing commit ('BIC') and the fix-inducing commit ('Fix'). The 'B→F' represents the number of commits between the bug-inducing commit and the fix-inducing commit. The columns with '#failures' give the number of the failures by the target bug ('T'), and the number of the failures by the other bugs ('F') collected for defining the failure symptom. For readstat-13262 and pcap++-23592, we added logging messages to define the failure symptoms.

Table I shows that BugOss covers various cases of regression bugs. The number of lines changed by bug-inducing commit is small (less than 10) for five, moderate (10 to 50) for other seven, and large (more than 50) for the other six artifacts. The numbers in the 'B→F' column estimate the time-to-fix aspects [9]. Among the 18 artifacts, 11 have less than short time-to-fix durations (less than 10 commits), while the other seven have moderate to long time-to-fix durations. The numbers of the failure samples implies that 8 artifacts are of the single-bug case, while the other 10 artifacts involve multiple bugs in their bug-inducing commits.

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

IV. Experiments with Fuzzers

We conducted experiments with the following five fuzzing techniques to demonstrate that BugOss can be used for comparing different techniques for continuous fuzzing:

- **libfuzzer** [1] of llvm-14.0.0 as a general-purpose fuzzer
- **AFL++** version 4.05c [10] as a general-purpose fuzzer
- **AFLChurn** [3] 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 [4] upon llvm-14.0.0. We configured libfuzzer to run fuzzing with the clean version for six hours and then re-use all generated inputs that cover a changed function for fuzzing the faulty version.

TABLE II: Fuzzing Results

	libfuzzer		AFL++		AFLChurn		CSR-libfuzzer		CSR-AFL++	
Name	ratio (%)	time (%)	ratio (%)	time (%)	ratio (%)	time (%)	ratio (%)	time (%)	ratio (%)	time (%)
aspell-18462	0	100	0	100	0	100	0	100	0	100
curl-8000	100	27	100	6	100	3	70	56	100	13
exiv2-50315	0	100	20	91	30	84	0	100	10	90
file-30222	0	100	0	100	0	100	0	100	0	100
gdal-47716	50	60	100	7	100	17	100	1	100	1
grok-28418	20	94	30	85	90	53	0	100	80	54
leptonica-25212	0	100	0	100	20	87	10	93	0	100
libhtp-17198	70	50	100	2	90	19	100	3	100	0
libxml2-17737	60	48	100	1	0	100	60	47	100	6
ndpi-49057	0	100	20	85	0	100	0	100	40	73
openh-26220	0	100	0	100	0	100	0	100	0	100
openssl-17715	80	43	70	60	100	1	100	19	100	32
pcap++-23592	0	100	20	89	0	100	0	100	10	94
poppler-35789	0	100	90	55	70	64	0	100	0	100
readstat-13262	0	100	70	77	70	39	0	100	0	100
usrsctp-18080	0	100	10	95	0	100	0	100	0	100
yara-38952	60	61	100	22	90	29	30	87	60	49
zstd-21970	100	6	100	2	100	5	100	8	100	5

- **CSR-AFL++** as an in-house implementation of the change-aware seed reuse technique [4] upon AFL++ version 4.05c [10]. For fuzzing faulty versions, CSR-AFL++ re-uses the inputs generated for the clean version with libfuzzer.

For each fuzzer and artifact, we conducted 6-hours fuzzing runs for 10 times. Table II shows the experiment results. The columns with 'ratio' presents the ratio of the fuzzing campaigns that detect the failure to total number of fuzzing conductions (i.e., 10). The 'time' columns show the ratio of average time to first failure detection to the time limit (i.e., 6 hours). For computing the average time, we count 6 hours for a fuzzing campaign if the expected failure was not found.

Table II show that BugOss includes various combinations of fuzzing results. There are three artifacts that no fuzzers succeeded to detect the failures. Among the other 15, AFL++ performs best at six artifacts, AFLChurn at six, CSR-AFL++ at three and CSR-libfuzzer at one artifact.

According to Table II, in many cases, the three studied regression fuzzing techniques are not effective, and often they draw negative effects. AFLChurn, CSR-libfuzzer, and CSR-AFL++ show better performance than a general-purpose greybox fuzzer AFL++ only for six, three, and five artifacts, respectively. Comparing with libfuzzer and AFL++, CSR-libfuzzer and CSR-AFL++ perform worse with three and eight artifacts, respectively. This result indicates BugOss discloses the remaining challenges of regression fuzzing techniques with real-world bug cases.

V. 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 information on possible failures by target bugs and other co-existing bugs. We detailed the procedures for constructing study artifacts based on OSS-Fuzz issues. The 18 artifacts currently registered for BugOss cover 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.

As on-going works, we are creating more artifacts, especially to cover more failure types. To better understand the current challenges of regression fuzzing, we are studying more fuzzers with BugOss in the context of continuous fuzzing. Automating benchmark construction would be an interesting future work. We will investigate a technique to associate OSS-Fuzz issues with exact fix-inducing commits based the evidences in project context such as commit messages.

Acknowledgement

This work is supported by the National Research Foundation of Korea grants funded by the Korea government (2020R1C1C1013512 and 2021R1A5A1021944).

References

[1] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *IEEE Cybersecurity Development*, 2016, pp. 157–157.

[2] ——, "OSS-Fuzz - Google's continuous fuzzing service for open source software," in *USENIX Security*. Vancouver, BC: USENIX Association, Aug. 2017.

[3] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[4] H. Yoo, J. Hong, L. Bader, D. W. Hwang, and S. Hong, "Improving configurability of unit-level continuous fuzzing: An industrial case study with SAP HANA," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[5] T. Klooster, F. Turkmen, G. Broenink, R. t. Hove, and M. Böhme, "Effectiveness and scalability of fuzzing techniques in CI/CD pipelines," 2022. [Online]. Available: https://arxiv.org/abs/2205.14964

[6] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 4, no. 3, jun 2021.

[7] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy*, 2016.

[8] G. An, J. Yoon, and S. Yoo, "Searching for multi-fault programs in defects4j," in *Proceedings of the 13th International Symposium on Search Based Software Engineering*, 2021, pp. 136–150.

[9] Z. Y. Ding and C. Le Goues, "An empirical study of OSS-Fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 131–142.

[10] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.