Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup

MADONNA HUANG, University of British Columbia, Canada
CAROLINE LEMIEUX, University of British Columbia, Canada

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 found that some bugs could be reached and triggered in timeouts as short as 10 minutes, suggesting the idea is feasible. 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. We evaluate both directed and undirected fuzzers in a simulated CI/CD environment. We use Magma as a source of benchmarks and run fuzzers for 10 minutes. We start the fuzzing process from a corpus generated by a 24-hour fuzzing run on a previous version of the code, and run the fuzzers on versions of Magma programs with a single bug injected. As the Magma patches provide very precise bug-location information as targets to the directed fuzzers, this may give an unfair advantage to the directed fuzzers. Thus, we also conduct experiments to evaluate whether directed fuzzers are sensitive to "bloat" in commits, i.e., whether directed fuzzers can still reach the bugs if given additional target locations. Surprisingly, we find that AFL and TuneFuzz, neither of which is directed to the targets from the commits, are the fastest at reaching and triggering bugs. AFL wins if we consider the time to build and instrument the program under test. This result is explained by contrasting bug reaching/triggering time and the time to load the input corpus. We find that nearly all of the bugs reached and triggered in our experiments are reached/triggered while loading the corpus generated by the 24-hour fuzzing run, rather than while generating new inputs. This suggests that for commits as small as Magma patches, using inputs generated from a long-running fuzzing run as regression tests in CI/CD gives nearly as much bug-revealing power as conducting a short-running fuzzing session.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; Agile software development.

Additional Key Words and Phrases: continuous integration, fuzz testing, directed fuzz testing, directed greybox fuzzing

ACM Reference Format:
Madonna Huang and Caroline Lemieux. 2026. Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 35 pages. https://doi.org/10.1145/3723159

1 Introduction

To keep up with the ever-increasing pace of technology advancements, software development nowadays often requires short iterations of product life cycles [5]. 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 [51]. Standard CI/CD pipelines include pushing new code in commits to a repository and automating the build, test, and deployment phases to deliver the program to production [42]. The automation in CI/CD has improved development quality and productivity, enhancing an agile and collaborative working environment for developers [6].

Authors' Contact Information: Madonna Huang, University of British Columbia, Vancouver, Canada, madonna.huang@gmail.com; Caroline Lemieux, University of British Columbia, Vancouver, Canada, clemieux@cs.ubc.ca.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
Manuscript submitted to ACM

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 [49]. 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 [22]. If bugs remain uncaught by CI/CD, common practices such as code cloning may propagate bugs to different places, further decreasing development efficiency [36].

One approach to increase the robustness of CI/CD pipelines would be to add automated testing. A widely adopted automated testing technique is *fuzzing* (or *fuzz testing*), which feeds programs with malformed and unexpected inputs. The term *fuzz* was coined by Miller for a university operating system class in the late 1980s [35]. These days, fuzzing has proven to efficiently expose security loopholes and software defects in various domains, including network security, web services, and FinTech systems [8, 40, 47]. The launch of Google's free OSS-Fuzz testing service further popularized fuzzing. As of May 2025, it has helped discover over 13,000 vulnerabilities and 50,000 bugs across 1,000 open source projects [31].

The automation and effectiveness of fuzz testing seem to meet the needs of 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 runtime to exploit a fuzzer's potential to collect such inputs. As Klees et al. suggest in their paper on fuzzing benchmarking practices [24], a timeout of 24 hours is recommended for a rigorous evaluation of fuzzers. However, testing each pull request for 24 hours is not realistic. Fowler notes that *10 minutes* is considered a "reasonable" build time in CI/CD pipelines [24], which is substantially shorter than 24 hours [15]. 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 [25]. 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. However, directed fuzzers are tailored to perform patch testing [9], which is similar to the CI/CD scenario: both aim at testing whether the changed statements will introduce new security vulnerabilities. But, Klooster et al. [25] did not evaluate these as they claimed their commit simulation would interfere with fuzzing performance. In order to evaluate directed fuzzers in this scenario, we adopt a different strategy than Klooster et al. [25] to simulate commit-level fuzzing, injecting only one bug per project. Further, Klooster et al. [25] use the default Magma corpus to start fuzzing from. We assume that if developers are interested in fuzzing in CI/CD, they may also be conducting longer-running fuzzing runs on a few checkpoint versions of their program. Thus, we are interested in investigating how bug discovery results change when starting from a seed corpus generated by a long-running fuzzing run on a previous version of the program, rather than the smaller Magma corpora.

In this work, we evaluate both directed and undirected fuzzers in a continuous testing setup. Similar to previous studies [25], we simulate the CI/CD pipelines by fuzzing patched code with a small time budget. We select Magma [19], a widely used fuzzing benchmark suite that consists of 9 open source libraries with critical bugs and vulnerabilities, as our source of libraries and buggy patches. For each library, Magma contains a number of patches, each injecting a single bug. Unlike prior work, we inject only a single buggy patch at a time, to simulate commits containing each of these patches separately. Our benchmarks are tuples of (*library*, *injected bug ID*, *fuzz target*): we randomly select 50 of these tuples for our experiments. 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 [25], 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.

We integrated AFLGo [9], Fuzz Factory Diff (FFD) [39], TuneFuzz [52], and WindRanger [13] as exemplary directed fuzzers into Magma. We compare these to undirected fuzzers AFL [50], AFL++ [14], and libFuzzer [41].

Our paper seeks to investigate the following research questions:

RQ1. Are directed or undirected fuzzers more efficient at reaching code change locations in a CI/CD setting?
RQ2. Are directed or undirected fuzzers more efficient at triggering bugs in a code change in a CI/CD setting?
RQ3. How easy is it to set up the directed and undirected fuzzers in CI/CD fuzzing?
RQ4. How sensitive are directed fuzzers to "bloat" in commits, i.e., can directed fuzzers still reach an injected bug if given additional target locations?

Our first set of experiments investigates whether directed or undirected fuzzers can generate inputs that *reach the changed code* and *trigger the injected bugs* faster. 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 evaluating the 10-minute timeout, we also record the instrumentation time each fuzzer takes on each benchmark, and present the results on reached and triggered code when adding this instrumentation time.

In terms of evaluating bug-triggering capacity, our plan admits a threat to external validity: the Magma patches, which exactly inject a bug, may not be representative of real commits. Therefore, we add additional experiments to measure whether directed fuzzers' performance degrades when the target locations are broader than simply the inserted patch. In particular, we add additional lines of code beyond the buggy patch as targets, and run directed fuzzers on these augmented "commits". For realism, we pull these additional lines of code from actual commits for the target projects.

In our preliminary study [20], we evaluated AFL, AFL++, libFuzzer, and AFLGo on 5 benchmarks[1] selected from the libraries provided by Magma. This study showed that the 10-minute timeout allows for short fuzzing campaigns to discover bugs. In this paper, we conduct the complete experiments as in accordance with our experimental plan [20], developed in conjunction with the reviewers of the registered report for this paper. Specifically, we run the complete evaluation of 7 fuzzers[2] on 50 benchmarks and examine directed fuzzers' sensitivity to the size of code changes. In addition, we also analyze auxiliary metrics, such as coverage of the program and target function, executions performed by each fuzzer, and true fuzzing time, to better understand our results.

In the end, we compare and contrast directed and undirected fuzzers in a CI/CD testing setup, validate the recommended 10-minute timeout for both types of fuzzers, and summarize the fuzzers' capability to reveal bugs when the code changes contain more than just bugs. Our work contributes the following:

- A validation of whether the suggested 10-minute timeout from previous findings is effective at exposing critical bugs if we start fuzzing with a close-to-saturated seed corpus.
- An evaluation of whether previous findings that short fuzzing campaigns are effective at exposing critical bugs hold in a single-bug injection setting.
- An evaluation of directed fuzzers' sensitivity to bloat in target locations.
- An analysis of whether bugs are found while loading the saturated seed corpus, or while truly fuzzing.
- A comparison of the fuzzing performance of directed and undirected fuzzers in a simulated CI/CD setting.

[1]In our context, benchmarks are (library, injected bug, fuzz target) tuples.
[2]We run AFLGo with two configurations, resulting in 8 total fuzzer configurations in our results.

The rest of the paper is organized as follows. Section 2 provides background on CI/CD testing, our CI/CD simulation procedure, and directed fuzzing. Section 3 details the complete experimental setup, including: benchmarks, fuzzers, configurations, and implementation. We present the results of experiments in Section 4. Section 5 gives a deeper discussion of these results. Section 6 outlines other related work, and Section 7 concludes.

2 Background

We first provide an overview of continuous integration and delivery, and the possible role of fuzzing in this process. We then outline previous work on fuzzing in CI/CD, and how our work differs from it. Then we provide background on directed fuzzing, and detail the fuzzers we evaluate in this paper, as well as our reasons for selecting them.

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 [45]. The CI process includes software development, building, and testing. The CD process further extends CI by automatically and continuously deploying applications to a production environment [45]. In CI/CD pipelines, small and frequent updates are encouraged: a previous study finds that around 21 commits are made per day for 575 open source projects [25].

Reducing costs in CI/CD pipelines is a long-lasting problem [45]. Shahin et al. suggest supporting test automation in CI/CD to reduce repetitive work and test case execution costs [45]. 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 [45]. 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 program under test (PUT) to detect bugs.

In CI/CD, developers do not manually run tests as they do in traditional deployment pipelines [45]. Shahin et al. suggest that the poor infrastructure for automating tests can lead to the lack of fully automated testing in CI/CD practices [45]. Test quality is another concern in automating deployment on a continuous basis [45]. For instance, having a high number of test cases with low test coverage or long-running tests can significantly slow down the deployment process [45]. Although fuzzers may not always generate inputs with good qualities, such as being syntactically valid or having new code coverage, they can quickly explore a large space of potentially bug-inducing inputs.

However, the recommended timeout for fuzzing is 24 hours [24], which is infeasible for testing frequently in CI/CD. Rangnau et al. suggest a "reasonable" threshold for build or test time in CI/CD is 10 minutes [42]. 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 detail next.

2.2 Prior Work of Fuzzing in CI/CD

Klooster et al. followed Magma's approach of front-porting bugs from the previous bug report to a stable version of libraries [25]. Magma [19] is an open-source benchmark that supports a variety of fuzzers, including AFL, AFL++, and libFuzzer. Magma includes 138 bugs in 9 open-source libraries that are commonly used in fuzzing evaluation. Additionally, Magma provides instrumentation for programs to report whether the bug is reached, triggered, or detected. Klooster et al. 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 [25]. They choose not to use live commits, as the ground truth knowledge of the existence and locations of bugs is unknown [25]. 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 [25]. This is how Magma works by default, and it allows a single fuzzing run to uncover multiple bugs. However, this may influence their fuzzing results [25], because the patches contain only bugs, and most libraries have over 10 bugs injected. Clearly, it is unlikely for real commits in a pull request to contain so many bugs. In our study, we apply patches with a single bug for each fuzzing experiment to simulate a more realistic commit.

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 [39]. Klooster et al. start the continuous fuzzing with the set of initial seeds provided by Magma [25]. However, these initial seeds are sourced from the original code base of libraries and may not provide good code coverage for fuzzing. Further, this may not be a realistic simulation of fuzzing in a CI/CD setting, where we might assume previous commits might have been fuzzed long enough to collect a close-to-saturated corpus. Thus, we collect the initial seed corpus by running a fuzzing campaign with AFL++ [14]. We use a timeout of 24 hours, hoping to get a close-to-saturated initial corpus that can cover most of the reachable code of the target library. Hopefully, this will allow the fuzzer to 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" [25]. 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, we conduct sensitivity experiments that address this concern by creating a set of additional "bloated" commits, where we disguise 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 [25]. Our preliminary study found we could indeed reach and trigger bugs on 5 benchmarks within a 10-minute timeout [20]. Our experiments aim to validate this finding for both directed and undirected fuzzers, starting from a close-to-saturated corpus.

2.3 Directed Fuzzing

Directed fuzzing focuses on testing a specific program location of the program under test, rather than generally increasing coverage. There exist various directed fuzzing techniques including: taint-based directed whitebox fuzzing [16], directed greybox fuzzing [9], and directed symbolic execution [32]. We are interested in the directed greybox fuzzing that generates inputs with the goal of reaching a specific set of program locations. This is very relevant to a CI/CD setup where our priority is to find inputs that exercise the changed code.

In our study, we evaluate four directed fuzzers. Three of these assume a user-provided set of *targets*. This set is a list of `file name:line number`, where each line in a particular file is a single target. We call these *user-provided-target* directed fuzzers. One of the modern directed fuzzers we evaluate, TuneFuzz [52], is also characterized as a directed fuzzer, but targets all sanitizer-inserted code in the program under test. This is similar to *sanitizer-guided* fuzzing introduced by Österlund et al. [38]. We call these *auto-target* fuzzers to reflect that the targets are automatically generated without a user-provided target file.

First, we evaluate the seminal AFLGo, which rewards generated inputs with shorter distances to the target program locations [9]. 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 purposes.

The static analysis AFLGo performs during compilation—in order to compute distance values—can take quite a long time [46]. Thus, for comparison, we pick a directed greybox fuzzer on the other side of the spectrum: Fuzz Factory Diff (FFD) [39]. At compile time, FFD simply injects additional Fuzz-Factory-style "waypoints" to the code target locations. FFD uses a simple heuristic of saving—and prioritizing for mutation—inputs that can both exercise the target code locations and explore new coverage after hitting these targets [39]. Unlike AFLGo, FFD does not require knowledge of each basic block's distance to target program locations. Thus, it is exemplary of a more lightweight directed greybox fuzzer, which should not have the same compilation costs as AFLGo.

AFLGo and FFD were published in 2017 and 2019, respectively. We add two more modern directed fuzzers to our evaluation to evaluate whether the proposed improvements are significant in our CI/CD context.

First, we evaluate WindRanger [13], published in 2022. WindRanger is a user-provided-target fuzzer in the style of AFLGo. Including WindRanger allows us to evaluate whether a modern optimization of directed fuzzing behaves significantly better than AFLGo in our CI/CD context. In the WindRanger paper [13], WindRanger found bugs faster than AFL and AFLGo.

We attempted to compare with the 2023 DAFL [23] tool, but had difficulty integrating it into Magma due to its reliance on a static analysis compilation pass before the main target instrumentation. Further, DAFL only supports a single target—i.e., we would have to select a single patch line to target [2]. It is not clear how to do this in a realistic manner to simulate the CI/CD setting. It is also not clear how to select the target in our sensitivity experiments. In its evaluation, DAFL found bugs much faster than both AFLGo and WindRanger [23]. Geretto et al.'s reimplementation of DAFL had more mixed results [17], being faster than the reimplementation of AFLGo on some benchmarks, and slower on others.

Finally, we include TuneFuzz [52], a 2024 update to FishFuzz. FishFuzz is a 2023 directed fuzzer optimized for fuzzing with large numbers of targets [53]. In its evaluation, it directs fuzzing towards sanitizer-inserted code. TuneFuzz [52] is an updated version of FishFuzz that has compiles benchmarks more quickly. Note that both TuneFuzz and FishFuzz are what we call *auto-target* directed fuzzers, which direct fuzzing towards all sanitizer-inserted code, rather than a specific set of target locations. The SBFT'24 paper on TuneFuzz does mention the ability to set custom targets, but this feature is not documented in the codebase. Thus, we decided to run TuneFuzz as an auto-target fuzzer. This allows us to compare an auto-target fuzzer to a multitude of user-provided-target directed fuzzers. But because of this, TuneFuzz does not feature in our sensitivity evaluations: bloat in commits does not change the auto-target target identification.

For a full evaluation of the value of modern directed fuzzer optimizations, we refer the reader to Geretto et al.'s reimplementation of modern directed fuzzers [17], which we discuss further in Section 6. We reproduce part of Geretto et al.'s results in Section 6, to better compare their findings to ours.

2.4 Selected Fuzzers

The selected fuzzers in this work consist of both directed and undirected fuzzers.

For directed fuzzers, as explained in the previous subsection, we chose AFLGo and FFD to compare heavier- and lighter-weight instrumentation methods. We evaluate TuneFuzz as a modern auto-target directed fuzzer, and WindRanger as a modern update to user-provided-target directed fuzzing. We evaluate AFLGo with its default time to exploitation of

10 minutes [28] (meaning, in our timeout, it will be 10 minutes of exploration), as well as with time to exploitation set to 0 minutes. We call this second exploitation-only fuzzer AFLGoE. In our preliminary experiments, we ran only AFLGo with 10 minutes of exploration [20]. We run WindRanger with time to exploitation set to 0 minutes. These exploitation-only settings should let us evaluate the pure power of the directed fuzzing algorithm: our saturated seed corpus has been built with 24 hours of AFL++ exploration, and should provide plenty of starting points for exploitation.

For undirected fuzzers, we select the state-of-the-art fuzzers AFL++ [14] and libFuzzer [41], as well as AFL [50]. AFL++ is considered the state-of-the-art coverage-guided fuzzer: in a March 2025 FuzzBench evaluation, it achieves a score of 95.4 (compared to 85.4 and 82.4 for libfuzzer and AFL), and is the top fuzzer by rank [29]. Nevertheless, we include AFL as a baseline because AFL++, AFLGo, and FFD are all built upon it. AFL's performance should help us distinguish whether differences in ability to reach/trigger bugs come from directedness or from general fuzzing improvements.

We choose libFuzzer as it is not built upon AFL, which might have different performance compared with AFL-based fuzzers under the continuous fuzzing setup. While AFL++ can execute instrumented binaries that take inputs from a file or `stdin` [14], libFuzzer is directly linked with the library under test and consumes inputs via a test harness. A libFuzzer test harness is a function named *LLVMFuzzerTestOneInput* that calls the API in the library under test to take an array of bytes and perform some tasks [41]. However, the disadvantage of having such a specialized fuzz target is the additional amount of work to write the harness. In our experiments, we use the libFuzzer test harnesses provided by Magma [19].

3 Experimental Design

We now precisely explain our CI/CD simulation and outline the setup for the complete evaluation. We also provide a number of implementation and configuration details in Section 3.5.

3.1 Our CI/CD Simulation

Commits. A key difference between our work and the prior study on fuzzing in CI/CD [25] is that we simulate the "commits" by injecting *one bug patch* in each commit instead of injecting *all bugs* at the same time. We inject one bug at a time to address the concern in the prior work [25] that injecting multiple bugs in commits might interfere with the fuzzing performance for directed fuzzers. We also conduct sensitivity experiments (Section 3.4) to deal with the threat that these bug patches do not correctly simulate real-world commits. We do this by adding additional targets sourced from real-world commits to the directed fuzzer target list.

Starting Corpus. As aforementioned, we start fuzzing from a close-to-saturated input corpus, rather than the default Magma corpus. This simulates a continuously-operating CI/CD fuzzing pipeline that might have a close-to-saturated starting corpus from previous fuzzing runs. We create this close-to-saturated corpus by fuzzing the original code with the default Magma corpus for 24 hours, using AFL++. Then, we use the corpus resulting from this 24-hour run to fuzz the code with buggy patch inserted for 10 minutes.

Benchmark Identification. Since each library in Magma has multiple patches for bugs, we use the term "benchmark" to refer to the tuple of *(library, injected bug, fuzz target)*. We 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 [19].

Timeout. We use a fuzzing timeout of 10 minutes. The Magma system allows users to set timeouts for fuzzing sessions, but this timeout does not include the time to instrument the program under test for fuzzing. This is presumably to help reduce re-computation in regular fuzzing experiments: the program under test is instrumented once, and then multiple

worker processes run fuzzing campaigns on the program under test. However, in a CI/CD setting, the program under test will have to be built before the fuzzing campaign can be launched—so for some CI/CD users, the instrumentation time may need to be captured in the timeout. Thus, in our main experiment results, we report the time to reach and trigger bugs with and without the inclusion of the program under test build/instrumentation time.

3.2 Benchmarks

Following prior work [25], we choose the same fuzzing platform, Magma, which supports various fuzzers with 9 open-source libraries and 138 critical bugs, detailed in Table 1. Similar to the reasons listed by Klooster et al [25], 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. We use Magma's instrumentation to evaluate whether a fuzzer is effective at reaching and triggering the bugs in the patches applied to the libraries. The clean code structure in Magma also makes it easy for users to integrate new fuzzers. In particular, Magma organizes 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.

There are 9 libraries in total, as shown in Table 1. Our preliminary study only included 4 libraries with one fuzz target for each [20]. In our complete evaluation, we include 8 libraries with multiple fuzz targets for some of them. Thus, we can compare fuzzing performance across different libraries and the performance across different fuzz targets for the same library. We exclude lua because Magma does not provide any fuzz targets suitable for libFuzzer. We include all the other libraries and fuzz targets that can be used with libFuzzer in the complete evaluation. We do not consider benchmarks with *(buggy patch, fuzz target)* pairs where target binaries have the same checksum before and after adding the patch (see *Fuzz Targets* in the next section). We select 50 benchmarks that pass this checksum check (i.e., binaries differ before and after adding the patch) for our complete evaluation.

3.3 Fuzzers and Fuzz Targets

As described in Section 2, we pick 7 fuzzers. For undirected fuzzers, we pick AFL++ and libFuzzer, two state-of-the-art fuzzers, as well as AFL, on which many of our directed fuzzers are built. These are already integrated into Magma, evaluated in the prior work of continuous fuzzing, and widely used in modern fuzzing platforms such as OSS-Fuzz [25]. For directed fuzzers, we select AFLGo [9], Fuzz Factory Diff (FFD) [39], WindRanger [13], and TuneFuzz [52].

Fuzzer Setup. For AFL-based fuzzers, we use *afl-clang-fast* and *afl-clang-fast++* for instrumentation. This is consistent with the suggestion in the documentation of AFL++ for continuous fuzzing: the authors state that the LTO instrumentation has a much longer compile time and *afl-clang-fast++* is preferred [3]. We choose two configurations of AFLGo in the complete evaluation: AFLGo and AFLGoE. By default, AFLGo switches from the exploration phase with a coverage-based fuzzing algorithm to the exploitation phase with a directed fuzzing algorithm after 10 minutes of fuzzing [28]. This is what was evaluated in our preliminary experiments [20], and what we denote as "AFLGo". For "AFLGoE", we change the default setting to let AFLGo enter the exploitation phase *from the first second* of the fuzzing campaign. Once AFLGo enters the exploitation phase, the fuzzer will fuzz seeds that are closer to the targets, thus generating substantially more inputs from these seeds [9]. As we start with a close-to-saturated corpus, we believe this "full exploitation" setting ("AFLGoE") is the best to evaluate the power of directed fuzzing. The purpose of the exploration phase in AFLGo is to ensure a variety of seeds exist in the corpus before going to exploitation: this is ensured by our large starting seed corpus. We do keep the default setting of AFLGo for comparison: note that as AFLGo's default is to switch to exploitation at 10 minutes, "AFLGo" is effectively "full exploration". We expect it to perform similarly to

Table 1. 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

AFL. For WindRanger, we only use the "full exploitation" setting (i.e., exploitation starts at 0 min), again to evaluate the effectiveness of the directed fuzzing. We keep the default setup for TuneFuzz.

Fuzzer Options. In the complete evaluation, we run fuzzers without any memory limits for fuzzing programs (*-m none* for AFL-based fuzzers and *-rss_limit_mb=0* for libFuzzer). We set the timeout of each run of test input to be 10 seconds (*-t 10000* for AFL-based fuzzers and *-timeout=10* for libFuzzer). We will additionally add two more flags to the default settings in Magma for AFL-based fuzzers. We set *AFL_SKIP_CRASHES=1*, which tolerates crashing inputs in the initial seed corpus. Crashing inputs may occur in our seed corpus because we generate an initial corpus with a 24-hour-long run of AFL++. We set *AFL_FAST_CAL=1*, which speeds up the calibration to halve the time that the saturated initial corpus needs to be loaded. *AFL_FAST_CAL=1* does this by reducing the number of times inputs are run during calibration from the default of 8 times to 3 times.

We have different settings for AFL++ as it supports an additional feature, *cmplog*, which enables logging of comparison operands in a shared memory [4]. We set *AFL_CMPLOG_ONLY_NEW=1*, which only performs the *cmplog* feature for newly found inputs, but not the ones in the initial corpus [3]. The *cmplog* feature is based on RedQueen [7], and imposes additional instrumentation overhead, at the advantage of getting potential values for mutation that can help get through difficult branches. Doing this only for new inputs makes sense, as this advantage has already been gained for the previous inputs in the 24-hour-long AFL++ seed corpus creation run. Note that AFL, AFLGo, WindRanger, and FFD do not support *AFL_CMPLOG_ONLY_NEW*. The two flags settings *AFL_FAST_CAL=1* and *AFL_CMPLOG_ONLY_NEW=1* are recommended in the documentation of AFL++ for continuous fuzzing [3]. We also compile benchmarks with this feature and fuzz the *cmplog* binaries, which is suggested in the documentation of continuous fuzzing with AFL++ [3].

Fuzz Targets. For each benchmark, we choose to run a single fuzz target and patch pair. We use the fuzz targets provided by Magma, as prior work does [25]. 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. In our preliminary study [20], we selected 5 fuzz targets manually, based on whether the changed code would affect the APIs being called in the fuzz targets. Identifying relevant fuzz targets requires nontrivial efforts. Therefore, to scalably increase our benchmark set from 5 to 50, we adopt the checksum-based method from Klooster et al. [25]. In this approach, we calculate the checksum of the fuzz target before and after applying the buggy patch to decide whether the target is affected by the changes. By comparing the checksum values, we find that eight fuzz targets for libtiff, openssl, and sqlite3 libraries remain the same before and after applying certain buggy

patches. We therefore exclude these, randomly choosing benchmarks from the remaining set to get to our 50. The 50 benchmarks are listed in full in Table 2.

Table 2. Details of benchmarks used in our evaluation. Bug IDs refer to Magma [19] bug IDs including project and index; we simplify them to only their index in the benchmark name.

Benchmark Name	Library	Bug ID	Fuzz Target
libpng_4_1	libpng	PNG004	libpng_read_fuzzer
libsndfile_2_1	libsndfile	SND002	sndfile_fuzzer
libsndfile_7_1	libsndfile	SND007	sndfile_fuzzer
libsndfile_15_1	libsndfile	SND015	sndfile_fuzzer
libtiff_6_1	libtiff	TIF006	tiff_read_rgba_fuzzer
libtiff_7_1	libtiff	TIF007	tiff_read_rgba_fuzzer
libtiff_9_1	libtiff	TIF009	tiff_read_rgba_fuzzer
libtiff_10_1	libtiff	TIF010	tiff_read_rgba_fuzzer
libxml2_1_2	libxml2	XML001	libxml2_xml_read_memory_fuzzer
libxml2_2_1	libxml2	XML002	libxml2_xml_reader_for_file_fuzzer
libxml2_8_1	libxml2	XML008	libxml2_xml_reader_for_file_fuzzer
libxml2_10_2	libxml2	XML010	libxml2_xml_read_memory_fuzzer
libxml2_12_2	libxml2	XML012	libxml2_xml_read_memory_fuzzer
libxml2_14_1	libxml2	XML014	libxml2_xml_reader_for_file_fuzzer
libxml2_15_1	libxml2	XML015	libxml2_xml_reader_for_file_fuzzer
libxml2_16_1	libxml2	XML016	libxml2_xml_reader_for_file_fuzzer
libxml2_16_2	libxml2	XML016	libxml2_xml_read_memory_fuzzer
openssl_1_3	openssl	SSL001	bignum
openssl_1_5	openssl	SSL001	client
openssl_4_4	openssl	SSL004	server
openssl_5_1	openssl	SSL005	asn1
openssl_6_4	openssl	SSL006	server
openssl_6_5	openssl	SSL006	client
openssl_6_6	openssl	SSL006	x509
openssl_7_2	openssl	SSL007	asn1parse
openssl_7_4	openssl	SSL007	server
openssl_8_1	openssl	SSL008	asn1
openssl_9_1	openssl	SSL009	asn1
openssl_10_5	openssl	SSL010	client
openssl_11_4	openssl	SSL011	server
openssl_11_6	openssl	SSL011	x509
openssl_12_6	openssl	SSL012	x509
openssl_13_2	openssl	SSL013	asn1parse
openssl_16_6	openssl	SSL016	x509
openssl_17_2	openssl	SSL017	asn1parse
openssl_17_4	openssl	SSL017	server

Continued on next page

Benchmark	Library	Bug ID	Fuzz Target
openssl_18_5	openssl	SSL018	client
openssl_19_1	openssl	SSL019	asn1
openssl_20_3	openssl	SSL020	bignum
openssl_20_4	openssl	SSL020	server
php_4_1	php	PHP004	json
php_6_2	php	PHP005	unserialize
php_11_2	php	PHP011	exif
php_15_2	php	PHP015	exif
php_16_3	php	PHP016	unserialize
poppler_3_1	poppler	PDF003	pdf_fuzzer
poppler_9_1	poppler	PDF009	pdf_fuzzer
poppler_17_1	poppler	PDF017	pdf_fuzzer
sqlite3_18_1	sqlite3	SQL018	sqlite3_fuzz
sqlite3_20_1	sqlite3	SQL020	sqlite3_fuzz

3.4 Sensitivity Experiments

As our "commits", we use the Magma-provided patches consisting exactly of the code necessary to add a bug to the program under test. This may unfairly advantage directed fuzzers, as the targets we provide are exactly those lines that insert a bug. In reality, commits may include a new bug as well as other code. So, we want to study whether directed fuzzers can still reach and detect the inserted bugs when the user-provided-targets contain not only the lines that insert bugs but also other lines of code.

AFLGo generates 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 distances or other feedback necessary for running the directed fuzzers. Thus, by inserting additional target lines of code (of the format "filename: line number of the target code") into this target file, we can evaluate the sensitivity of the directed fuzzers to non-bug-related lines in the target file.

But which additional lines of code should we add to the target files, and how many should we add? We originally considered randomly selecting lines of code and adding a certain number of these, as outlined in our preliminary study [20]. However, we decided the best way to simulate a realistic commit is simply to take lines of code modified in an existing commit as our extra lines of code. We found that the average number of lines of code added for the last 10 commits in each target library is not always higher than the average number of lines of code added in the Magma patch files. For example, for *libsndfile*, the mean number of lines of code added in Magma patch files is around 7, while the mean number of lines of code added in the last 10 commits is around 4. Similarly, the mean values of LoC added in both patch files and the last 10 commits for *libxml2* are the same. Thus, adding lines of code from a real commit keeps us at a reasonable number of user-provided targets.

For each injected bug, we randomly pick one commit from the most recent 10 commits and set the added lines of code in the commit as additional code targets. Note we still keep the Magma bug patch lines in the targets. We then evaluate the directed fuzzers with the injected bug and a target file consisting of the commit-touched lines of code *and* the original bug patch lines of code. We evaluate this for each bug that at least one directed fuzzer reaches/triggers in the main evaluation.

3.5 Implementation

We conduct all experiments in the Magma [19] fuzzing platform. In this section, we describe the key changes we have implemented in Magma in preparation for our experiments, and other observations about Magma internals that affect the analysis of our results. We use Magma v1.2 as our base version. Our codebase is distributed as open-source at https://github.com/carolemieux/cicd-magma.

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 avoid 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 the corpus for each experiment: Magma only allows a fixed path to the corpus. In this way, users can better customize each fuzzing campaign by changing the configuration files.

Integration of Fuzzers. We integrated four directed fuzzers, AFLGo, FFD, WindRanger, and TuneFuzz into Magma. While the organization of scripts in Magma separates each building block of fuzzers neatly, we spent a 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. [37] suggest, incompatibilities between the fuzzing environment and project, issues in build tools or external dependencies, as well as issues in the corpus, can lead to build failures. In fact, this is not the first time that someone tried to integrate AFLGo into Magma: a 2021 issue was opened to discuss this matter [48]. However, the issue of supporting directed fuzzers in Magma is still open as of the time we wrote this paper. The build errors when compiling libraries with AFLGo were still unresolved after more than three years since the issue was first brought up. In our work, we were able to fix all the build errors when building the targets with AFLGo, FFD, and TuneFuzz. WindRanger segfaulted when building PHP; since our 6 other fuzzers had no problem building this benchmark, we assume this is some issue in the additional WindRanger instrumentation.

Automation of Generating Target Program Locations. Previous attempts to integrate AFLGo into Magma [48] retrieve 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.

Coverage Evaluation Pipeline. After fuzzing sessions, we rebuild benchmarks and gather the code coverage information via *llvm_cov*. We first compile benchmarks using the *-fprofile-instr-generate* and *-fcoverage-mapping* flags for coverage instrumentation. Then we run instrumented programs on all inputs saved in the queue during fuzzing, as these are "interesting" inputs that either discover new coverage or have certain features, such as exercising basic blocks that are closer to the target basic blocks. Finally, we use *llvm_cov* to generate the coverage reports for the total coverage of the program and the coverage of the files with the target lines of code.

Magma Off-by-POLL Error. While investigating our results in Section 4.5.2, we noticed that Magma bug reporting times were off by `POLL` (in our case, we used the default of 5 seconds) in the version of Magma we were using. This bug meant that the file saying that a bug was reached at 10 seconds actually was recording that a bug was reached at 5 seconds. We were able to root cause the issue and issued a pull request to Magma [27], but at this point, we had already

run our experiments. As the issue causes *all* bug reached/triggered times in our experiments to be `POLL` seconds later than they actually were, we subtract `POLL` (5 seconds) from all the results presented in Section 4.

Magma POLL Delay in Reporting Triggers We noticed another unusual timing behaviour while analyzing results for Section 4.5.2, but this issue is much more complicated than the above. The main consequence of this behaviour is that if Magma reports a bug is first triggered at time $t + \texttt{POLL}$, it is possible it was triggered in time $(t - \texttt{POLL}, t + \texttt{POLL}]$ rather than strictly in time $(t, t + \texttt{POLL}]$. We use this fact in our analysis in Section 4.5.2.

For those interested in why the behaviour exists, the reason for it is as follows. Magma uses a canary system to report bug reach/trigger times. The Magma logging inserted in the program under tests writes to the canary file when a bug is reached/triggered, and this same canary file is read by the monitor every `POLL` seconds.

In detail, the canary file consists of a `consumed` boolean, as well as a producer array and a consumer array. The program under test logging writes to the producer array [11]. In the default `–fetch file` mode, the monitor reads from the consumer array [12]. After reading the consumer array, the monitor sets `consumed` to `true`. While the logging invoked by the program under test always updates the producer array, it only copies over the producer array to the consumer array *when the* `consumed` *boolean* is true (and immediately sets `consumed` to false) [11]. While this may prevent race conditions, this means there is a lag larger than `POLL` between the time that a bug has actually been reached/triggered and what the monitor reads.

This design impacts results if a bug is reached by an input and then triggered by a *later* input *in the same poll interval*. Say the monitor has just consumed the canary file at time t ($t = n \cdot \texttt{POLL}$ for some n), and the bug of interest has not been reached/triggered at t. The monitor sets `consumed` to true after this poll. Say at $t + \delta$, where $t + \delta < t + \texttt{POLL}$, the bug is reached, but not triggered. The canary logging adds this information to the producer array, and as `consumed` is true, copies the producer array (which says the bug is reached, but not triggered) to the consumer array. It then sets `consumed` to false. Now, say at $t + \delta'$ where $t + \delta < t + \delta' < t + \texttt{POLL}$, the bug is reached *and* triggered. The program under test logging adds this information to the producer array, but as `consumed` is false, it does not update the consumer array. Now at time $t + \texttt{POLL}$, the monitor reads the consumer array, which says the bug is reached, but not triggered. As long as sometime from $t + \texttt{POLL}$ to $t + 2 \cdot \texttt{POLL}$ the program under test hits the canary logging again, the consumer array will be updated to reflect that the bug has been triggered. This means that looking at the Magma results, we would see that at poll $t + \texttt{POLL}$ the bug is reached but not triggered, and only at poll $t + 2 \cdot \texttt{POLL}$ the bug is triggered—even though the bug was triggered before $t + \texttt{POLL}$. This is a surprising delay. Bug triggering could be missed entirely in the (quite unlikely) event that nothing is ever logged to the canary file after the bug is triggered.

Unlike the Off-by-POLL error above, we cannot definitively adjust our results to account for this delay. Suppose we have a bug reaching time of 5s and a triggering time of 10s. Due to the above delay, it is possible that the triggering time is ≤5s. But it is possible that the first log event after 5s was the first time the bug was triggered; in this case, the triggering time would be 5s$< t \leq$10s, in alignment with the poll reporting times. As this is a consistent delay for all fuzzers, it should not have a large effect on our comparative analysis in the main results. But it does affect the analysis in Section 4.5.2 where we try to determine whether bugs are reached/triggered while loading seeds or while the main fuzzing loop occurs. For that analysis to remain accurate, we must make a more conservative assumption that a bug triggered at time t means it could actually have been triggered anytime in $(t - 2 \cdot \texttt{POLL}, t]$.

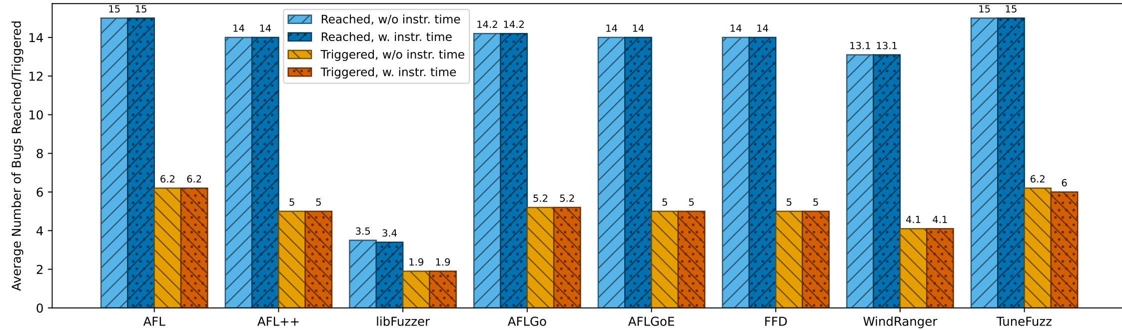

Fig. 1. Average number of bugs reached/triggered by fuzzers, over all benchmarks and repetitions. Including instrumentation time in the timeout, 1 libFuzzer iteration fails to reach a bug (libxml2_8_1, with reached time of 570s), and TuneFuzz fails to trigger two bugs, which were both triggered in only one iteration (libxml2_12_2 triggered in 530s, openssl_20_ triggered in 580s).

4 Results

In this section, we present the results of our full evaluation, and analyze these results to investigate the performance of directed and undirected fuzzers in our CI/CD setup. Our codebase and processed data can be found at https://github.com/carolemieux/cicd-magma. Our raw data can be found on Zenodo with DOI 10.5281/zenodo.17664060.

To run the main experiments, we use two machines with an AMD EPYC 7343 Processor, and 1 TB memory. Both machines have 64 logical cores and 32 physical cores. We run experiments only on physical cores to ensure the best performance. Compared to the machines in our preliminary experiments [20], this processor has a lower base clock (3.2 GHz vs 3.6 GHz) and boost clock (3.9 GHz vs 4.9 GHz) frequency, which may result in slightly slower fuzzing.

While we were able to integrate WindRanger into Magma for most benchmarks, it failed to build PHP, with a segmentation fault occurring during the build. As all other fuzzers were able to build PHP without segmentation faults, we note this as a compilation failure in our results.

4.1 Reaching Code Change Locations

Recall our first research question:

> ***RQ1.*** *Are directed or undirected fuzzers more efficient at* ***reaching*** *code change locations in a CI/CD setting?*

First, let us consider the summary results of the *number of bugs* reached by all fuzzers within the 10-minute timeout. Figure 1 shows the average number of bugs reached and triggered, across all 10 iterations and 50 benchmarks, for each fuzzer. We include the results with and without instrumentation time. Across all 50 benchmarks, fuzzers were only able to reach/trigger bugs on 15 benchmarks in total.

From Figure 1, we can spot roughly 4 equivalence classes of fuzzers. (1) AFL and TuneFuzz both reach 15 bugs on average, consistently. (2) AFLGo reaches 14.2 bugs and WindRanger 13.1 bugs[3]—both reach the bug sqlite3_20_1 in only 2 and 1 iterations, rather than all 10 iterations as AFL and TuneFuzz do. (3) Then, AFL++, AFLGoE, and FFD all reach 14 bugs, failing to reach sqlite3_20_1. (4) Finally, libFuzzer only reaches 3.5 bugs on average.

As seen in Figure 1, including instrumentation time barely changes the reachability results. Only one iteration of libFuzzer, which reached libxml2_8_1's bug in 570s, is pushed over the 600s timeout.

[3]Recall WindRanger fails to compile PHP. If we count this as a failure, this puts WindRanger just above libFuzzer. If we assume the build failure can be fixed, as php_11_2 is reached by all fuzzers within the first 5 seconds of fuzzing, it is fair to assume WindRanger would reach it too.

Table 3. The mean survival time in seconds of the bugs reached (R) and triggered (T), without instrumentation time. AFLGo runs exploration more from the start; AFLGoE runs exploitation mode from the start. Empty entries indicate the fuzzer fails to reach/trigger bug; × indicates the fuzzer fails to compile benchmark. Fastest non-tied values highlighted in grey. Bugs not reached/triggered in all 10 iterations wavily underlined.

	AFL		AFL++		libFuzzer		AFLGo		AFLGoE		FFD		WindRanger		TuneFuzz	
	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
libpng_4_1	5		5				5		5		5		5		5	
libsndfile_7_1	5	20	5	15			5	30	5	30	5	50	5	35	5	46
libtiff_10_1	5		5				5		5		5		5		5	
libtiff_7_1	5	10	5	5	100	192	5	10	5	10	5	10	5	10	5	10
libxml2_12_2	5		5		5		5		5		5		5		5	593
libxml2_1_2	5	10	5	41			5	10	5	10	5	15	5	15	5	15
libxml2_8_1	9		12		483		8		5		5		5		10	
openssl_10_5	5		5				5		5		5		5		5	
openssl_16_6	5		5				5		5		5		5		5	
openssl_1_5	15		52				30		25		30		25		20	
openssl_20_4	5	590	5				5		5		5		5		5	598
php_11_2	5	10	5	10	5	10	5	10	5	10	5	10	×	×	5	10
poppler_9_1	12		10				20		20		25		25		24	
sqlite3_18_1	5	10	5	5			5	10	5	10	5	10	5	10	5	10
sqlite3_20_1	60	60					579	579					591	591	218	218

Table 4. Mean survival time in seconds of bugs reached (R) and triggered (T), with instrumentation time. AFLGoE runs exploitation mode from the start. Empty entries indicate fuzzer fails to reach/trigger bug; × indicates the fuzzer failed to compile benchmark; ** indicates a bug that is no longer reached/triggered within 600s when including instrumentation time. Fastest non-tied values highlighted in grey. Bugs not reached/triggered in all 10 iterations wavily underlined.

	AFL		AFL++		libFuzzer		AFLGo		AFLGoE		FFD		WindRanger		TuneFuzz	
	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
libpng_4_1	19		32				29		28		20		17		32	
libsndfile_7_1	73	88	131	141			104	129	109	134	72	118	62	92	115	156
libtiff_10_1	39		54				62		61		41		47		61	
libtiff_7_1	36	41	53	53	121	211	56	61	60	66	40	46	42	47	58	63
libxml2_12_2	40		53		34		73		79		42		62		95	**
libxml2_1_2	39	44	58	94			70	75	73	78	38	48	46	56	80	90
libxml2_8_1	46		59		497		80		65		43		52		88	
openssl_10_5	27		42				261		257		28		129		311	**
openssl_16_6	25		40				209		240		28		63		320	
openssl_1_5	36		88				265		259		48		112		330	
openssl_20_4	27	595	42				259		259		28		59		311	
php_11_2	111	116	157	162	80	85	142	147	129	134	125	130	×	×	385	390
poppler_9_1	60		76				182		187		69		99		271	
sqlite3_18_1	68	73	44	44			84	89	85	90	106	110	57	62	100	105
sqlite3_20_1	131	131					595	595					596	596	308	308

However, one may also be interested not only in the average number of bugs reached, but in the time to reach those bugs and the probability of reaching them. This information is illustrated in Tables 3 and 4, which show the mean survival time of each bug, and wavily underline those bugs that are not reached/triggered in all 10 iterations of the given fuzzer. Most of the bugs in our evaluation that are reached by fuzzers are reached in all 10 iterations; the only exceptions are libxml2_8_1 for libFuzzer and sqlite3_20_1 for AFLGo and TuneFuzz.

For our survival analysis, we adopt Magma's scripts, which implement the Kaplan-Meier estimator [21] to model the survival function of each inserted bug. Survival analysis allows us to take into consideration the right cut-off nature of the data; we do not know, if a fuzzer fails to find a bug within our timeout, whether the fuzzer would never find the bug, or whether it would have found it in a bit more time. The survival function plots how long each bug "survives" without being found. Unlike most medical applications of this analysis, a lower survival time is a better result for a fuzzer.

Note that Tables 3, 4, and 5 show only the mean survival time reported by Magma. The mean survival time reported by Magma is, in fact, the Restricted Mean Survival Time (RMST) up to the fuzzer trial length. Han et. al's explanation of RMST [18] states that RMST is "the area under the survival curve up to a specific time point" (in our case, up to 600s), which "can be interpreted as the average survival time" during the defined time period. This area under the curve interpretation explains why, for bugs that are only triggered in one run, quite late in the fuzzing runs (e.g., openssl_20_4, triggered by TuneFuzz in only one run, at 580s without instrumentation time), the mean survival time is near 600s (e.g., 598s for openssl_20_4 with TuneFuzz without instrumentation time)—the curve would have the bug surviving at 1 for nearly all the trial time. To compute the survival times in Table 4, we added the instrumentation time to the bug reaching/triggering time reported by Magma. As aforementioned, Magma only builds the target once for each 10 fuzzing runs: so, we add the same instrumentation time to all runs. And while Magma's monitors only collect bug triggering/reaching time in 5-second intervals, we have the instrumentation time in tenths-of-seconds precision. This means that small differences in Table 4 should be taken with a grain of salt.

Let us now analyze Tables 3 and 4. When we don't take instrumentation time into consideration (Table 3), we see only small differences in minimum reaching time: AFL++ reaches one bug 3 seconds faster than AFL; AFLGo reaches 1 bug 1 second faster than AFL; AFL reaches one bug 6 seconds faster than TuneFuzz. The major difference in reachability times is on sqlite3_20_1, where AFL takes 65 seconds compared to 200+ seconds for all other fuzzers.

The trends are clearer when we take instrumentation time into consideration (Table 4): AFL is almost consistently the fastest at reaching bugs. As aforementioned, small differences in Table 4 should be taken with a grain of salt; there are several benchmarks where AFL and FFD have very similar mean reaching times. However, when considering our earlier equivalence classes based on the number of bugs found, the analysis simplifies somewhat. When comparing AFL to TuneFuzz, we have a mean bug reaching time of 57s vs 196s: TuneFuzz is clearly slower. This makes sense when considering the trends in instrumentation time, illustrated in Figure 2. We see that TuneFuzz has the highest instrumentation times of all fuzzers we evaluated. While both mean bug reaching times (57s for AFL and 196s for TuneFuzz) fit within the 10-minute timeout, AFL appears to be a slightly better tool to quickly run through a seed corpus and find easy bugs in the program under test.

As for the other fuzzers, we see that AFLGo/AFLGoE has reasonably high instrumentation times (though lower than TuneFuzz), especially evident on the openssl benchmarks. FFD has similar instrumentation times as AFL, but reaches/triggers bugs more slowly than AFL on nearly all benchmarks. WindRanger and AFL++ are somewhere in between the fast instrumenters (AFL, FFD) and the slow instrumenters (AFLGo/AFLGoE and TuneFuzz).

In our evaluation, libFuzzer performs quite poorly compared to all other fuzzers. This is strange, given how consistently most of the bugs in Table 3 are reached by all other fuzzers. While conducting analysis of other fuzzer statistics

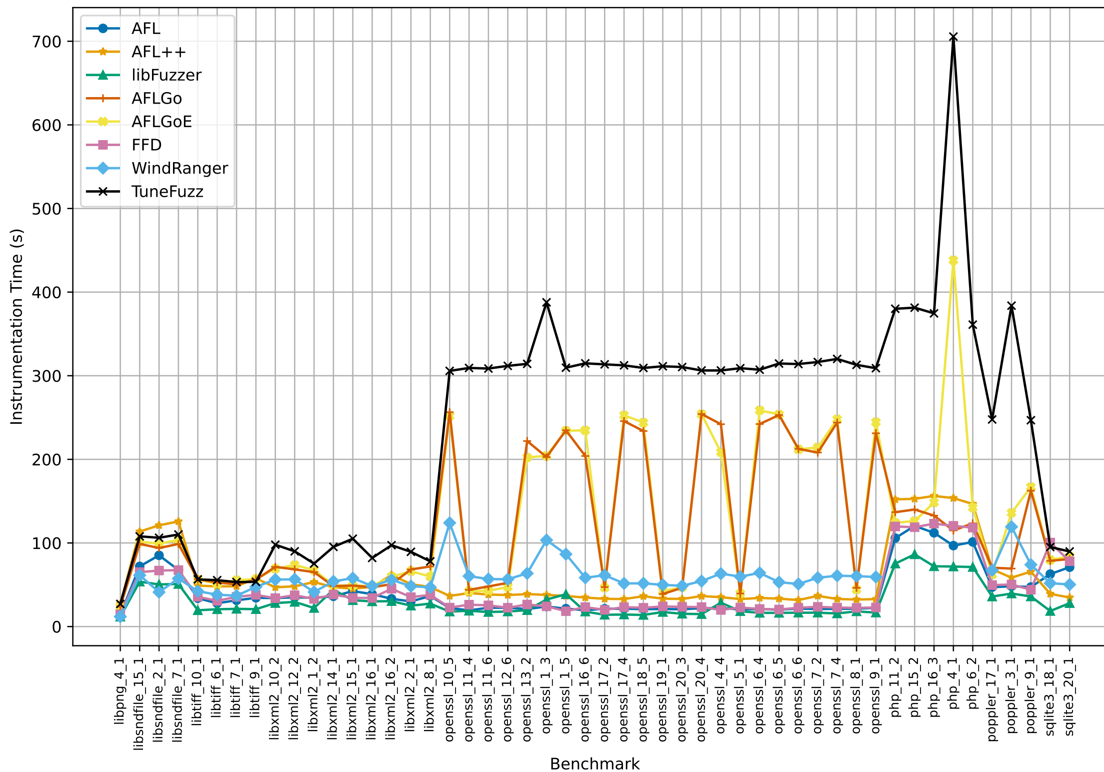

Fig. 2. Instrumentation time in seconds of full benchmark set. The build times for AFLGo and AFLGoE should be the same modulo noise in iterations; there appears to be an outlier on php_4_1, where AFLGoE is much slower than AFLGo.

(Section 4.5.2), we noticed that we inherited from Magma's configuration the libFuzzer option `-fork=1`. The libFuzzer documentation [41] states that in fork mode, "the top libFuzzer process [...] will spawn up to N concurrent child processes providing them small random subsets of the corpus". If the small random subsets of the corpus do not cover the whole close-to-saturated corpus, this may significantly disadvantage libFuzzer. It would mean libFuzzer spends time doing new exploration, which, in a 10-minute time budget, is much less likely to uncover interesting inputs than simply reading through our seed corpus.

RQ1: Directed vs. Undirected for Reaching Bugs

AFL and TuneFuzz reach the most bugs within the 10-minute timeout, more and more consistently than all other techniques. When taking instrumentation time into consideration, AFL is much faster at reaching those bugs (57s vs 196s). User-provided-target directed fuzzers (AFLGo, FFD, WindRanger) do not show advantages in the number of bugs reached or speed at reaching them compared to AFL. In our CI/CD simulation, an undirected fuzzer (AFL) is most efficient at reaching code change locations.

4.2 Triggering Bugs in the Code Change

Recall our second research question:

> ***RQ2.*** *Are directed or undirected fuzzers more efficient at* ***triggering*** *bugs in a code change in a CI/CD setting?*

Again, let us first consider the summary results of the *number of bugs* triggered by all fuzzers within the 10-minute timeout. By analyzing Figure 1, we again have our 4 equivalence classes. (1) AFL and TuneFuzz both trigger 6.2 bugs on average, with openssl_20_4 and libxml2_12_2 being triggered with low probability. (2) AFLGo triggers 5.2 and WindRanger 4.1 bugs[4]—again, it is sqlite3_20_1 that is triggered in only 2 and 1 iterations, rather than all 10 iterations as AFL and TuneFuzz do. (3) Then, AFL++, AFLGoE, and FFD all trigger 5 bugs, failing to trigger sqlite3_20_1. (4) Finally, libFuzzer only triggers 1.9 bugs on average.

When including instrumentation time, however, TuneFuzz fails to trigger openssl_20_4 and libxml2_12_2 within the 10-minute timeout. This gives a tiny advantage to AFL in terms of average bugs triggered in the evaluation.

This tiny advantage is bigger if we consider the mean bug triggered survival time, with instrumentation (Table 4). On the 6 benchmarks on which AFL and TuneFuzz both trigger bugs, AFL averages 87s to trigger the bugs, compared to TuneFuzz's 190s. This result is consistent with the result on bug reaching times from the last subsection.

Finally, bug triggering is done slightly less consistently than reaching bugs on our benchmarks. Recall mean survival times which include fewer than 10 iterations reaching/triggering the bug are wavily underlined in Tables 3 and 4. In particular, without instrumentation time: libtiff_7_1 is triggered in only 9 runs by libFuzzer; openssl_20_4 is triggered in only 2 runs by AFL and 1 by TuneFuzz; libxml2_12_2 is triggered only in 1 run by TuneFuzz, and sqlite3_20_1 is triggered only in 2 runs by AFLGo and WindRanger. Including instrumentation excludes the two low-probability TuneFuzz-triggered bugs.

> **RQ2: Directed vs. Undirected for Triggering Bugs**
>
> AFL and TuneFuzz *trigger* the most bugs within the 10-minute timeout. When considering instrumentation time, AFL slightly outperforms TuneFuzz and is also faster at triggering bugs. None of the user-provided-target directed fuzzers (AFLGo, FFD, WindRanger) outperform AFL in terms of the number of bugs triggered or the mean triggering time. In our CI/CD simulation, starting from a 24-hour-run corpus, an undirected fuzzer (AFL) is most efficient at triggering buggy behaviour at code change locations.

4.3 Setting up Fuzzers in CI/CD Fuzzing

As for our third research question:

> ***RQ3.*** *How easy is it to set up the directed and undirected fuzzers in CI/CD fuzzing?*

We answer this by relaying our experiences with setting up fuzzers in Magma. Consider this one case study in setting up fuzzers; Nourry et al.'s study on the human side of fuzz testing contains a much more thorough survey [37].

For each new fuzzer that we integrate into Magma, we examined the setup instructions in their source repositories and separated the code into different scripts that install dependencies, compile libraries, apply patches, and run the compiled binaries. Because some fuzzers are last maintained over 4 years before this study was conducted, the documentation was outdated, thus requiring debugging to find the correct set of configurations for fuzzing. Moreover, as different fuzzers

[4] Again, remember WindRanger fails to compile PHP. If the build failure can be fixed, as php_11_2 is triggered by all fuzzers within the first 10 seconds of fuzzing, it is fair to assume WindRanger would trigger it too.

have different runtime requirements, such as library versions, the way to compile libraries with the compilers provided by each fuzzer differs. Finally, we resolved all errors mentioned in an issue opened in 2021 for integrating AFLGo into Magma [48]. In addition, we tested the build of all benchmarks with newly added fuzzers: FFD, WindRanger, and TuneFuzz, and modified the scripts to be compatible with new and existing fuzzers in Magma. We note that WindRanger segfaulted when trying to build PHP; thus PHP is excluded from its experiments. We tried to integrate DAFL [23] into Magma but could not get the extra static analysis build required working in a reasonable time.

Our experiences are only a single set of experiences; it is possible that the WindRanger and DAFL issues can be solved with further debugging. But our experiences are in line with build failures being a very prominent topic in Nourry et al.'s study on challenges faced by humans when adopting and running fuzz testing [37]. We would expect to face similar build issues when actually incorporating fuzzers into a CI/CD pipeline, rather than into the Magma framework. Clearly documenting the build process and dependencies or common issues, beyond supporting build in a single Docker artifact, may help the adoption of a fuzzer in a CI/CD context.

Beyond build errors, running the experiments is easy: Magma provides configuration scripts *captainrc* and *configrc* to specify fuzzing settings and library-specific settings, such as fuzzing time and how to insert buggy patches.

The build time in seconds for each fuzzer is: libFuzzer (2s), AFL (8s), AFL++ (20s), FFD (21s), WindRanger (123s), AFLGo (229s), and TuneFuzz (507s). The build time of libFuzzer is negligible, because recent versions of Clang include libFuzzer by default. AFL++ has longer build and instrumentation time than its ancestor AFL as it supports additional features such as cmplog. FFD is built upon AFL but adds additional instrumentation to keep track of whether the input can exercise the target code locations. As a result, FFD has a slightly higher build time than AFL. The other directed fuzzers, AFLGo, WindRanger, and TuneFuzz, rely on much heavier instrumentation that requires static analysis, leading to a much longer build time. In practice, we expect that the fuzzers would be pre-compiled, so this should not affect the suitability of a fuzzer for CI/CD.

RQ3: Ease of Setting up Directed/Undirected Fuzzers in a CI/CD Setting

While we are able to set up 4 distinct directed fuzzers in Magma (AFLGo, FFD, WindRanger, TuneFuzz), the process of doing this required extensive build process debugging. We posit this is not easy for a non-fuzzing expert. Fuzzers that require additional build configurations (e.g., to run a program analysis separate from the main fuzzing) will require more effort to set up.

4.4 Sensitivity to Additional Target Locations

Finally we address our fourth research question:

> ***RQ4.*** *How sensitive are directed fuzzers to "bloat" in commits, i.e., can directed fuzzers still reach an injected bug if given additional target locations?*

As aforementioned, user-provided-target directed fuzzers like AFLGo require target code locations to run. We provide them with the target locations corresponding exactly to the patch lines added by Magma to insert each bug. This may not be a realistic set of targets; usually, we would expect a commit to not *solely* consist of a front-patched bug. Therefore, we conduct what we call our *sensitivity experiments* to see if directed fuzzers can still reach or trigger the injected bugs when given additional code locations as targets.

As mentioned in Section 3.4, to decide on the additional target code locations, we sample a random commit in the past 10 commits of each library. We then add the lines of code modified in the commit as additional target code locations

Table 5. Mean survival time of the bugs reached (R) and triggered (T) in the sensitivity evaluation, without instrumentation. Unit in seconds, rounded to the nearest second. Empty entries indicate that the fuzzer is unable to find inputs that can reach or trigger the injected bugs. Differences compared to main results are highlighted.

	AFLGo		AFLGoE		FFD		WindRanger	
	R	T	R	T	R	T	R	T
libpng_4_1	5		5		5		5	
libsndfile_7_1	5	30	5	30	5	50	5	35
libtiff_10_1	5		5		5		5	
libtiff_7_1	5	10	5	10	5	10	5	10
libxml2_12_2	5		5		5		5	
libxml2_1_2	5	12	5	15	5	20	5	15
libxml2_8_1	5		5		5		5	
openssl_10_5	5		5		5		5	
openssl_16_6	5		5		5		5	
openssl_1_5	25		30		30		25	
openssl_20_4	5		5		5	561	5	
php_11_2	5	10	5	10	5	10		
poppler_9_1	20		20		25		21	
sqlite3_18_1	5	10	5	10	5	10	5	10
sqlite3_20_1	578	578						

Table 6. Differences in mean reaching/triggering time, without instrumentation time, after adding additional lines to commits. All other bugs reached/triggered in the same mean time. Differences listed are in regular mean time to reach/trigger, not mean survival time to reach/trigger. WR stands for WindRanger, AGoE stands for AFLGoE. *p*-values calculated with Mann-Whitney-U test when the same number of iterations reach/trigger the bug.

Fuzzer	Benchmark	Difference explanation
AFLGo	libxml2_1_2	triggered 2.0s slower ($p = 0.017$)
AFLGo	libxml2_8_1	reached 3.0s faster ($p = 0.0025$)
AFLGo	openssl_1_5	reached 5.0s faster ($p = 8 \cdot 10^{-6}$)
AFLGo	sqlite3_20_1	reached 2.5s faster ($p = 0.31$)
AFLGo	sqlite3_20_1	triggered 2.5s faster ($p = 0.31$)
AGoE	libxml2_1_2	triggered 5.0s slower ($p = 8 \cdot 10^{-6}$)
AGoE	openssl_1_5	reached 5.0s slower ($p = 8 \cdot 10^{-6}$)
FFD	libxml2_1_2	triggered 5.0s slower ($p = 8 \cdot 10^{-6}$)
FFD	openssl_20_4	newly triggered (1 iteration)
WR	poppler_9_1	reached 4.0s faster ($p = 0.00022$)
WR	sqlite3_18_1	triggered 0.5s slower ($p = 0.18$)
WR	sqlite3_20_1	no longer reached
WR	sqlite3_20_1	no longer triggered

for directed fuzzing. We only consider user-provided-target directed fuzzers in sensitivity experiments because the other fuzzers do not require target code locations for fuzzing. This means we evaluate AFLGo, AFLGoE (exploit-only AFLGo), FFD, and WindRanger in these sensitivity experiments.

Table 5 reports the mean survival time (without instrumentation) of each bug reached/triggered in our sensitivity experiments—that is, with additional target locations added to the user-provided-target given to each directed fuzzer. For simplicity of analysis, we highlight differences in mean survival times compared to Table 3. Most reaching and triggering results are unchanged.

For the few changes, we document the changes in more detail in Table 6. This table provides an explanation of the observed difference in mean bug reaching/triggering time for each result that differs in Table 5. For those cases where the same number of iterations reach/trigger the bug before and after adding the additional lines of code, we conduct Mann-Whitney U-Tests to determine if the differences in reaching/triggering time are statistically significant. These p-values are also listed in Table 6. Most of the p-values are quite small, indicating that the bug reaching/triggering time does reliably differ when adding additional targets. So, we can conclude that adding additional targets does change bug reaching/triggering time for directed fuzzers on some bugs. However, the magnitude of these differences is quite small—at most 5 seconds of difference. These small differences are unlikely to be empirically relevant.

The most empirically relevant difference is that WindRanger no longer reaches/triggers sqlite3_20_1 in any iterations with additional targets inserted. This is in contrast with the results in Table 3, where WindRanger reaches/triggers the bug in 2 of 10 iterations. However, this is taking a quite unlikely event to an even less likely event, rather than completely blocking WindRanger from finding a bug it finds reliably.

As for instrumentation times, recall that, due to Magma's construction, we only measure instrumentation time once per fuzzer-benchmark pair. Adding instrumentation time to Table 5 and comparing it to Table 4 would likely yield small, noisy differences. However, we can examine if directed fuzzer instrumentation times change when adding

additional target locations. We will then analyze whether changes in instrumentation time outweigh the changes in time to reach/trigger bugs (Table 6).

Due to our lack of per-benchmark instrumentation repetitions, we consider the average instrumentation time across all benchmarks. For WindRanger, average instrumentation time across the 15 benchmarks in Table 5 goes from 57s to 56s; surprisingly little change. FFD also shows a small drop in average instrumentation time when adding additional targets, 46s to 43s. Both these changes are unlikely to be significant in practice. This suggests that the same minimal effect on bug reaching/triggering time we see in Tables 3 and 6 is likely to be preserved.

Recall that the difference between AFLGo and AFLGoE is in fuzzing runtime option changes; so the instrumentation times for AFLGoE should be a second iteration of AFLGo. This gives us a tiny bit more statistical power in our analysis. What we see is that average instrumentation time across all benchmarks, when adding additional targets, drops from 123s to 101s for AFLGo, and drops from 125s to 103s for AFLGoE. Upon further investigation, it appears that we can reproduce a similar drop in instrumentation time by *manually providing* the original set of target locations, rather than relying on the automated script to generate them from the git diff. It is not entirely clear why this is the case, because from the log files, the time saved is through the whole instrumentation process, not just the generation of the target file. Still, this suggests that users concerned with instrumentation time may want to pre-generate a target file rather than relying on the script to generate it from the git diff.

RQ4. Directed Fuzzers' Sensitivity to Bloat in Targets

We observe some statistically significant changes to bug reaching/triggering time by directed fuzzers when adding bloat in the user-provided target list, but none of these changes are likely to be empirically significant. Overall, in our CI/CD setting, which starts from a 24-hour-fuzzing corpus, bloat in target lists does not have a large impact on the results of directed fuzzers.

4.5 Other Factors Impacting the Fuzzing Efficiency

Above, we answered all the research questions from Section 1. Next, we analyze secondary fuzzing metrics—namely, branch coverage, execution counts, and true fuzzing time—to see whether any of these metrics explain our main results.

Table 7. Median branch coverage in percentage of the entire library during the 10-min fuzzing campaigns in the complete evaluation. × means the fuzzer failed to compile the target. Values that are statistically significantly higher than more than half the other fuzzers, and not tied to any other values, are underlined.

	AFL	AFL++	libFuzzer	AFLGo	AFLGoE	FFD	WindRanger	TuneFuzz
libpng_4_1	34.03	33.98	34.36	34.09	34.04	33.99	33.98	33.98
libsndfile_15_1	29.34	29.34	29.36	29.34	29.34	29.34	29.34	29.34
libsndfile_2_1	29.34	29.34	29.36	29.34	29.34	29.34	29.34	29.34
libsndfile_7_1	29.39	29.39	29.41	29.39	29.39	29.39	29.39	29.39
libtiff_10_1	44.85	44.85	45.05	44.85	44.85	44.85	44.85	44.85
libtiff_6_1	44.76	44.76	44.79	44.76	44.76	44.76	44.76	44.76
libtiff_7_1	44.80	44.81	44.90	44.80	44.80	44.80	44.80	44.80
libtiff_9_1	44.76	44.76	44.78	44.76	44.76	44.76	44.76	44.76
libxml2_10_2	20.33	20.34	20.34	20.33	20.33	20.34	20.35	20.38

Continued on next page

	AFL	AFL++	libFuzzer	AFLGo	AFLGoE	FFD	WindRanger	TuneFuzz
libxml2_1_2	20.35	20.35	20.36	20.35	20.35	20.35	20.35	20.40
libxml2_12_2	20.35	20.35	20.36	20.35	20.35	20.36	20.35	20.41
libxml2_14_1	22.08	22.08	22.08	22.07	22.07	22.08	22.09	22.09
libxml2_15_1	22.09	22.09	22.08	22.07	22.07	22.09	22.09	22.09
libxml2_16_1	22.08	22.09	22.08	22.07	22.07	22.09	22.08	22.09
libxml2_16_2	20.33	20.33	20.34	20.30	20.30	20.34	20.33	20.38
libxml2_2_1	22.08	22.08	22.08	22.08	22.09	22.08	22.09	22.10
libxml2_8_1	22.11	22.10	22.10	22.10	22.10	22.11	22.10	22.12
openssl_10_5	19.83	19.83	19.83	19.83	19.83	19.83	19.83	19.83
openssl_11_4	19.73	19.73	19.73	19.70	19.70	19.73	19.73	19.73
openssl_11_6	14.70	14.70	14.70	1.70	1.70	14.70	14.70	14.71
openssl_12_6	14.70	14.70	14.70	1.70	1.70	14.70	14.70	14.71
openssl_1_3	3.70	3.70	3.70	3.70	3.70	3.70	3.70	3.70
openssl_13_2	2.16	2.16	2.16	2.16	2.16	2.16	2.16	2.16
openssl_1_5	19.84	19.84	19.84	19.84	19.84	19.84	19.84	19.84
openssl_16_6	14.72	14.72	14.72	14.72	14.72	14.72	14.72	14.73
openssl_17_2	2.16	2.16	2.16	1.93	1.93	2.16	2.16	2.16
openssl_17_4	19.73	19.73	19.73	19.73	19.73	19.73	19.73	19.73
openssl_18_5	19.81	19.81	19.81	19.81	19.81	19.81	19.81	19.81
openssl_19_1	13.64	13.64	13.64	13.63	13.63	13.64	13.64	13.64
openssl_20_3	3.70	3.70	3.70	3.69	3.69	3.70	3.70	3.70
openssl_20_4	19.74	19.74	19.74	19.74	19.74	19.74	19.74	19.74
openssl_4_4	19.73	19.73	19.73	19.73	19.73	19.73	19.73	19.73
openssl_5_1	13.64	13.64	13.64	13.62	13.62	13.64	13.64	13.64
openssl_6_4	19.73	19.73	19.73	19.73	19.73	19.73	19.73	19.73
openssl_6_5	19.81	19.81	19.81	19.81	19.81	19.81	19.81	19.81
openssl_6_6	14.70	14.70	14.70	14.70	14.70	14.70	14.70	14.71
openssl_7_2	2.16	2.16	2.16	2.16	2.16	2.16	2.16	2.16
openssl_7_4	19.73	19.73	19.73	19.73	19.73	19.73	19.73	19.73
openssl_8_1	13.64	13.64	13.64	13.63	13.63	13.64	13.64	13.64
openssl_9_1	13.64	13.64	13.64	13.64	13.64	13.64	13.64	13.64
php_11_2	2.23	2.23	2.23	2.23	2.23	2.23	×	2.23
php_15_2	2.23	2.23	2.23	1.52	1.52	2.23	×	2.23
php_16_3	2.36	2.35	2.36	1.42	1.42	2.35	×	2.36
php_4_1	2.00	1.99	2.00	1.52	1.52	1.99	×	2.00
php_6_2	2.23	2.23	2.23	1.52	1.52	2.23	×	2.23
poppler_17_1	38.93	38.93	38.93	38.93	38.93	38.93	38.93	38.93
poppler_3_1	38.24	38.93	38.93	38.93	38.93	38.24	38.93	38.93
poppler_9_1	38.95	38.95	38.95	38.95	38.95	38.95	38.95	38.95
sqlite3_18_1	56.74	56.74	56.74	56.74	56.74	56.74	56.74	56.74
sqlite3_20_1	57.28	57.28	57.28	57.28	57.28	57.28	57.28	57.28

4.5.1 Coverage Table 7 shows the median branch coverage of the entire library during all fuzzing campaigns for each fuzzer. The coverage value underlined is statistically greater than most other coverage values, and statistically greater than at least half of them. For example, if the coverage for TuneFuzz is statistically greater than that of four fuzzers while the coverage for other fuzzers is statistically greater than only one to two fuzzers, we underline only the coverage for TuneFuzz. We do not underline any numbers if there is a tie for the highest coverage value.

TuneFuzz has the highest median branch coverage for the greatest number of benchmarks (13), followed by libFuzzer (8). The broad program exploration of TuneFuzz is not surprising, as its auto-targets are all sanitizer-inserted code: the TuneFuzz paper showed broad coverage increases [52]. TuneFuzz is primarily advantageous in coverage on libxml and openssl. Further, TuneFuzz is in our best equivalence class of fuzzers at reaching and triggering injected bugs in Figure 1, tied with AFL when not considering instrumentation time.

However, there does not seem to be a clear relationship between branch coverage and bug-finding capability. Despite the same mean number of bugs reached and triggered as TuneFuzz, AFL does not have any coverage values statistically greater than many others. libFuzzer, which has the highest branch coverage for benchmarks composed from libpng, libsndfile, and libtiff libraries, only triggers the bug in libtiff_7_1 as shown in Table 3. It is possible that libFuzzer has higher coverage on these benchmarks because the initial corpus is created by running AFL++. As libFuzzer has slightly different mutations than the other AFL-family fuzzers, some of these mutations may uncover coverage that the AFL-family mutations do not. Also, since libFuzzer does not load the whole seed corpus with the `-fork=1` setting, it necessarily spends more time exploring.

Interestingly, the branch coverage for most benchmarks is almost identical for all fuzzers. AFLGo and AFLGoE have relatively lower branch coverage than the other fuzzers. The low coverages on openssl_11_6 and openssl_12_6 are the most stark, with 1.7% branch coverage instead of 14.7% branch coverage. Coverage for a directed fuzzer is not the main goal, so this tendency in itself is not necessarily a problem, but it is a clear distinction.

In Table 8, we report the median line coverage of the target function(s) for each benchmark. We retrieve the target function(s) by looking at the lines modified in the Magma patch, finding the function they are contained in, and filtering the coverage report to those functions. We use line coverage as some target functions had no branches, so their branch coverage percentage could not be recorded.

This table contains only 20 benchmarks, because for the 30 other benchmarks, the median line coverage of the target function(s) is zero. For openssl_20_3 and poppler_3_1, we are unable to retrieve the line coverage of any target function;[5] we denote this as '-' instead of omitting them from the table altogether.

The large number of benchmarks on which the fuzzer reaches zero line coverage on the target function(s) means that the target function(s) is never called. It is not possible to say whether it is callable or not without further analysis. But given that only 3 benchmarks have non-zero line coverage but no bug reached (vs 30 with 0 line coverage), it seems quite possible the target function is not callable on those benchmarks. This suggests the checksum filter for benchmarks may not be strong enough in practice to find good (bug, fuzz target) pairs.

Beyond the zeroes, it is surprising how identical the coverage reports are across all fuzzers. We sanity-checked the scripts and found that the `llvm-cov` command used to generate the target function line coverage operates on the same coverage profile data that produced the report in Table 7; the only differences are the given target file names to filter coverage for and the `-show-functions` argument. So, the coverage profile data does differ over the whole program (Table 7), but it is not reported to differ on the target function(s). The only visible difference is for php_6_2, where both

[5]The coverage report filtered to files containing any target lines is empty for openssl_20_3. For poppler_3_1, the coverage report only reports coverage on the file where class state is modified for Magma logging purposes, not on any of the files containing the actual logging functions.

Table 8. Median line coverage of the target function(s) in percentage. For all other benchmarks, the median line coverage of the target function(s) is zero. We report line coverage as some target function(s) lack branches. '-' means we could not retrieve the coverage for target functions; × means that the benchmark could not be built for the fuzzer; † means that no bugs were reached/triggered by any fuzzer for this benchmark. Surprisingly, the coverage numbers are nearly identical across fuzzers, except for AFLGo/AFLGOE on php_6_2.

	AFL	AFL++	libFuzzer	AFLGo	AFLGoE	FFD	WindRanger	TuneFuzz
libpng_4_1	38.37	38.37	38.37	38.37	38.37	38.37	38.37	38.37
libsndfile_7_1	95.83	95.83	95.83	95.83	95.83	95.83	95.83	95.83
libtiff_10_1	76.54	76.54	76.54	76.54	76.54	76.54	76.54	76.54
libtiff_7_1	89.69	89.69	89.69	89.69	89.69	89.69	89.69	89.69
libxml2_10_2†	65.36	65.36	65.36	65.36	65.36	65.36	65.36	65.36
libxml2_1_2	86.17	86.17	86.17	86.17	86.17	86.17	86.17	86.17
libxml2_12_2	96.23	96.23	96.23	96.23	96.23	96.23	96.23	96.23
libxml2_2_1 †	45.20	45.20	45.20	45.20	45.20	45.20	45.20	45.20
libxml2_8_1	83.90	83.90	83.90	83.90	83.90	83.90	83.90	83.90
openssl_10_5	1.63	1.63	1.63	1.63	1.63	1.63	1.63	1.63
openssl_1_5	75.47	75.47	75.47	75.47	75.47	75.47	75.47	75.47
openssl_16_6	91.59	91.59	91.59	91.59	91.59	91.59	91.59	91.59
openssl_20_3†	-	-	-	-	-	-	-	-
openssl_20_4	56.50	56.50	56.50	56.50	56.50	56.50	56.50	56.50
php_11_2	100.00	100.00	100.00	100.00	100.00	100.00	×	100.00
php_6_2†	10.96	10.96	10.96	0.00	0.00	10.96	×	10.96
poppler_3_1†	-	-	-	-	-	-	-	-
poppler_9_1	92.86	92.86	92.86	92.86	92.86	92.86	92.86	92.86
sqlite3_18_1	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
sqlite3_20_1	70.00	70.00	70.00	70.00	70.00	70.00	70.00	70.00

AFLGo variants have 0 coverage of the target function(s). This is consistent with the overall project coverage on these benchmarks (Table 7). But this is unexpected for AFLGoE, which should be generating inputs going towards the target lines for the whole 10 minutes of fuzzing. It is possible that the directed feedback heuristics are leading AFLGo astray on this particular benchmark.

Particularly surprising is that libFuzzer achieves the same coverage, even on benchmarks where it does not reach the same bugs, as all other fuzzers do. This may be because our coverage collecting methodology runs the whole set of inputs in the corpus through `llvm-cov`. As previously discussed, libFuzzer's `-fork=1` option may cause it to not run through the whole seed corpus while fuzzing. This would explain the discrepancy between not reaching bugs during fuzzing, but having the same target function coverage in our secondary analysis.

Coverage Differences

libFuzzer and TuneFuzz have significantly higher project coverage than other fuzzers on some benchmarks. Overall, higher project coverage does not seem to explain faster times to reach/trigger the bugs in our evaluation. Interestingly, target function coverage is nearly identical over all fuzzers, and 0 for the grand majority of benchmarks we do not reach/trigger bugs on.

4.5.2 Execution Counts and Time Fuzzing We noticed discrepancies in the number of executions conducted by the different fuzzers, as well the *true fuzzing* time. By true fuzzing, we mean time spent mutating and running new inputs,

rather than simply loading the seed corpus. We analyze differences in these metrics to better understand what may lead to the differences in performance we witness in our main evaluation. Table 9 shows the results.

For all AFL-family fuzzers, the number of executions conducted can be collected from `fuzzer_stats`, in the `execs_-done` field. We compute actual fuzzing time differently for AFL-family (AFL, AFLGo, AFLGoE, FFD, WindRanger) vs AFL++-family (AFL++, TuneFuzz) fuzzers. For AFL++-family fuzzers, we simply collect the newly-added `run_time` field from `fuzzer_stats`. For AFL-family fuzzers, we take the starting time for actual fuzzing as the first entry in `plot_data`, as this file is first written to after going through the seed corpus. We take the end time as the field `last_updated` in `fuzzer_stats`, as `plot_data` is not always updated within the time budget if findings have stalled. Thus for AFL, AFLGo, AFLGoE, FFD, and WindRanger, the time shown in the right-hand columns of Table 9 is `last_updated` in `fuzzer_stats` minus the first time in `plot_data`.

We exclude libFuzzer from this analysis as it is not clear how to reliably get these metrics. We can estimate the number of executions conducted by looking at the last entry in the libFuzzer logs, but this may miss a tail of executions. Being in-process, libFuzzer generally has many more total executions than AFL-family fuzzers. For instance: 95M on libpng_4_1 (vs 10M for AFL), 5M for libxml2_10_2 (vs 0.2M for AFL), 0.4M for openssl_10_5 (vs 0.3M for AFL), and 0.3M for poppler_17_1 (vs 27k for AFL).

More importantly, libFuzzer does not have a "true fuzzing time" comparable to the ones collected with the above process. This is because we inherited from Magma's configuration the option `-fork=1`. In this mode, libFuzzer gives child fuzzing processes "small random subsets of the corpus" to fuzz [41], rather than running through the whole seed corpus at the start. This means that from the logs, it appears that nearly all time is spent in actual fuzzing; the first entry in the libFuzzer logs is reliably at 1-3s. But reading corpus elements may be interspersed with true fuzzing time when new child processes are created.

Table 9. Median total executions conducted during the 600s-fuzzing sessions, rounded to millions (M) or thousands (k). The median true fuzz time is the portion of the 600s spent fuzzing (i.e., mutating and running new inputs), rather than loading the seed inputs, rounded to the nearest second. Values that are significantly lower are underlined. In both sides of the table, AGo=AFLGo, AGoE=AFLGoE, WR=WindRanger, TF=TuneFuzz.

	Median Executions (millions **M**, thousands **k**)							Median True Fuzz Time (**s**)						
	AFL	AFL++	AGo	AGoE	FFD	WR	TF	AFL	AFL++	AGo	AGoE	FFD	WR	TF
libpng_4_1	11M	1M	10M	11M	2M	6M	16M	599	595	599	599	600	599	599
libsndfile_15_1	0.5M	0.5M	0.5M	0.5M	0.3M	0.4M	0.2M	582	541	566	566	547	560	546
libsndfile_2_1	0.5M	0.4M	0.5M	0.4M	0.3M	0.4M	0.2M	582	544	567	567	548	562	547
libsndfile_7_1	0.5M	0.4M	0.5M	0.5M	0.3M	0.5M	0.2M	581	543	568	568	548	563	547
libtiff_10_1	6M	0.4M	5M	5M	2M	3M	1M	549	528	515	516	537	528	497
libtiff_6_1	7M	0.6M	6M	5M	2M	3M	0.2M	550	520	516	517	536	530	507
libtiff_7_1	6M	0.6M	5M	5M	2M	3M	0.2M	547	499	517	520	535	528	507
libtiff_9_1	6M	0.3M	6M	5M	2M	3M	0.3M	549	528	515	514	535	529	505
libxml2_10_2	0.1M	0.6M	82k	82k	1M	1M	2M	573	530	554	555	528	552	541
libxml2_12_2	0.1M	0.6M	0.1M	0.1M	0.4M	77k	2M	574	534	557	557	527	551	541
libxml2_14_1	2M	0.4M	1M	1M	1M	1M	0.9M	575	419	579	578	527	540	518
libxml2_15_1	1M	0.3M	1M	1M	1M	1M	1M	570	398	575	576	524	540	514
libxml2_16_1	2M	0.3M	1M	1M	1M	1M	0.8M	558	401	565	563	517	539	509

Continued on next page

	Median Executions (millions **M**, thousands **k**)							Median True Fuzz Time (**s**)						
	AFL	AFL++	AGo	AGoE	FFD	WR	TF	AFL	AFL++	AGo	AGoE	FFD	WR	TF
libxml2_16_2	0.1M	0.6M	9M	9M	1M	83k	2M	573	533	578	578	528	549	538
libxml2_1_2	0.1M	0.5M	85k	93k	0.6M	77k	2M	576	530	560	560	530	552	542
libxml2_2_1	1M	0.4M	2M	2M	1M	2M	0.9M	572	410	549	551	528	537	519
libxml2_8_1	1M	0.4M	2M	2M	1M	1M	1M	566	407	545	548	525	535	515
openssl_10_5	0.3M	94k	0.1M	0.1M	0.1M	0.2M	0.2M	556	471	493	495	510	499	489
openssl_11_4	83k	42k	0.1M	0.1M	24k	41k	38k	523	460	499	499	453	417	444
openssl_11_6	0.6M	0.3M	13M	13M	0.2M	0.6M	5M	592	571	592	592	576	584	586
openssl_12_6	0.8M	0.4M	12M	12M	0.3M	0.5M	5M	592	571	592	592	576	584	586
openssl_13_2	0.8M	0.5M	0.5M	0.5M	0.5M	0.5M	9M	600	596	599	599	599	599	598
openssl_16_6	0.7M	0.3M	0.4M	0.5M	0.4M	0.5M	5M	592	572	583	582	577	584	586
openssl_17_2	0.8M	0.5M	13M	13M	0.6M	0.5M	9M	600	596	600	599	599	599	598
openssl_17_4	82k	44k	36k	36k	25k	41k	38k	523	463	399	397	449	421	443
openssl_18_5	0.3M	98k	0.1M	0.1M	98k	0.2M	0.2M	556	490	494	495	509	489	505
openssl_19_1	83k	65k	0.2M	0.2M	27k	49k	46k	509	469	519	521	301	434	467
openssl_1_3	2M	51k	0.6M	0.7M	23k	0.5M	98k	596	587	588	588	596	586	586
openssl_1_5	0.3M	99k	0.1M	0.1M	93k	0.2M	0.2M	556	490	494	494	509	495	505
openssl_20_3	1M	35k	0.3M	0.3M	17k	0.6M	88k	596	587	592	592	596	587	586
openssl_20_4	78k	44k	37k	38k	25k	43k	33k	523	463	413	414	449	417	446
openssl_4_4	81k	42k	35k	38k	25k	42k	34k	522	461	409	413	453	418	444
openssl_5_1	83k	74k	0.2M	0.2M	24k	48k	68k	507	468	517	519	301	428	466
openssl_6_4	82k	42k	36k	35k	25k	42k	39k	527	456	404	399	449	422	444
openssl_6_5	0.3M	95k	0.1M	0.1M	0.1M	0.2M	0.2M	556	494	493	494	511	497	505
openssl_6_6	0.9M	0.4M	0.5M	0.5M	0.3M	0.6M	6M	592	572	583	582	576	584	586
openssl_7_2	0.8M	0.5M	0.5M	0.5M	0.5M	1M	9M	600	596	600	599	599	599	598
openssl_7_4	83k	40k	35k	35k	25k	42k	41k	522	460	405	408	450	422	444
openssl_8_1	86k	75k	0.2M	0.2M	24k	50k	49k	508	468	517	518	301	430	466
openssl_9_1	82k	74k	37k	37k	25k	48k	48k	507	469	396	393	299	427	466
php_11_2	3M	0.1M	1M	1M	1M	×	7M	599	587	599	599	597	×	596
php_15_2	3M	0.1M	11M	10M	0.8M	×	7M	599	586	599	599	596	×	596
php_16_3	8M	0.1M	11M	11M	3M	×	9M	600	590	600	600	600	×	597
php_4_1	7M	0.1M	12M	11M	2M	×	9M	600	588	599	599	599	×	597
php_6_2	3M	0.1M	9M	11M	0.5M	×	7M	599	586	599	599	597	×	596
poppler_17_1	27k	8k	28k	27k	16k	19k	26k	0	0	0	0	0	0	0
poppler_3_1	27k	9k	27k	27k	16k	19k	26k	0	0	0	0	0	0	0
poppler_9_1	27k	8k	19k	19k	16k	19k	26k	0	0	0	0	0	0	0
sqlite3_18_1	27k	4k	25k	25k	24k	25k	53k	0	0	0	0	0	0	0
sqlite3_20_1	27k	4k	25k	25k	24k	25k	55k	0	0	0	0	0	0	0

Let us now get to the analysis of Table 9. Table 9 shows, on the left-hand side, the median number of executions conducted by each fuzzer. This number will be strictly larger than the number of inputs run by the fuzzer, as it includes

additional calibration executions for each interesting input, as well as the executions involved in running the seed corpus. On the right-hand side, we show the median true fuzz time—i.e., the time spent generating and running new inputs rather than running through the corpus—rounded to the nearest second. This number is calculated as mentioned above. The value that is statistically significantly lower (at $\alpha = 0.05$, calculated with a Mann-Whitney U-test) than most other values is underlined; this turns out to just be the smallest observed value.

On summary over the entire table, we see that AFL++ most often has the lowest number of executions (on 27 benchmarks) and the lowest median true fuzz time (on 26 benchmarks). FFD is the second in line as the most-often lowest, with the lowest median executions on 14 benchmarks, and the lowest median fuzz time of 7 benchmarks. Interestingly, TuneFuzz, which is derived from AFL++, only has the lowest median number of executions on 4 benchmarks, and the lowest median fuzz time on 5 benchmarks. This relatively slower performance from AFL++, paired with the fact that the starting corpus is generated by AFL++, may explain why AFL++ does not outperform AFL in our main evaluation.

Two sets of benchmark stand out on both these metrics: the poppler family benchmarks and the sqlite3 family benchmarks, both at the bottom of the table. Both of these have median true fuzz times of 0. These are both quite complex programs, so we accumulated large seed corpora in our 24-hour run (14k inputs for poppler, 9.2k inputs for sqlite). Further, due to their complexity, the programs have relatively slower execution times. So for AFL, the average execution time of seed inputs is 0.022s on poppler and 0.017s on sqlite; for AFL++ it is 0.073s for poppler and 0.151s for sqlite. Thus, the fuzzers do not finish loading the seed corpus before the 600s timeout is reached. The different speeds at which the fuzzers go through the seed corpus likely account for the difference in bug reaching/triggering time for sqlite3_20_1. Thus, AFL++'s inability to find this bug likely reflects the fact that it did not make it through loading the entire seed corpus, not that it has inferior search capabilities.

We do not know a priori which bugs our seed inputs trigger, as the AFL++ runs generating our seed corpora were run on versions of the targets without bugs added. Calculating the true fuzz time allows us to analyze whether bugs were reached or triggered *while loading the seed inputs* or *during the true fuzzing time*. Magma's additional instrumentation, which tracks whether bugs are reached or triggered, is running while fuzzers run through the seed corpus at the start of the fuzzing round. Magma then measures whether bugs are reached or triggered by regularly polling this file. As AFL-family fuzzers run seed inputs through the instrumented program under test, *runs of the seed inputs* can also cause bugs to be flagged as reached or triggered.

If our true fuzz time calculations (ref. Table 9) are correct, we can determine the time at which "true fuzzing" starts: $fuzzStartTime = timeout - trueFuzzTime$. If bug reaching/triggering time is before *fuzzStartTime*, we can say the bug was reached/triggered *while running through seed inputs*. On the other hand, if bug reaching/triggering time is far after *fuzzStartTime*, we can say that the bug was reached/triggered *during true fuzzing*.

A tricky situation occurs if the bug is reached/triggered after *fuzzStartTime*, but not too far after. Because we used the default Magma polling interval of 5 seconds, if a bug is reached in the next POLL after *fuzzStartTime*, we cannot know whether the bug is reached in the time interval $(bugTime - 5, fuzzStartTime]$ (while running through seed inputs) or in the time interval $(fuzzStartTime, bugTime]$ (while true fuzzing). For triggering, the time interval of uncertainty is on any bug triggered $2 \cdot \texttt{POLL}$ after *fuzzStartTime* (ref. the discussion on *Magma POLL Delay in Reporting Triggers* at the end of Section 3.5). In these cases, we can only say that *maybe* the reaching/triggering occurred before true fuzzing.

A quick glance at Table 3 and the right-hand side of Table 9 suggests that several bugs are indeed reached while running through seed inputs. Consider, for instance, libsndfile_7_1, which is reached in 5 seconds by most fuzzers. The true fuzz times in Table 9 are at most 581 for this benchmark, and closer to 560 for most fuzzers. Thus, $fuzzStartTime > 19$ for all fuzzers while $bugTime = 5$. This means this bug is *definitely* reached while going through seed inputs.

We do this calculation for each pair of (bug reaching/triggering time, true fuzz time) in our dataset. Considering each iteration separately, and discarding iterations when we cannot calculate true fuzz time (some runs are lacking `plot_data` or `fuzzer_stats` files), we have a total of 1360 pairs to analyze.

Of these 1360, we can conclude that 1172 bug reaching/triggering times *definitely happened while loading seeds*. This should not be too surprising, considering the number of bug reaching/triggering times of 5s in Table 3. A further 177 bug reaching/triggering times cannot be determined to happen while loading seeds or during true fuzzing, due to the polling interval of 5 seconds—our "maybe"s. This means that for the AFL-family results, *only 4* bug reaches/triggers are *definitely* found during true fuzzing time. This is a measly *0.3%* of our observations on the AFL-family fuzzers.

The 4 bug reaches/triggers that occurred during true fuzzing time are all triggers: openssl_20_4 by AFL and TuneFuzz, and libxml2_12_2 by TuneFuzz. This aligns nicely with the observation that these are only triggered in a few runs by these fuzzers. Given the randomness inherent in fuzzing, the fact that the triggers that occurred during true fuzzing time were not triggered in 100% of runs—especially in a short timeout—makes complete sense.

The "maybe" set is significantly larger, but it is not very diverse. Out of the 177, we have 100 maybes about reaching/triggering php_11_2, due to short queue load times. For AFL++, with its longer seed loading time, this bug was reached/triggered *while* loading seeds. So these 100 maybes most likely also occur while loading seeds.[6] Another 67 maybes are about reaching the libpng_4_1 bug, again due to short queue load times. To investigate, we ran three 10s runs of AFL++ on this benchmark, with `POLL` = 1. All runs had a true fuzzing time of 6s, meaning true fuzzing started at 4s. They also all had a bug reaching time of 3s. This suggests the 67 libpng maybes also occurred while loading seeds. The 10 remaining maybes are about the triggering of libsndfile_7_1 by AFL. Since for all the other fuzzers, the triggering time is neatly before *fuzzStartTime*, it is likely these maybes also occurred while loading seeds.

In short, it seems quite likely these "maybe" times are all "while loading seeds" times. This maintains that 99.7% of our observed bugs reached/triggered for AFL-family fuzzers happened *while loading seeds*.

Note there could be inaccuracies in our accounting of true fuzz time: we calculated this metric in a post-hoc manner, using `fuzzer_stats` and `plot_data` for the AFL-(not AFL++)-family fuzzers. Nevertheless, this result aligns with the results in Table 3, where out of the 135 entries for survival times AFL-family fuzzers, 96 of them (71%) are exactly 5 seconds. A further 22 entries (16%) are bug triggers at 10 seconds, which could actually reflect bug triggers at 5s (ref. to the end of Section 3.5). While there is a difference between 87% and 99.7%, they both point to the majority of bugs reaching/triggering observations happening while loading seeds.

This result can be viewed in a few ways. One could say that the bugs reached/triggered are quite shallow. But, remember that our seed corpus was generated by running AFL++ on each benchmark for 24 hours. Thus, what is more likely shown in our evaluation is that *a long fuzzing run by AFL++* on a prior version of the code generates a seed corpus with good bug-revealing power, even on a slightly different version of the code. The true fuzzing time leads to new bug finds in only a few observations.

Execution Speed and True Fuzzing Time Differences

AFL++'s underperformance of AFL might be explained by its slower execution speed. For AFL-family fuzzers, 99.7% of bug reaches and triggers from our evaluation likely occurred while loading the seed corpus. However, using a saturated seed corpus leads to long seed loading times, which may impede fuzzing on some benchmarks.

[6] It is while investigating this difference in triggering that we uncovered the *Magma POLL Delay in Reporting Triggers*, discussed at the end of Section 3.5.

5 Discussion

What is to be learned from our results?

First let us consider threats to generalization. Our CI/CD setting is a simulation. The main aspects of this simulation are: (a) a short timeout on fuzzing times, (b) "commits" consisting of Magma patches, and (c) a large seed corpus, generated by previous fuzzing runs. Perhaps the aspect that could change our results the most is (b): Magma patches add a single bug, while commits in practice could change behaviour in ways different from this. This may mean fuzzing in a CI/CD setting with bigger commits may show more definitive gains over simply using the seed corpus as a regression suite. Further, for (c), our seed corpus, while large and generated from a 24-hour fuzzing run, may not be as fully saturated as the seed corpus from a multi-day or multi-week fuzzing run. A larger seed corpus may increase the power of the seed corpus as a regression suite, but running through it completely may take too long.

Then, there is our benchmark selection. We selected our benchmarks—(fuzzer, bug, fuzz target) tuples—randomly amongst those that passed the checksum check (checksum of binaries was different before/after applying the Magma patch). It is possible that for some of these, the bug is not reachable at all from the fuzz target. This is supported by the fact that for most of the benchmarks, target function coverage is zero. Thus, our analysis may be overly negative.

An alternative benchmark selection is suggested by Geretto et. al [17] in libAFLGo, published after we selected our benchmarks. They select their tuples by running 7-day-long fuzzing campaigns, keeping only benchmarks that were reachable in 7 days. Interestingly, however, two of the benchmarks we trigger bugs in—libxml2_12_2 and libxml2_1_2—are excluded from their benchmark set. We contrast our results in more detail in the next section. It is not clear which of these benchmark selections is most reflective of CI/CD fuzzing in practice; commits could very well affect code that is not reachable from a fuzz driver.

In terms of the research questions we set out to answer, first outlined in our registered report [20], we find that in our CI/CD simulation, undirected fuzzing (in particular, AFL) is more efficient in reaching/triggering bugs, and that bloat in commits has little effect on the bug reaching/triggering abilities of directed fuzzers.

Our analysis in Section 4.5.2 of true fuzzing times—and whether bugs were reached/triggered during true fuzzing times—helps contextualize the answers to these research questions in practice. If our measurement of true fuzzing times is correct, most of the bugs reached and triggered in our evaluation (99.7% of all observations for AFL-family fuzzers) are reached/triggered *while loading the starting seed corpus*. As our starting seed corpus is generated from a 24-hour run of AFL++, this suggests that in our evaluation, *adding the results of a very short fuzzing run* on a current version of the code *does not greatly increase* bug reaching/triggering power, compared to *the results from a long fuzzing run* on a previous version of the code.

In practical terms: in our simulation, most bugs reached/triggered could be reached or triggered simply by using the corpus generated from a longer fuzzing run as a regression test suite. Our context is only a simulation, but it supports the choice to use fuzzing as a long-running background testing process, and use the corpus from this fuzzing run as a regression test suite in CI/CD. An additional advantage of this choice is determinism. Assuming the program under test is deterministic, a regression test suite will pass/fail in the same way on the same version of the code. This should be more stable than a fuzzing run, which may or may not report some new bug result. The disadvantage of the corpus-as-regression-test method is the loss of potential to find inputs beyond the seeds; the advantage is determinism and saved computation time.

The long queue load times for sqlite3 and poppler suggest an additional disadvantage of the corpus-as-regression-test approach: large, bloated seed corpora may be prohibitively expensive to execute as a regression test suite. The Magma

seed corpus for poppler included some relatively large pdf files, so the size of our saturated corpus was nearly 5GB. Reducing this corpus for regression testing purposes may be of interest.

The poor performance of libFuzzer in our evaluation is curious, given that libFuzzer scores higher than AFL on current FuzzBench evaluations [29]. We believe the explanation of this is the use of the `-fork=1` flag in Magma's configuration of libFuzzer. This setting allows libFuzzer to continue fuzzing even after encountering a crash: its default setting stops fuzzing at the first crash found. However, it also means that each child process is given "small random subsets of the corpus" to fuzz from. From the best we can tell from the libFuzzer implementation [30], given a corpus C, these random subsets are of size $\sqrt{|C| + 2}$. If this interpretation is correct, this means libFuzzer does not systematically go through all of the seed corpus while fuzzing. For some default Magma corpuses–e.g., 4 elements for libpng—this may not be a huge disadvantage. But with our large seed corpora, it is a huge disadvantage. Given that most bugs are reached/triggered while going through the seed corpus for AFL-family fuzzers in our evaluation, we judge this as the most plausible explanation of libFuzzer's poor performance.

Long instrumentation times (i.e., time to build the program under test with fuzzing instrumentation) have been pointed out as a disadvantage of directed fuzzers [46], especially of directed fuzzers in a short timeout environment. In our evaluation, a directed fuzzer (TuneFuzz, which aims to target all sanitizer-instrumented code) had the longest instrumentation times. For openssl, TuneFuzz's instrumentation times were around 300s (ref. Figure 2). That said, there was little effect in our evaluation on instrumentation times actually affecting bug reaching or triggering times. Only the triggering of libxml2_12_2 and openssl_20_4 by TuneFuzz was pushed over the timeout due to instrumentation times.

However, this result bears a bit more weight when considering our analysis of which bug reaches/triggers happened while loading the seeds. Recall that we identified in our main evaluation *only 4* bug reaches/triggers by AFL-family fuzzers that happened during true fuzzing in our main evaluation. In the sensitivity evaluation, we see one additional bug reach/trigger: FFD triggers openssl_20_4 once (Table 5), in 210s (232s with instrumentation). So, the two bug triggers by TuneFuzz that were pushed over timeout are 2/5 of the AFL-family bug reaches/triggers we observed during "true fuzzing time". The number of observations here is small, but the large proportion that are timed out suggests that long instrumentation times are still of importance in short-timeout fuzzing.

Overall, our results provide support for the simple approach of using long-running fuzzing session corpora as regression test suites in a CI/CD setting. Further evaluation and development of directed fuzzers may want to consider starting from a saturated, or at least 24-hour-run, seed corpus, to identify more impactful techniques for CI/CD fuzzing.

6 Related Work

Testing plays an important role in CI/CD pipelines. While continuous testing originally referred to executing unit tests on a continuous basis [33], the term now 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 [33].

Continuous testing can enhance development productivity. Saff and Ernst propose a model for investigating the usefulness of continuous testing [43]. Their study first shows that regression errors caught earlier are easier to fix [43]. 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, which 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 [43]. The authors further show in a study that developers who use continuous testing are three times more likely to complete the coding task before the deadline [44]. Most participants in their study confirmed the usefulness of continuous testing

Table 10. Mean survival times in seconds to trigger each bug, including instrumentation time, from our evaluation and Gerreto et al.'s reimplementation of directed fuzzers (their Table 10) [17]. Fastest between each (AFL, libAFL) and (AFLGo, AFLGoE, libAFLGo) highlighted in grey . - means the benchmark was not present in the evaluation; n/t means the bug was not triggered in any observation.

	Ours			Geretto et al [17]		
	AFL	AFLGo	AFLGoE	libAFL	libAFLGo	libDAFL
libsndfile_7_1	88	128	134	300	210	114
libtiff_7_1	41	61	66	42	68	210
libxml2_1_2	44	75	78	-	-	-
openssl_20_4	595	n/t	n/t	36k	7.5k	27k
php_11_2	116	147	134	162	360	12k
sqlite3_18_1	73	89	90	71	120	534
sqlite3_20_1	131	595	n/t	n/t	n/t	n/t

and found it helpful for them to write better code faster [44]. Continuous fuzzing in this line—continuously fuzzing a program, updated for code changes, over a whole development day—would be an interesting translation of this work to the fuzzing space. Our results suggest it may provide better power of fuzzing than short-timeout runs on every commit.

Challenges like reducing test time, increasing the visibility of test results, and supporting (semi-)automated testing are important for testing in a CI/CD setting [45]. However, in huge codebases, testing each commit is not sustainable due to limited computing resources [25]. A study of Google-scale continuous testing has confirmed that code modified more often is more likely to cause breakages than code modified less often [34]. Motivated by similar findings, Zhu and Böhme propose a regression greybox fuzzer that focuses on fuzzing code that was changed more recently [54]. 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 [54]. We find that, when starting with a corpus from a longer fuzzing run, the main bug-finding effectiveness of 10-minute fuzzing runs comes from running the corpus. Zhu and Böhme's findings imply that there is still value from more fuzzing at this point—but our results suggest that its benefits may appear at longer fuzzing times.

In our CI/CD simulation, we run fuzzing campaigns for changed code. Our setup might be applicable to evaluating patch testing, which automatically tests code patches [32]. 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 [32]. Their experiments show that automatic techniques can increase patch coverage—of which reaching a high value is genuinely hard—and detect bugs during testing [32]. Kuchta et al. further propose executing the old and new versions of the code in the same symbolic execution instance, with the old shadowing the new one [26]. 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 [26]. The symbolic execution used in these prior works on patch testing requires heavy-weight program analysis and constraint solving, which might not be scalable for large-scale projects or working environments that need rapid test feedback. Given that we found little yield for additional fuzzing on top of running a seed corpus, it may be worth trading off this time to run symbolic-execution-based patch testing.

Our evaluation aimed to compare the power of directed and undirected fuzzing at reaching bugs in a CI/CD setting. But in our setting with 24-hour-saturated corpora and short timeouts, we saw few differences between the approaches. Geretto et al.'s libAFLGo [17] provides a comparison of directed and undirected fuzzers with a 24-hour timeout, allowing

differences in behaviour to be better observed. Geretto et al. reimplement three directed greybox fuzzers—AFLGo [9], Hawkeye [10], and DAFL [23]—in the modern libAFL infrastructure, to evaluate whether the improvements proposed in these works remain improvements when implemented in a more modern and efficient fuzzing platform. They find that libAFlGo only improves on libAFL's time to trigger bugs on 1 of 40 benchmarks, and that the longer build times of libHawkeye and libDAFL hurt their performance.

Unlike our evaluation, Geretto et al. use the seed corpora provided by Magma.[7] For an idea of how using a saturated corpora might change results, we compare the mean survival times, including instrumentation, for the bugs we trigger to those reported in libAFlGo in Table 10. This table contains the 6 benchmarks we have that overlap with those in libAFLGo, as well as libxml2_1_2, which we trigger bugs in but is not present in libAFLGo. We omit libxml2_12_2, because the bug trigger is pushed over 600s when considering instrumentation time.

In general, in spite of our older versions of fuzzers, we witness lower triggering times on most of these benchmarks. The timings are eerily similar for libtiff_7_1 and sqlite3_18_1. Note that the experiments are run on different hardware architectures, so the benchmarks with small differences in runtimes may not be directly comparable. Quite different is the time to expose openssl_20_4; this was all in the hours in the libAFLGo evaluation, but AFL twice (and FFD, once in the sensitivity experiments) triggered it within 10 minutes. The maximum value of mean survival time is capped to the timeout, so we might see longer mean survival times for this benchmark if we had increased our timeout. We might see it triggered more consistently as well: in libAFLGo, this bug is triggered in 94% of runs (unlike 20% in our case). Also interesting is that sqlite3_20_1 was consistently triggered by our AFL, but not triggered in Geretto et al.'s evaluation at all. Perhaps the AFL++ cmplog feature, which was enabled while creating the starting corpus, is important in uncovering this bug. Overall, this comparison suggests, not surprisingly, that a 24-hour seed corpus on a previous version of the code remains very powerful at exposing bugs—even when compared to modern directed fuzzers.

7 Conclusion

In a timeout of 10 minutes, we reach or trigger bugs in 15 of our 50 benchmarks. AFL and TuneFuzz are the best-performing fuzzers in our evaluation: both reach 15 bugs and trigger 6 bugs on average during fuzzing campaigns. Overall, there is a tie between directed and undirected fuzzers, as the difference in the number of bugs reached and triggered is not significant between these two kinds of fuzzers. This is explained by the fact that the near entirety of bugs reached and triggered in our evaluation are reached and triggered while loading the seed corpus, not while truly fuzzing (i.e., mutating and running new inputs). We find that long instrumentation times have a minimal effect on whether bugs are reached or triggered within timeout overall, but push over a few of the "true fuzzing" bug finds over the 10-minute timeout. Incidentally, we identify a few details of the Magma infrastructure that lead to minor delays in the reported time to reach/trigger bugs. These are unlikely to have a significant impact on the conclusions of work that uses the results to compare fuzzers, but should be considered for analyses that need precision in reaching/triggering times. Our results suggest that, in settings where long-running fuzzing campaigns produce thorough seed corpuses, using these seed corpuses as regression tests may provide most of the bug-reaching power of a short (i.e., 10-minute) fuzzing run. We note that in our CI/CD simulation, the changes in the simulated commits are quite small; it is possible that short fuzzing runs may have more bug-finding capability in the presence of larger changes to the code. Nevertheless, our results suggest that future work in fuzzing which aims to provide impact in a CI/CD setting may want to focus on solving the particular challenges that emerge when starting from a near-saturated seed corpus, rather than faster time to exposure starting from a small seed corpus.

[7]Except on sqlite3, because the Magma corpus contains TCL files rather than SQL statements—they use SQL statements from the codebase as seeds.

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. This work is supported in part by a Google Research Scholar Award. Thanks to our FUZZING'24 and TOSEM reviewers for their in-depth comments and suggestions; the paper is better thanks to their feedback. Thanks to Cornelius Aschermann for rubber-ducking some of our analysis and data presentation decisions.

References

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

[2] 2024. DAFL Issue 4. https://github.com/prosyslab/DAFL-artifact/issues/4. Accessed: 2025-08-21.

[3] Advanced Fuzzing League ++. [n. d.]. Fuzzing with AFL++. https://aflplus.plus/docs/fuzzing_in_depth/. Accessed: 2024-11-10.

[4] Advanced Fuzzing League ++. 2024. afl-fuzz.c. https://github.com/AFLplusplus/AFLplusplus/blob/stable/src/afl-fuzz.c. Accessed: 2024-10-30.

[5] 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

[6] 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

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

[8] 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

[9] 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

[10] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. 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, 2095–2108. https://doi.org/10.1145/3243734.3243849

[11] Magma Contributors. 2020. Implementation of magma_log. https://github.com/HexHive/magma/blob/v1.2/magma/src/canary.c#L58. Accessed: 2025-09-19.

[12] Magma Contributors. 2020. Reading over the consumer array in –fetch file mode. https://github.com/HexHive/magma/blob/v1.2/magma/src/monitor.c#L198. Accessed: 2025-09-19.

[13] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2440–2451. https://doi.org/10.1145/3510003.3510197

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

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

[16] 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

[17] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. 2025. LIBAFLGO: Evaluating and Advancing Directed Greybox Fuzzing. In *Proceedings of the 10th IEEE European Symposium on Security and Privacy (EuroS&P 2025)*. Retrieved Jun 3, 2025 from https://download.vusec.net/papers/libaflgo_eurosp25.pdf

[18] Kyunghwa Han and Inkyung Jung. 2022. Restricted mean survival time for survival analysis: a quick guide for clinical researchers. *Korean Journal of Radiology* 23, 5 (2022), 495.

[19] 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

[20] 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* (Vienna, Austria) *(FUZZING 2024)*. Association for Computing Machinery, New York, NY, USA, 33–41. https://doi.org/10.1145/3678722.3685532

[21] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] Caroline Lemieux and Adrian Herrera. 2025. MAGMA pull request #179: Fix check for empty monitor directory when setting starting counter, which leads to off-by-POLL error in monitor times. https://github.com/HexHive/magma/pull/179. Accessed: 2025-09-17.

[28] AFLGo Maintainers. 2023. Default Time to Exploitation is 10 minutes in AFLGo. https://github.com/aflgo/aflgo/blob/fa125da5d70621daf7141c6279877c97708c8c1f/afl-2.57b/afl-fuzz.c#L313. Accessed: 2025-09-23.

[29] Fuzzbench Maintainers. 2025. FuzzBench Report: 2025-03-12. https://www.fuzzbench.com/reports/2025-03-12/index.html. Accessed: 2025-08-22.

[30] LLVM Maintainers. 2025. Code Deciding Sub-Corpus Size. https://github.com/llvm/llvm-project/blob/dcce216289b10f01ac4e974efcbd486f79cc35ea/compiler-rt/lib/fuzzer/FuzzerFork.cpp#L139. Accessed: 2025-09-19.

[31] OSS-Fuzz Maintainers. 2025. OSS-Fuzz Trophies. https://github.com/google/oss-fuzz?tab=readme-ov-file#trophies. Accessed: 2025-09-24.

[32] 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

[33] 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

[34] 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

[35] 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.

[36] 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

[37] 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

[38] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[39] 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

[40] 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

[41] LLVM Project. 2024. libFuzzer – a Library for Coverage-Guided Fuzz Testing. https://web.archive.org/web/20250911024621/https://llvm.org/docs/LibFuzzer.html. Accessed: 2025-09-12.

[42] 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

[43] 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

[44] 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

[45] 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

[46] Arindam Sharma, Cristian Cadar, and Jonathan Metzman. 2024. Effective Fuzzing within CI/CD Pipelines (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop* (Vienna, Austria) *(FUZZING 2024)*. Association for Computing Machinery, New York, NY, USA, 52–60. https://doi.org/10.1145/3678722.3685534

[47] 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

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

[49] 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

[50] MichałZalewski. 2014. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2025-09-26.

[51] 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

[52] Han Zheng, Flavio Toffalini, and Mathias Payer. 2024. TuneFuzz: Adaptively Exploring Target Programs. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing* (Lisbon, Portugal) *(SBFT '24)*. Association for Computing Machinery, New York, NY, USA, 61–62. https://doi.org/10.1145/3643659.3648564

[53] 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

[54] 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