Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup (Registered Report)

Madonna Huang
University of British Columbia
Vancouver, Canada
huicongh@cs.ubc.ca

Caroline Lemieux
University of British Columbia
Vancouver, Canada
clemieux@cs.ubc.ca

Abstract

Fuzzing best practices suggest that fuzzing should be run for at least 24 hours, if not longer. This recommendation makes it hard to integrate fuzzing into CI/CD contexts, to rapidly check a commit for bugs. Existing studies on CI/CD fuzzing simulated a CI/CD environment by running undirected fuzzers on Magma benchmark programs, which have multiple bugs injected into a single version of the program. Directed fuzzers, such as AFLGo, aim to generate inputs that reach specific target locations in the program being fuzzed. Thus, they should be more effective at fuzzing in a CI/CD environment. In this study, we propose to evaluate both directed and undirected fuzzers in a simulated CI/CD environment. Like prior work, we will use Magma as a source of benchmarks, and run fuzzers for 10 minutes. Unlike prior work, we will start the fuzzing process from a saturated corpus, rather than Magma's default corpus. Also unlike prior work, we will run the fuzzers on versions of Magma programs with a single bug injected. To deal with the threat that Magma patches give directed fuzzers access to too precise information as to the bug location, we will also conduct experiments where we add additional lines of target code, to evaluate the sensitivity of directed fuzzers. Our registered report gives preliminary results on a small subset of benchmarks.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

continuous integration, fuzz testing, directed fuzz testing

ACM Reference Format:
Madonna Huang and Caroline Lemieux. 2024. Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3678722.3685532

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FUZZING '24, September 16, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685532

1 Introduction

To keep up with the ever-increasing pace of technology advancements, software development nowadays often requires short iterations of product life cycles [2]. Continuous integration/continuous deployment (CI/CD) is one of the most popular ways to enable fast and reliable releases of software products in both industry and open source [30]. Standard CI/CD pipelines include pushing the new code in commits to the repository and automating the build, test, and deployment phases to deliver the program to production [23]. The automation in CI/CD has shown effectiveness in improving development quality and productivity, enhancing an agile and collaborative working environment for developers[3].

A crucial part of CI/CD is testing. Testing can help detect software defects at an early stage before production. Normally, 75% of errors appear in the first year after release [29]. Therefore, locating and fixing errors in CI/CD may reduce software maintenance costs, which constitute around 60% to 80% of the total software life cycle costs [10]. If bugs remain uncaught by CI/CD, common practices such as code cloning may propagate the bugs to different places, further decreasing development efficiency [19].

One approach to increase the robustness of CI/CD pipelines would be to add automated testing. A widely adopted automated testing technique is fuzzing (fuzz testing), which feeds programs with malformed and unexpected inputs. The term was coined by Miller for a university operating system class in the late 1980s [18]. These days, fuzzing has been proved to efficiently expose security loopholes and software defects in various domains including network security, web services, FinTech systems, etc [4, 22, 27]. The launch of Google's free OSS-Fuzz testing service further popularized fuzzing, by helping discover over 8,800 vulnerabilities and 28,000 bugs fixed across hundreds of critical open source projects [6].

The automation and effectiveness of fuzz testing seems to meet the needs for CI/CD testing. The goal of fuzzing is to find inputs that have unusual behaviours such as causing as many unique program crashes as possible. However, it usually takes hours of run to exploit a fuzzer's potential to collect such inputs. As Klees et al. suggest in their paper on fuzzing benchmarking practices [12], a timeout of 24 hours is recommended for a rigorous evaluation of fuzzers. However, testing each pull request for 24 hours is infeasible in a realistic setting. Fowler notes that *10 minutes* is considered as a "reasonable" build time in CI/CD pipelines [12], which is substantially shorter than 24 hours [7]. Thus, directly following common fuzzing practices in CI/CD is in conflict with the original goal of such pipelines to update code bases quickly and frequently.

Fortunately, prior work by Klooster et al. has confirmed that running fuzzers in continuous security testing can cover important

bugs even with a short timeout of 10 minutes [13]. Despite showing that fuzzing in CI/CD is possible, their work focuses only on evaluating coverage-guided fuzzers like AFL++. The authors did not evaluate directed fuzzers like AFLGo as they claimed that their commit simulation will interfere with the fuzzing performance. However, directed fuzzers are tailored to perform patch testing [5], which is similar to the CI/CD scenario: both aim at testing whether the changed statements will introduce new security vulnerabilities. For this reason, we are interested in studying whether directed fuzzers can live up to their potential in continuous fuzzing, i.e., fuzzing in a CI/CD context. In order to do this, we will adopt a different strategy than Klooster et al. [13] to simulate commit-level fuzzing.

In this work, we plan to evaluate both directed and undirected fuzzers in a continuous testing setup. Similar to previous studies [13], we simulate the CI/CD pipelines by fuzzing patched code with a small time budget. We select Magma, a widely used fuzzing benchmark suite that consists of 9 open source projects with critical bugs and vulnerabilities, as our source of libraries and buggy patches. We inject only a single buggy patch at a time, to simulate commits containing each of these patches separately. We integrate AFLGo and Fuzz Factory Diff (FFD) as two exemplary directed fuzzers into Magma. Further, in a realistic CI/CD setup, we assume that the Program Under Test (PUT) has been thoroughly fuzzed at a prior version. Thus, instead of using the initial seed inputs provided by Magma as in prior work [13], we collect a seed corpus by running a 24-hour fuzzing campaign with AFL++. This allows us to investigate CI/CD fuzzing in a setting with a close-to-saturated seed corpus, which might also lead to different results compared to prior work.

We seek to investigate the following research questions:

- Are directed or undirected fuzzers more efficient at reaching code change locations in a CI/CD setting?
- Are directed or undirected fuzzers more efficient at triggering the bug(s) embedded in the code change?
- How easy it is to set up the directed and undirected fuzzers in CI/CD fuzzing?
- How sensitive are directed fuzzers to "bloat" in commits, i.e., can directed fuzzers still reach an injected bug if given additional target locations?

To answer these questions, our first experiment will investigate whether directed or undirected fuzzers can generate inputs that reach the changed code and trigger the injected bugs more effectively. To do this, we simulate commits by injecting a single buggy Magma patch at a time, and giving this patch as the target location to the directed fuzzers. In terms of evaluating bug-triggering capacity, there remains a threat to external validity: these patches, which exactly inject a bug, may not be representative of real commits. Therefore, we will add additional experiments to measure whether directed fuzzers' performance degrades when the target locations are broader than simply the inserted patch. In particular, we will add additional lines of code, beyond the buggy patch, as targets for each "commit", and run directed fuzzers on these augmented commits.

We have conducted a preliminary study by evaluating AFL, AFL++, libFuzzer, and AFLGo on a subset of the libraries provided by Magma. The contributions of our registered report are:

- Preliminary analysis and comparison of the fuzzing performance of different directed and undirected fuzzers in a simulated setting of CI/CD.
- Validating whether the suggested 10-minute timeout from previous findings is effective at exposing critical bugs if we start fuzzing with a saturated seed corpus.
- Evaluating whether previous findings that short fuzzing campaigns are effective at exposing critical bugs hold in a single-bug injection setting.
- Designing an evaluation plan to study the fuzzers on a broader benchmark set, as well as study the fuzzers' sensitivity of code changes.

We will conduct the full experiments upon the acceptance of this paper by: evaluating all the five fuzzers on all libraries and performing experiments evaluating directed fuzzers' sensitivity to the size of code changes. In the end, we aim at establishing a detailed guideline for continuous fuzzing, which will: compare and contrast directed and undirected fuzzers in the CI/CD testing setup, validate the recommended 10-minute timeout for both types of fuzzers, and summarize the fuzzers' capability to detect bugs when the code changes contain more than just bugs.

The rest of the paper is organized as follows. Section 2 explains the CI/CD testing and related background on fuzzing. Section 3 details the benchmarks, fuzzers, and CI/CD simulation in our experimental setup. Section 4 describes the implementation details. Section 5 analyzes the results in our preliminary evaluation. Section 6 explains our plan for the complete evaluation of this work. Section 7 lists the requirements that we plan to fulfill for revising the registered report. Section 8 outlines the related work and section 9 summarizes our current conclusion.

2 Background and Motivation

2.1 CI/CD Testing

Continuous Integration (CI) and Continuous Delivery (CD) are widely accepted engineering practices that enable organizations to have frequent and steady releases of new features and products, improve code quality, and increase team productivity [26]. The CI process includes software development, building, and testing. The CD process further extends CI by automatically and continuously deploy applications to a production environment [26]. In the CI/CD pipelines, small and frequent updates are encouraged: previous study finds that around 21 commits are made per day for 575 open source projects [13].

Testing each commit is impractical in day-to-day development. Indeed, reducing the costs in CI/CD pipelines is a long-lasting problem [26]. Shahin et al. suggest supporting automated testing in CI/CD to reduce repetitive work and test case execution costs [26]. On the other hand, they note that the lack of automated testing approaches and the poor test quality are challenging for successfully adopting CI/CD practices [26]. This makes fuzzing, which automates the input generation and testing process, seem like a good fit in continuous testing. Note that in this paper, we use the

term fuzzers to refer to programs that automatically generate test inputs and feed these inputs into the PUT to detect bugs.

There are two common ways of fuzzing a program: either test the compiled binary of the PUT directly with the generated inputs or feed inputs into a fuzzing driver that serves as the main entry point to execute a specific function [13]. In either case, fuzz testers do not need to manually write and run tests themselves as they do in traditional deployment pipelines [26]. Shahin et al. suggest that the poor infrastructure for automating tests can lead to the lack of fully automated testing in CI/CD practices [26]. Thanks to various mature fuzzing platforms such as OSS-Fuzz, fuzz testers do not need to implement additional infrastructures to automate tests. Test quality is also one of the concerns in automating deployment on a continuous basis [26]. For instance, having a high number of test cases with low test coverage or long running tests can significantly slow down the deployment process [26]. Although fuzzers may not always generate inputs with good qualities such as being syntactically valid or having new code coverage, the automation of input generation makes it easier to quickly explore the input space.

However, the recommended timeout for fuzzing is 24 hours [12], which is infeasible for testing frequently in CI/CD. Rangnau suggest a "reasonable" threshold for build or test time in CI/CD is 10 minutes [23]. Using such a short timeout for fuzzing may not fully exploit the fuzzer's potential to generate inputs that maximize code coverage or discover "interesting" behaviours such as revealing bugs. Thus, investigating whether a short fuzzing campaign is effective at exposing vulnerabilities in the CI/CD setup is crucial for adopting fuzzing in such pipelines. There exists prior work that proposes a "reasonable" timeout in continuous fuzzing, which we discuss in details in the following subsection.

2.2 Prior Work of Fuzzing in CI/CD

Klooster et al. followed the Magma's approach of front-porting bugs from previous bug report to a stable version of libraries [13]. The authors simulate commits by applying the full set of Magma patches with bugs. This allows them to evaluate when the bugs are reached and triggered by fuzzers via Magma's lightweight instrumentation [13]. They choose not to use live commits as the ground truth knowledge of the existence and locations of bugs are unknown [13]. We adopt this front-porting approach for the same reason: to evaluate whether fuzzers can trigger the injected bugs.

Unlike what we propose, Klooster et al. injected all the bugs provided by Magma into each library [13]. However, this may influence the fuzzing results in [13] because the patches contain only bugs and most libraries have around 20 bugs injected. Clearly, it is unlikely for real commits in a pull request to contain so many bugs. In our study, we will apply patches with a single bug for each fuzzing experiment to simulate a more realistic commit. In addition, we plan to conduct the sensitivity experiments that add additional target locations—beyond the inserted bugs—to evaluate whether fuzzers can still trigger the bugs when the patch contains not only the bug but also other lines of code.

Further, seed inputs are crucial as the fuzzing performance may depend on whether the initial seeds can cover interesting code paths or provide good coverage as the starting point [21]. Prior work starts the continuous fuzzing with the set of initial seeds provided by Magma [13]. However, these initial seeds are sourced from the original code base of libraries and may not provide a good code coverage for fuzzing. Thus, we collect the initial seed corpus by running a fuzzing campaign with AFL++, one of the most popular state-of-the-art fuzzers. We use the recommended timeout of 24 hours, hoping to get a saturated initial corpus that can cover most of the reachable code of the target library. We want to use a saturated corpus so that the fuzzer will focus more on generating and mutating inputs that can cover the changed code, instead of the reachable code of the original library.

Moreover, Klooster et al. did not evaluate any directed fuzzers in their study: they claim that "[their] technique of simulating commits would have interfered with the results" [13]. Essentially, if the changed code contains exactly the bug to be injected, this might be seen as giving the directed fuzzers an unfair advantage. We agree with this concern but design additional experiments to address it. In particular, our sensitivity experiments will address this concern by creating a set of addition "augmented" commits, where we mask the locations of the injected bugs by adding additional, non-buggy, lines of code as targets.

Finally, the prior work also suggests that running per-commit fuzzing campaigns of 10 minutes achieves an optimal balance between the recommended testing time and the effectiveness of continuous fuzzing [13]. We plan to validate this finding for both directed and undirected fuzzers.

2.3 Directed Fuzzing

Directed fuzzing, as the name suggests, focuses on testing a specific program location of the PUT rather than the whole. There exists various directed fuzzing techniques including the taint-based directed white box fuzzing [8], directed grey box fuzzing [5], and directed symbolic execution [15]. We are interested in the directed grey box fuzzing that generates inputs with the goal to reach a specific set of program locations. This is applicable in a CI/CD setup because we want to find inputs that can exercise the changed code. And grey box fuzzing is the state-of-the-art in vulnerability detection without heavy-weight instrumentation [5]. For example, the most commonly used fuzzer AFL++ is a such a grey box fuzzer that generates inputs based on the code coverage feedback.

In our study, we choose to investigate AFLGo, which awards generating inputs with shorter distance to the target program locations [5]. In particular, AFLGo first calculates the distance of each basic block to the targets from the call graphs and intraprocedural control flow graphs (CFGs) generated at compile time. Then, AFLGo aggregates the distance values of each basic block exercised by the seed inputs to compute the mean value for minimization purpose. The static analysis stage of AFLGo sometimes can take up to hours to instrument the PUT and compute the distance values. Thus, for comparison purpose, we pick another directed greybox fuzzer Fuzz Factory Diff (FFD) that does not require such heavy computation at compile time. FFD uses a simpler heuristic that rewards generating inputs that can both exercise the target code locations and explore new coverage after hitting these targets [21]. Unlike AFLGo, FFD does not require the knowledge of each basic block's distance to target program locations. In addition to the inputs with new coverage, FFD also saves inputs when they exercise a new execution path after hitting the target code location [21].

3 Experimental Design

We outline the experimental setup for the complete evaluation in this section. For the preliminary evaluation, we ran experiments on a subset of the libraries and bugs provided by Magma, in order to give an idea of what our broader evaluation will do. Our preliminary evaluation does not include the fuzzer FFD and the experiments that investigate the sensitivity of fuzzers to the amount of changed (target) code.

3.1 CI/CD Simulation

A key difference between our work and the prior study [13] is that we simulate the commits in CI/CD by injecting *one bug* in each commit instead of injecting *all bugs* at the same time. Our simulated commits consist of an injected bug as we want to evaluate a fuzzer's capability of both *reaching* the changed code and *triggering* the vulnerability introduced by the change. As explained in Section 2, we inject one bug at a time to address the concern in the prior work [13] that injecting multiple bugs in commits might interfere with the fuzzing performance. In addition, using a close-to-saturate corpus might affect the fuzzing results. Most grey box fuzzers use code coverage as guidance, meaning that they will save inputs that exercise new parts of the program. When we start continuous fuzzing with a saturate corpus, executing these inputs already explore most of the reachable execution paths in a program before applying the code changes. Therefore, mutating these inputs might save some efforts of exploring the reachable execution space of the program before the changes. Consequently, the fuzzer might have a higher chance of generating inputs that can reach new execution paths, and in a continuous setup, that is the changed code.

We will run the fuzzing campaigns for each simulated commit that contains a single bug separately. We only simulate one commit for each experiment because pushing the changed code in a single or multiple commits does not affect how the fuzzers generate inputs and feed them into the PUT. We will include sensitivity experiments in the final evaluation to better simulate real commits that contains not just bugs, but also other changed lines, and study whether the fuzzers can still reach or trigger the bugs when the changed code does not reflect the precise location of the bugs. This is described precisely in Section 6. Since each library in Magma has multiple patches for bugs, we use the term *benchmark* to refer to a *pair* of library and the ID of the injected bug in the rest of the paper. As Table 1 shows, we assign a unique ID to each benchmark in the format of the library name followed by the numeric ID of the bug inserted.

In our CI/CD simulation, the configuration for each fuzzing campaign includes the library and bug ID to identify benchmarks, a fuzz target, and additional arguments to run the PUT. We also adopt Magma's instrumentation to decide whether a bug is *triggered* and *reached*. The source-code instrumentation in Magma reports a bug is *reached* when the line of the inserted buggy code is reached. A bug is *triggered* when the input satisfies the faulty condition [9]. Magma further distinguishes *detecting* a bug from *triggering* a bug. A bug is *detected* when a faulty behaviour is observed. In order to collect the bug detection data, we need to recompile the libraries with sanitizers. We will leave the evaluation of bug detection rate for the final evaluation.

Table 1: Benchmark details for the preliminary evaluation.

Benchmark	Library	Bug ID	Fuzz Target
libpng001	libpng	PNG001	libpng_read_fuzzer
libtiff012	libtiff	TIF012	read_rgba_fuzzer
libxml003	libxml2	XML003	libxml2_xml_read_memory_fuzzer
openssl001	openssl	SSL001	asn1
openssl009	openssl	SSL009	x509

In each experiment, we prepare the original library and the one with the bug inserted. We first collect the initial seed corpus from fuzzing the original code with a 24-hour timeout using AFL++. Then, we use this corpus to fuzz the code with changes for 10 minutes, which has been proved to show the effectiveness of continuous fuzzing by Klooster et al [13].

3.2 Benchmarks

Following the prior work [13], we choose the same fuzzing platform, Magma, which supports various fuzzers with 9 widely-used open-source libraries and 138 critical bugs as shown in Table 2. Similar to the reasons listed by Klooster et al. [13], we pick Magma as it provides the ground-truth for bugs and their exact locations in the PUT. This satisfies our needs for simulating commits that contain localizable bugs.

Magma already supports a variety of fuzzers including AFL, AFL++, and libFuzzer. The clean code structure in Magma also makes it easy for users to integrate new fuzzers. In particular, Magma organizes the fuzzers, libraries, and other utilities in separate folders. The folder of each fuzzer provides step-by-step scripts to build the fuzzers, build and instrument the libraries, run the fuzzing campaign, etc. Additionally, Magma instruments the target programs to report whether the bug is reached, triggered, or detected. We utilize this instrumentation to evaluate whether a fuzzer is effective at reaching and triggering the bugs in the patches applied to the libraries.

3.3 Fuzzers and Fuzz Targets

For undirected fuzzers, we pick AFL++ and libFuzzer, two state-of-the-art grey box fuzzers. We pick these two as they are already integrated into Magma, evaluated in the prior work of continuous fuzzing [13], and widely used in modern fuzzing platforms such as OSS-Fuzz [13]. For directed fuzzers, we select AFLGo and Fuzz Factory Diff (FFD) as explained in Section 2. Both AFLGo and FFD generate inputs based on heuristics that guide the inputs towards exercising a set of program locations in the format of file name followed by the line number of code. We additionally include AFL as both AFLGo and FFD are built on top of it. We want to examine how well AFLGo and FFD perform compared to the AFL baseline.

It is also important to select fuzz targets instead of fuzzing with all available targets to save computing resources [13]. We use the fuzz targets provided by Magma as the prior work does [13]. In our preliminary study, we select the fuzz targets outlined in Table 1 based on whether the changed code will affect the APIs being called in the fuzz targets. Identifying the relevant fuzz targets requires nontrivial efforts. Therefore, we plan to adopt the checksum-based method in [13] by calculating the checksum of the fuzz target before

Table 2: Libraries and bugs provided by Magma.

Library	File Type	Language	Fuzz Targets	Bugs
libpng	PNG	C	1	7
libtiff	TIFF	C	2	14
libxml2	XML	C	3	17
openssl	Binary blobs	C	6	20
libsndfile	Audio files	C	1	18
poppler	PDF	C++	3	22
sqlite3	SQL queries	C	1	20
php	Various	C	4	16
lua	Various	C	1	4

Table 3: Mean survival time of the bugs reached (R) and triggered (T) for the selected benchmarks. Rounded to the nearest second.

	AFL		AFL++		libFuzzer		AFLGo	
	R	T	R	T	R	T	R	T
libpng001	5	NA	5	NA	13	NA	5	NA
libtiff012	5	456	5	504	22	485	5	592
libxml003	5	NA	5	140	NA	NA	5	NA
openssl001	5	NA	170	NA	29	NA	5	NA
openssl009	5	NA	5	NA	NA	NA	5	NA

and after the commit to decide whether the target is affected by the changes. It is also worth noting that not all fuzz targets are suitable for our experiments due to the limitations of libFuzzer. libFuzzer, by design, tests specific functionalities of the PUT by passing the inputs to a fuzzing entry point, i.e., target function. Therefore, we cannot fuzz the compiled binary of the libraries with libFuzzer and thus exclude these from our selection of fuzz targets.

4 Implementation

We conduct all experiments using Magma as the uniform fuzzing platform. In this section, we describe the key changes we have implemented in Magma in preparation for our experiments.

Customization of fuzzing experiments. Magma inserts bugs into libraries by applying patches that contain the buggy code. By default, Magma only allows applying all the patches in the *patches* folder. Therefore, users need to move the patches outside of the folder to exclude inserting certain bugs. We modified the scripts in Magma to support either applying all patches (default) or the patches specified in a configuration file, allowing the users to switch each bug on and off at ease. We additionally add support for user-defined paths to corpus for each experiment: Magma only allows a fixed path to corpus. In this way, users can better customize each fuzzing campaign by changing the configuration files.

Integration of fuzzers. We integrated two directed fuzzers, AFLGo and FFD, into Magma. While the organization of scripts in Magma separates each building block of fuzzers neatly, we spent nontrivial amount of time debugging the instrumentation errors when building the projects with the new fuzzers. Build errors are common challenges in fuzzing. As Nourry et al. suggest, the incompatibility between fuzzing environment and project, the bugs in fuzzer, issues in build tools or external dependencies, as well as issues in corpus can lead to build failures [20]. In fact, this is not the first time that someone tries to integrate AFLGo into Magma: a 2021 issue was opened to discuss this matter [28]. However, the issue of supporting directed fuzzers in Magma remains open till the time we write this paper. The build errors when compiling libraries with AFLGo are still unresolved after more than three years since the issue was first brought up. In our work, we fixed all the build errors when building the targets with AFLGo.

Automation of Generating Target Program Locations. AFLGo requires instrumenting the libraries twice: one pass to compute the call graphs and intraprocedural CFGs in order to compute the distance of the seed inputs to the target program location, and the other pass to instrument the libraries with the computed distances. We further automate the process of generating the locations of the changed code to target files for both AFLGo and FFD. Previous attemps to integrate AFLGo into Magma [28] retrieves the information of the changed code from pre-generated target files. However, this approach is not scalable when the code changes or the number of benchmarks is huge. We use the external tool *showlinenum* as recommended in the AFLGo documentation to generate the program locations from the output of git diff [1]. This automation is beneficial for fuzzing large commits or running experiments on multiple benchmarks.

5 Preliminary Results

We select five benchmarks as detailed in Table 1 to run the preliminary evaluation of the four fuzzers: AFL, AFL++, libFuzzer, and AFLGo. We do not include FFD or perform the experiments the sensitivity of fuzzers due to time constraints. Each experiment for a single benchmark is repeated 10 times to reduce the effect of randomness in fuzzing [12]. In our proposed evaluation, we will run these experiments for all the five fuzzers on many more benchmarks, as well as run our sensitivity experiments.

We perform all preliminary experiments on a machine with Intel core i7-12700K processor and 64 GB memory. We summarize the findings as below.

5.1 Can fuzzers reach the inserted bugs?

As Figure 1 shows, AFL, AFL++, and AFLGo generated inputs that can reach the bugs in the changed code for all the five benchmarks. libFuzzer performed the worst as the inputs generated can only reach the bugs for three benchmarks. Table 3 shows the mean survival time for the bugs in all experiments. We adopt Magma's approach that uses the Kaplan-Meier estimator to model a bug's survival function. The shorter the survival time is, the better a a fuzzer's performance is because the survival function computes the probability that a bug remains undiscovered (i.e., "survived") within a given time [9]. As shown in Table 3, AFL and AFLGo are most effective at generating inputs that reach the bugs in the changed code. Both have a mean survival time of 5 seconds for the 10 repetitions of experiments per benchmark. This suggests that on average, it takes 5 seconds until AFL and AFLGo generates an input that can reach the inserted bug. AFL++ performed slightly worse as the mean survival time for the openssl001 benchmark is 170 seconds (around 3 minutes).

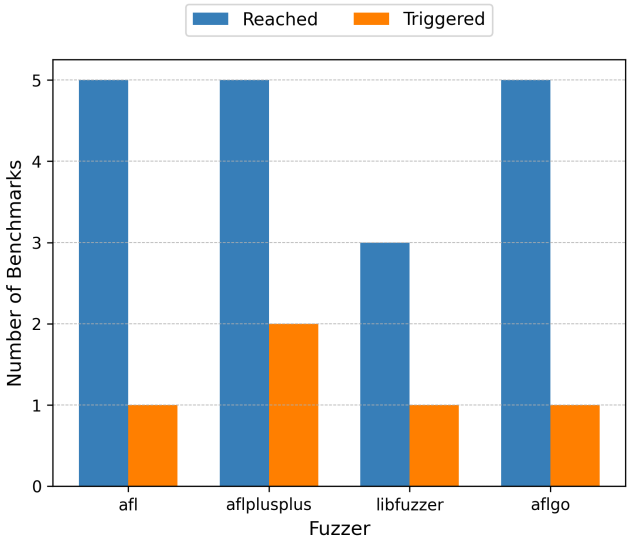

Figure 1: The number of benchmarks that the fuzzers have generated at least an input that can reach or trigger the bug in the changed code.

Based on the number of benchmarks for which the fuzzers have successfully reached the inserted bugs and the mean survival time, AFL and AFLGo are the best-performing fuzzers. Since one is undirected and the other is directed, we need the complete evaluation to conclude which type of fuzzers is more effective.

5.2 Can fuzzers trigger the inserted bugs?

Regarding the bug-triggering capability, AFL++ performed the best as it generated inputs that triggered the bugs in two benchmarks while the rest only triggered the bugs in one benchmark. A short timeout of 10 minutes seems insufficient for fuzzers to generate bug-triggering inputs for most of the benchmarks. libtiff012 is the only benchmark that all fuzzers have generated inputs that trigger the inserted bug. However, none of the fuzzers have triggered the bugs in libpng001 and the two openssl benchmarks. The time to trigger the bug also differs across benchmarks. It took 140 seconds (around 2 minutes) for AFL++ to trigger the bug in libxml003 while it took three times as long to trigger the bug in libtiff012. The results suggest that it is much harder for the fuzzers to generate inputs that can not only reach but also trigger the inserted bugs. Moreover, 10 minutes may not allow a fuzzer to live up to its potential to generate inputs that can reach and trigger the inserted bugs.

It is worth noting that libFuzzer has the highest execution counts for most benchmarks except for libxml003 as Table 6 indicates. However, it is clear from Figure 1 that libFuzzer also has the least number of bugs reached and triggered. Within the 10-minute fuzzing session, having a higher number of executions does not guarantee better bug-finding capability.

5.3 How easy it is to set up the fuzzers in CI/CD simulation?

It is straightforward to run the undirected fuzzers within the Magma framework. However, running directed fuzzers sometimes might fail as it requires the correct target code locations and the corresponding target functions. In the preliminary study, we correctly generated the target program locations for the selected benchmarks using the automatic approach suggested by AFLGo. We plan to test whether the automatic approach works for all benchmarks in the complete evaluation.

Table 4: The build time of fuzzers, rounded to the nearest second.

Fuzzer	AFL	AFL++	libFuzzer	AFLGo
Build Time (s)	4	20	3	458

Table 5: The instrumentation time of libraries, rounded to the nearest second.

	AFL	AFL++	libFuzzer	AFLGo
libpng001	15	18	9	22
libtiff012	25	80	38	43
libxml003	33	71	32	37
openssl001	122	256	62	280
openssl009	223	141	123	337

Table 6: Mean execution counts during the 10-min fuzzing campaigns, rounded to the nearest thousand.

	AFL	AFL++	libFuzzer	AFLGo
libpng001	19,719	3,812	132,108	22,115
libtiff012	13,250	2,502	13,421	9,284
libxml003	8,155	1,803	7,391	5,296
openssl001	120	83	190	50
openssl009	1,294	336	26,074	600

As Table 4 shows, the build time for undirected fuzzers is pretty short (3-20 seconds). However, the build time for AFLGo is around 7 minutes, so it might be too time-consuming to use AFLGo in CI/CD without compiling it in advance. As Table 5 shows, libFuzzer takes the least average time to instrument benchmarks for four out of the five benchmarks. For AFLGo, we use the python version of the code to generate the distance values, which is claimed to be faster than the original shell scripts [1]. Surprisingly, although AFLGo requires instrumenting libraries twice, it takes roughly the same time as AFL and AFL++ to compile the benchmarks. Note that the two openssl benchmarks are longer than others to be instrumented, with the shortest time of around 1 minute and the longest time of around 5 minutes. This is consistent with the fact that the openssl library has the largest repository size [13]. However, such a long instrumentation time might not be suitable in a CI/CD context because reducing the testing time is crucial for optimizing pipeline efficiency [26].

6 Plan for a Complete Evaluation

We propose the following plan for the complete evaluation to better answer the four research questions in Section 1.

We will evaluate all the five fuzzers (AFL, AFL++, libFuzzer, AFLGo, and FFD) on benchmarks selected from the libraries and fuzz targets provided by Magma. We use the term benchmark to refer to a *pair* of library and corresponding fuzz target. There are 9 libraries in total as shown in Table 2. We plan to cover all the libraries with a subset of available fuzz targets. In this way, we can compare the fuzzing performance across different libraries and the performance across different fuzz targets for the same library. Since the number of fuzz targets and bugs vary for different libraries, we set a cap on the number of fuzz targets and bugs per library. In particular, we plant allow for 3 fuzz targets and 10 bugs per library at maximum. The total number of benchmarks we aim at is 50 but not all the fuzz targets and bugs provided due to the limited computing resources.

Benchmark selection. Ideally, we should select the fuzz targets and bugs randomly to avoid introducing biases. However, it is not always possible for the fuzzer to generate inputs that can exercise the inserted code with the selected fuzz targets. Recall that we run one fuzz target for each library with a single bug inserted. If the fuzz target does not invoke any functions that involve the changed code, we waste the time to fuzz the unchanged parts of the library. To address this issue, we plan to use the checksum-based approach in [13] to calculate and compare the checksum of a fuzz target before and after applying the code changes. If the checksum remain the same after the code changes, then it indicates that the fuzz target might not be relevant for fuzzing the changed code. We hope to prune the fuzz targets that are unable to exercise the changed code with any inputs.

Sensitivity experiments. We will add experiments to investigate the directed fuzzers' sensitivity to the changed code. In particular, we want to study whether a directed fuzzer like AFLGo can still reach and detect the inserted bugs when the changed code contain not only bugs but also other lines of code (LOC). Both AFLGo and FFD generate a target file that contains the target program locations with the format "filename: line number of the target code". The target file will be used to compute the heuristics used in the two directed fuzzers. Thus, we plan to insert additional target LOC into this target file in the same format. The configurations for the sensitivity experiments are as follows:

- *Same-file.* set 2X LOC in the file with the bug as the additional target LOC. For patch length N, we will randomly choose $2N$ lines within the patched source file (but outside the patch) to add to the target list.
- *Same-project.* set 4X LOC in different files including the file with the bug as the additional target LOC. For patch length N, we will randomly choose 2 files in the project, and $2N$ lines within those files to add to the target list.

The numbers in the configurations are subject to change. We will collect the average number of LOC per pull request based on the recent commit histories of the libraries. Then we will use this metric to decide how many more LOC to insert into the target file in order to better simulate a real CI/CD setting.

Timeout experiments. We plan to validate whether the proposed 10-minute timeout [13] is enough for fuzzers to generate inputs that can reach and trigger the inserted bugs. If, however, most fuzzers cannot find such inputs with the 10-minute timeout, we plan to conduct additional experiments to examine which timeout can balance the short time budget in CI/CD pipelines and fuzzing effectiveness. The candidate timeouts are 15 minutes and 30 minutes, because timeouts longer than these two might not be suitable for CI/CD pipelines. We will run the timeout experiments only on the bugs missed by the first set of evaluation with the 10-minute timeout. We want to see if the fuzzers are able to find the bugs that are unreachable with the 10-minute timeout given more fuzzing time. Although the prior work [13] has shown that increasing the time budget for per-commit fuzzing campaign does not boost the fuzzing effectiveness very much, we would like to see whether this also holds for directed fuzzers.

Additional evaluation metrics. In the preliminary evaluation, our evaluation metrics are: the statistics related to the bugs reached and triggered, the build time of fuzzers, the instrumentation time of libraries, and the mean execution counts during fuzzing. We will add the following metrics in the complete evaluation:

- The branch coverage of the changed code.
- The branch coverage of the entire library after hitting the changed code.
- The number of times the fuzzer detects the bugs and the mean survival time of the bugs detected.

With the additional metrics, we hope to answer which fuzzer performs better in terms of increasing the code coverage of the changed code and encouraging exploring deeper in the program after hitting the changed code. Moreover, we want to study if these coverage measures are correlated to a fuzzer's capability of generating inputs that reach, trigger, and detect the inserted bugs in the changed code. In addition, we want to compare the bug detection rate for all the five fuzzers. We will add statistical tests to compare the quantitative results in the complete evaluation for better assessment.

7 Revision Requirements

In the revision of the registered report, we plan to extend the scope of experiments in terms of the choice of directed fuzzers and the number of benchmarks. We plan to include more recent directed fuzzers such as FishFuzz [31] and DAFL [11] because both AFLGo and Fuzz Factory Diff are developed before 2020. We will run fuzzing campaigns on at least 50 benchmarks from Magma, where each benchmark refers to a pair of library in Magma and the ID of the injected bug. As for the experiment design, we will provide more detailed explanation and evidence for configurations including but not limited to the fuzzing setup for collecting the initial seed corpus, where and how to add additional target code locations for the sensitivity experiments, and settings for different fuzzers. For example, instead of running AFLGo with the default exploration mode first, we will change to run AFLGo with the exploitation mode from the first second of the fuzzing campaign. This is because AFLGo switches from exploration with coverage-based fuzzing algorithm to exploitation with directed fuzzing algorithm after 20 minutes of fuzzing. Since we start fuzzing experiments with a saturated seed corpus, we want to fully investigate the performance of AFLGo's directed fuzzing logic in the CI/CD context and thus we plan to start fuzzing with the exploitation mode of AFLGo.

8 Related Work

Testing plays an important role in CI/CD pipelines. The concept of continuous testing originally referred to executing unit tests on a continuous basis [16]. Now the term refers to running tests in the background to provide rapid and frequent feedback for developers to detect critical errors before deploying applications to a production environment [16].

Continuous testing can enhance development productivity. Saff and Ernst propose a model for investigating the usefulness of continuous testing [24]. Their study first shows that regressions errors caught earlier are easier to fix [24]. Then they evaluate three techniques to reduce the development time wasted on discovering and fixing regression errors. They compare running continuous testing and manually running tests with different test frequencies and test case prioritization strategies. Saff and Ernst conclude that continuous testing, that uses limited CPU resources to run tests in a continuous setup, can reduce the wasted development time by 92-98% over the other two approaches [24]. The authors further show in a study that developers that used continuous testing are three times more likely to complete the coding task before the deadline [25]. Most participants in their study confirmed the usefulness of continuous testing and found it helpful for them to write better code faster [25].

Challenges like reducing test time, increasing the visibility of test results, and supporting (semi-)automated testing are important for testing in a continuous setup [26]. However, testing each commit is not sustainable due to limited computing resources [13]. A study of Google-scale continuous testing has confirmed that code modified more often is more likely to cause breakages than the one modified less often [17]. Motivated by similar findings, Zhu and Böhme propose a regression grey box fuzzer that focuses on fuzzing code that was changed more recently [32]. However, they found that regression errors are hard to find even with a multi-day time budget and that fuzzing efforts are needed for the newly changed code even after a project is well-fuzzed [32]. To better address detecting bugs in recent code changes, Klooster et. al evaluate three modern fuzzers in a continuous setup [13], which inspires our work in this paper. Klooster et al. have shown that selecting fuzz targets can reduce the computational resources for testing and running fuzzers for 10 minutes strikes a good balance between the recommended continuous testing time and fuzzing effectiveness [13]. Following their findings, we construct the experimental plan to select fuzz targets based on checksum, include directed fuzzers for comparison, and use the same fuzzing timeout. We compare and contrast our work with this prior work in Section 2.

In our CI/CD simulation, we run fuzzing campaigns for changed code. Our setup and findings might be applicable to patch testing, which automatically tests code patches [15]. To test software patches, Marinescu and Cadar propose KATCH, an automated technique that uses symbolic execution to generate inputs that can quickly reach the code patch [15]. Their experiments show that automatic techniques can increase patch coverage, which is genuinely hard to reach a high value, and detect bugs during testing [15]. Kuchta et al. further proposes executing the old and new versions of the code in the same symbolic execution instance, with the old shadowing the new one [14]. The shadow symbolic execution technique generates inputs when the execution of the old and new code versions diverges in order to explore new behaviours of the new code [14]. However, the symbolic executions used in these prior work of patch testing require heavy-weight program analysis and constraint solving, which might not be scalable for large-scale projects or working environments that need rapid test feedback. If our work can show that running fuzzers for a short period of time can reach and detect bugs in the changed code, our experimental setup might be useful for patch testing. For example, we can collect an almost saturated corpus from fuzzing the old version of code for 24 hours, and then run the fuzzers for on the patched version for 10 minutes. This CI/CD simulation approach might lead to different fuzzing performance than using traditional patch testing tools like KATCH.

9 Conclusion

In the preliminary evaluation, a timeout of 10 minutes is sufficient for both directed and undirected fuzzers to generate inputs that can reach and trigger some but not all bugs inserted into the libraries. AFL++ seems to be the best-performing fuzzer from the preliminary results. AFL++ triggered the bugs in two out of the five selected benchmarks while the others triggered the bug in only one benchmark. However, with a limited set of benchmarks, we cannot conclude which fuzzer performs the best in continuous fuzzing yet. Additionally, the build time for AFLGo is around 7 minutes, which makes it a less optimal choice of fuzzer especially when the time budget is small. However, the instrumentation time of libraries for AFLGo when using the fast python version of code to compute seed distance is almost compatible to the time for other fuzzers. This suggests that if we have already compiled AFLGo in the CI/CD pipelines, we can still use AFLGo to test programs without spending most of the 10-minute time budget on building the fuzzer. Overall, using a 10-minute timeout under a single bug injection setup in continuous fuzzing experiments allow both directed and undirected fuzzers to generate inputs reaching the changed code. We need more experiments to compare the bug-triggering capabilities of fuzzers and conclude either directed or undirected fuzzers perform better in the CI/CD context. We will summarize our findings after the complete evaluation, aiming to provide a detailed guidelines for continuous fuzzing.

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. NN66001-22-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

References

[1] 2017. AFLGo: Directed Greybox Fuzzing. https://github.com/aflgo/aflgo. Accessed: 2024-06-20.

[2] Samar Al-Saqqa, Samer Sawalha, and Heba Abdelnabi. 2020. Agile Software Development: Methodologies and Trends. *Int. J. Interact. Mob. Technol.* 14 (2020), 246–270. https://api.semanticscholar.org/CorpusID:225548331

[3] S.A.I.B.S. Arachchi and Indika Perera. 2018. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In *2018 Moratuwa Engineering Research Conference (MERCon)*. 156–161. https://doi.org/10.1109/MERCon.2018.8421965

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[6] Oliver Chang. 2023. Taking the Next step: OSS-Fuzz in 2023. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html. Accessed: 2024-06-13.

[7] Martin Fowler. 2024. Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html. Accessed: 2024-06-13.

[8] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[9] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

[10] Siim Karus and Harald Gall. 2011. A Study of Language Usage Evolution in Open Source Software. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA) *(MSR '11)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/1985441.1985447

[11] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing Guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4931–4948. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[12] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[13] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 25–32. https://doi.org/10.1109/SBFT59156.2023.00015

[14] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. 2018. Shadow Symbolic Execution for Testing Software Patches. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 10 (sep 2018), 32 pages. https://doi.org/10.1145/3208952

[15] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/2491411.2491438

[16] Maximiliano A Mascheroni and Emanuel Irrazábal. 2018. Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review. *Computación y Sistemas* 22, 3 (2018), 1009–1038. https://doi.org/10.13053/cys-22-3-2794

[17] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[18] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[19] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. An Empirical Study on Bug Propagation through Code Cloning. *Journal of Systems and Software* 158 (2019), 110407. https://doi.org/10.1016/j.jss.2019.110407

[20] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers during Fuzzing Activities. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 14 (nov 2023), 26 pages. https://doi.org/10.1145/3611668

[21] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

[22] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[23] Thorsten Rangnau, Remco v. Buijtenen, Frank Fransen, and Fatih Turkmen. 2020. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. 145–154. https://doi.org/10.1109/EDOC49727.2020.00026

[24] D. Saff and M.D. Ernst. 2003. Reducing Wasted Development Time via Continuous Testing. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* 281–292. https://doi.org/10.1109/ISSRE.2003.1251050

[25] David Saff and Michael D. Ernst. 2004. An Experimental Evaluation of Continuous Testing During Development. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/1007512.1007523

[26] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

[27] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3597926.3598059

[28] spencerwuwu. 2021. Support Directed Fuzzing #62. https://github.com/HexHive/magma/issues/62. Accessed: 2024-06-20.

[29] J.Christopher Westland. 2002. The Cost of Errors in Software Development: Evidence from Industry. *Journal of Systems and Software* 62, 1 (2002), 1–9. https://doi.org/10.1016/S0164-1212(01)00130-3

[30] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 471–482. https://doi.org/10.1109/ICSME52107.2021.00048

[31] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1343–1360. https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

[32] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. https://doi.org/10.1145/3460120.3484596

Received 2024-06-21; accepted 2024-07-22