Shifting Fuzzing Left in Software Workflows

Dylan J. Wolff[1*], Ridwan Shariffdeen[1], Yannic Noller[2], Abhik Roychoudhury[1]

[1]National University of Singapore, Singapore.
[2]Ruhr University Bochum, Germany.

*Corresponding author(s). E-mail(s): wolffd@comp.nus.edu.sg;
Contributing authors: ridwan@comp.nus.edu.sg; yannic.noller@acm.org;
abhik@nus.edu.sg;

Abstract

Fuzzing has proven to be an effective tool for finding bugs in software, with Google's OSSFuzz alone being responsible for finding thousands of critical security vulnerabilities in open source projects. As software development practices evolve, there is a growing recognition of the need to integrate security testing earlier in the development process. Yet in our survey of software practitioners, only 20% used fuzzing as part of their development workflow. In this paper, we explore how fuzzing can fit into the software development life cycle. We do so with two empirical studies, including perspectives from over 40 industry practitioners. First, in a survey of software professionals, we identify several gaps between current state-of-the-art fuzzers' capabilities and engineers' expectations. In particular, we find that most developers are willing to use fuzzers, but prefer shorter, more frequent fuzzing runs as part of a continuous integration/continuous deployment (CI/CD) workflow. Next, based on results of this survey, we assess state-of-the-art fuzzers' capabilities in the context of CI/CD and local development workflows. We observe that existing fuzzers can find up to 50% of bugs within 5 minutes of fuzzing, meeting developer expectations from our survey, but that further research is needed to uncover more difficult bugs within tolerable time limits. Additionally, we see that the initial corpus and time needed to build and analyze a project for a particular fuzzer both have a significant effect on fuzzer effectiveness in this context. We hope that our work will help the community to drive wider adoption of fuzzing in the software development lifecycle.

Keywords: fuzzing, empirical study, program repair, software security

1 Introduction

Software bugs introduced during the development process can have catastrophic consequences if they are not uncovered and fixed before reaching production (Goodin (2023); Newman (2021)). In 2017, exploitation of a single vulnerability[1] in Apache Struts cost US credit agency Equifax an estimated 1.7 billion USD (Lane (2020)). To mitigate the impact of such critical bugs, one approach that has seen increasing attention (Winters et al (2020)) is to push validation activities towards earlier phases of development, sometimes referred to as a "shift left" (Smith (2001)).

Unit testing (Daka and Fraser (2014)), for example, has become emblematic of this effort, with developers manually writing tests for smaller units of functionality along with or even before (Beck (2002)) writing the application code itself. Increasingly, these unit tests are run as part of Continuous Integration and Continuous Deployment (CI/CD) workflows to ensure that code that is committed or deployed has always passed these checks. Still, in our surveyed users, many found that such tests can be onerous to write (31/44 participants) and difficult to maintain as software evolves (34/44). Moreover, unit testing (for testing functionality) has been widely employed by practitioners for roughly two decades (Runeson (2006); Micco (2018)), but security and reliability issues continue to plague the software industry today. Static analysis (Calcagno and Distefano (2011)) has also been leveraged to detect bugs earlier in development cycles (Churchill (2018); Distefano et al (2019); Jin et al (2023)). However, static analyses typically generate large numbers of false positives, resulting in trust issues among the developer community (Phan et al (2023)).

Recently, fuzzing (Miller et al (1990)) has emerged as another effective technique for finding security vulnerabilities and other classes of critical bugs (Liang et al (2018); Mansur et al (2020)). Unlike static analysis, it has higher precision, providing a reproducing test case for each bug found. Unlike unit testing, it is highly automated, generating test cases without human intervention. However, while much research has been dedicated to improve the bug detection capabilities of fuzzers (Meng et al (2022); Menendez and Clark (2022); Pham et al (2020)), even expert users face many challenges in using existing fuzzers (Nourry et al (2023)). Furthermore, little or no research has been conducted on user needs for fuzzers in left-shifted software development workflows such as automated CI/CD pipelines.

In this paper, we investigate what would be needed to make fuzzers useful at earlier stages of development for a software practitioner who is not necessarily a fuzzing expert. What are the gaps between these sofware professionals' expectations and existing fuzzing tools or workflows? What changes would drive broader adoption of fuzzing at early stages in the development process? How do these changes impact the way researchers design and evaluate fuzzers? We examine industry practitioner perspectives on development workflows that utilize fuzzing to provide early assurance during development . We hope these user insights will open new research directions in which fuzzing can be a pathway for shifting automated validation to the left in software development.

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5638

Concretely, we first examine user expectations in fuzzing through a survey, aiming to answer the following research questions:

RQ1 What workflows and use cases do practitioners have for fuzzing?
RQ2 What kinds of inputs and setup will users undertake for fuzzers?
RQ3 What kinds of outputs and artifacts do users expect from fuzzers?
RQ4 How do users evaluate the effectiveness of fuzzing tools?

We surveyed 44 industry practitioners with 1-10 years of experience and asked how they envision a fuzzer's role in the development process. Most of our survey participants are highly technical developers who are not necessarily fuzzing experts, with half of them having more than five years of experience in software development.

From the results of our survey, we formulate and answer two additional questions to examine the feasibility of one highly desired development workflow with existing fuzzing tools, namely:

RQ5 How effective are existing fuzzers at testing code-changes in a software project?
RQ6 How do additional variables affect fuzzer effectiveness in this scenario?

For these questions, we analyze the current capabilities of state-of-the-art fuzzers' in continuous testing. We compare the efficacy of these tools with respect to developer expectations and desired workflows. In total, we evaluate two general purpose fuzzers (AFL[2] and AFL++ by Fioraldi et al (2020)), two commit/regression fuzzers (AFLChurn by Zhu and Böhme (2021), and CIDFuzz by Zhang et al (2023)) and two directed fuzzers (AFLGo by Böhme et al (2017) and SelectFuzz by Luo et al (2023)). Our results indicate that state-of-the-art fuzzers can find rougly 50%-60% of the regression bugs in our data set within a time period desired for local development or CI/CD workflows, respectively. Additionally, we find that the context of the fuzzing campaign: how long a fuzzer takes to instrument and analyze the code of a target program, and the initial test cases given to the fuzzer both have a *significant* impact on fuzzer effectiveness in this scenario. These results show that, while existing tools can already provide a substantial benefit in this context, more research is needed to find all bugs within the shorter time frames desired by developers.

2 Background

In this section we first provide a background on software testing and software fuzzing. To explain the challenges faced by industry practitioners in using fuzzing, we also provide background on regression testing, directed fuzzing and commit fuzzing.

2.1 Software Testing

Regression Testing

Regression testing is a technique designed to ensure that existing features continue to function as expected after a code change. It leverages a previously executed test suite to identify any new defects introduced by the recent modifications that were not present before the changes.

[2] https://github.com/google/AFL

Test Cases and Inputs

In this paper, we use test cases and inputs interchangeably to refer to data that is passed to the program under test during execution. For example, a test case for a PDF reader like OpenPDF[3] would be a PDF file.

2.2 Software Fuzzing

Fuzzing

Fuzzing (Miller et al (1990)) is broadly defined as passing randomized inputs to a target program and observing whether or not the program crashes. In the context of this paper, when we mention fuzzing we always refer specifically to *mutational grey-box* fuzzing. This class of fuzzer has been widely successful in research and industry, and has been deployed at scale across hundreds of open-source projects within the OSS-Fuzz framework (Serebryany (2017)).

Mutational Grey-Box Fuzzing

In grey-box fuzzing, the program is *instrumented* to provide a signal to the fuzzer when new behavior is observed. This allows the fuzzer to bias the randomized search towards (or away from) particular behaviors. Typically, grey-box fuzzers use some form of code-coverage (e.g. branch coverage) as feedback to bias the search. Mutational fuzzers generate new random inputs by mutating existing inputs, rather than sampling from the space of all possible inputs.

Initial Corpus

As mutational fuzzers generate random inputs from existing inputs, they are usually provided with a set of inputs that explore some basic program behaviors to expedite the search, called an *initial corpus*. These are often either hand-crafted by developers, downloaded from an online repository of inputs with a common format (e.g. PNG files), or generated by a prior run of a fuzzing tool.

Fuzz Campaign

A set period of time during with a fuzzer is applied to a target program with a given initial corpus in the hopes of finding bugs.

Directed Fuzzing

Directed fuzzing is a class of grey-box fuzzing in which the fuzzer prioritizes exploring behaviors of specific code regions of interest (Böhme et al (2017)). These areas of interest can vary based on the testing requirements, and could be provided by a human auditor, static analysis tool, or other source. This approach is in contrast to general purpose fuzzers, which are designed to explore *any* new (hopefully buggy) behaviors in the program, regardless of location. In the context of this work, directed fuzzers can be set to target the behavior of changed lines of code, similar to commit and regression fuzzing (defined below).

[3] https://github.com/LibrePDF/OpenPDF

Commit Fuzzing

Commit fuzzing targets behaviors in locations affected by individual commits. When a new commit is made, inputs are generated specifically to exercise the program locations impacted by the code changes introduced in that commit (Zhang et al (2023)).

Regression Fuzzing

Regression fuzzing is a specialized process aimed at targeting code that has been recently and frequently modified. It focuses on directing the input generation towards these frequently changed code regions (Zhu and Böhme (2021)). Unlike commit fuzzing, which targets specific program locations affected by individual commits, regression fuzzing focuses on code regions that are frequently modified across multiple commits.

Code Coverage

Code coverage is a measure of what parts of the program were executed during one or more executions. This can be measured a number of different ways at different levels of abstraction set of source lines, paths, branches, basic blocks. In this work, when we discuss code coverage or coverage, we refer to control-flow graph *edge* coverage of a program, which is commonly used by fuzzers like AFL and AFL++ (Fioraldi et al (2020)). This property measures whether or not a transition between two basic blocks was observed in one or more executions of the program.

Sanitizers

Fuzzing is typically used to detect crashes, but not all bugs will necessarily crash the program. As such, several tools have been developed to serve as *test oracles* for fuzzing, exiting the program with a crash signal when buggy behavior is observed. The LLVM sanitizers are some commonly used examples[4], identifying memory safety, undefined behavior and other issues visible to a fuzzer at runtime in a target program, even if they would otherwise not crash the program.

Cumulative and Continuous Fuzzing

Cumulative fuzzing is when fuzzers are used on the same program over a long period of time (often multiple years), frequently as the program itself is undergoing active development. The version of the program being fuzzed is updated automatically as the program evolves. The most prominent example of this is OSS-Fuzz (Serebryany (2017)), where hundreds of open-source projects are fuzzed in the cloud.

3 Survey Methodology

To address the RQ1-4, we designed an online survey, which we distributed in March 2023 via invitations to contacts from global companies, using the "snowball" method (Kitchenham and Pfleeger (2008)). The participants did not receive any compensation;

[4] https://compiler-rt.llvm.org/

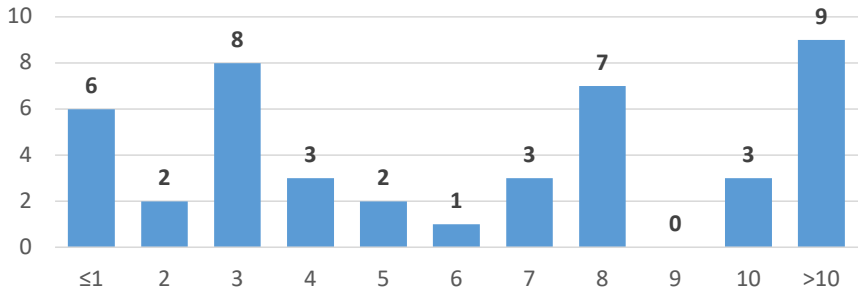


Fig. 1: Participant's years of experience in their current role.

however, for each participant, we donated $5 SGD to Doctors Without Borders/Médecins Sans Frontières (MSF). Note that before the study, we received the corresponding approval from our organization's Institutional Review Board (IRB).

Survey Overview

In total, we asked 47 questions about the *goals* and *usage scenarios* of fuzzing, as well as the relevant *inputs*, *setups*, *interactions* and *outputs*. Prior to the survey questions, all participants were presented with a brief textual definition and description of fuzzing, as well as a link to a short 4-minute video introduction to fuzzing produced by the Communications of the Association for Computing Machinery (CACM) (Godefroid (2020)). The survey includes open-ended questions like "If you use fuzzers regularly, please describe your step-by-step workflow(s) when using these tools." and close-ended questions like "At what scale would you conduct fuzzing campaigns?" with Multiple Choice, a 5-point Likert scale, or ranking options. The survey was created and deployed with Microsoft Forms. Our artifact (Section 8) contains a complete list of our questions.

Participants

In total, we received 51 survey responses from over eleven countries in a wide range of industries, e.g., technology (11), software services and consulting (7), general software development (5), and software security (4). Of the 51 total participants, seven identified themselves as academics or students. To accurately capture the opinions and experiences from *software practitioners*, rather than those of academics, we omit these seven responses from our analysis. Of the remaining participants, 61% (27/44) identified themselves as software developers. 78% (21/27) of these software developers have not used a fuzzer in their professional work setup. Other participants identified themselves as technical leads (6), managers (2) or in other roles like security architect, security engineer, or technical consultant. Figure 1 and 2 show the practical experience and familiarity with fuzzing of our participants. Across all participants, 52% (23/44) have not used a fuzzer yet. From the remaining participants, only 27% (12/44) indicated that they are expert users of automated software testing tools. However, 77% (34/44) of the participants are at least familiar with the concept of

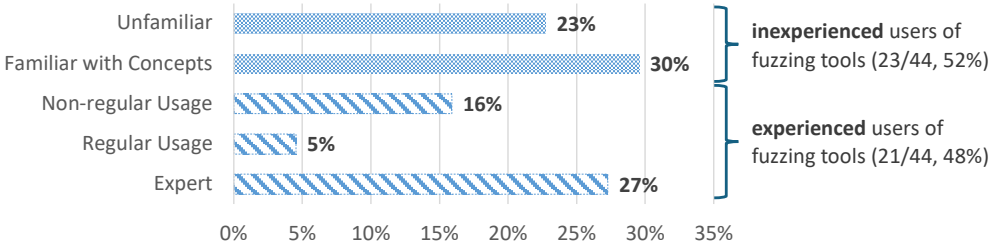


Fig. 2: Participant's familiarity with fuzzing/testing tools.

fuzz testing. Most of our participants are not experts in fuzzing, but typically had a large amount of technical experience (7.2 years mean, 6.5 years median). While only 12 participants considered themselves fuzzing experts, 34/44 were at least familiar with fuzzing concepts. The 10 remaining participants who are not at all familiar with fuzzers have a mean of 5.3 years technical experience. Therefore, we believe we have reached an interesting set of participants who can provide insights into expectations and current obstacles for the integration of fuzzing in broader software development workflows. We used the data shown in Figure 2 to separate the participants into two groups: experienced (21/44, 48%) who reported at least non-regular usage of fuzzers and inexperienced (23/44, 52%) practitioners. In our discussion, we highlight differences between these two groups, if they exist. For the remainder of this paper, when we say "experienced" or "inexperienced" developers, we refer to their experience with fuzzing/testing tools.

Analysis

For all questions with a 5-point Likert scale, we analyzed the distribution of negative/disagreement (1 and 2), neutral (3), and positive/agreement (4 and 5) responses. For the ranking questions, we analyzed which answers have been ranked highest. For the Multiple Choice questions, we analyzed which choices were selected most; further, we analyzed the open-ended "Other" choices and mapped them to the existing options or treated them as new ones if necessary. For all other open-ended questions, we performed a qualitative content analysis coding (Schreier (2012)) to summarize the themes and opinions. The first iteration of the analysis and coding was done by one author, followed by the other authors' review. The following sections summarize and discuss the most mentioned responses indicating their frequency in brackets. To compare the two groups of experienced and inexperienced developers in the context of fuzzing, we use the chi-square test of independence (Pearson (1900)) ($\alpha = 0.01$). The chi-square test of independence tests whether the two categorical variables are related. For example, a) the experience of developers in using fuzzing and b) their preferred (local/CI/on-demand) fuzzing campaign length. In particular, we test whether there is any statistically significant difference in their general preferences. We also test the significance of the obtained trends/majorities with the Binomial Test (Dodge (2008)) ($\alpha = 0.05$), e.g., is an observed majority of preferred usage actually significant in relation to the total number of participants. We present the corresponding P values. Our research artifact includes all data, statistics, and codes.

4 Survey Results

4.1 Workflows and Use Cases (RQ1)

The first section of our survey focused on high-level goals and workflows, to understand *how developers envision using fuzzing* in their day-to-day development process.

Testing Goals

To determine which kinds of bugs and test-oracles fuzzing researchers and practitioners should focus on, we asked our participants *"What properties are important to test your software for?"* (Figure 3). Classes of bugs commonly targeted by existing fuzzers and sanitizers, such as memory safety bugs, are deemed important by a strong majority of experienced and inexperienced developers. Functional correctness, however, was the most important property identified by nearly all of our participants (39/44, 89%, $P < .001$). Participants mentioned memory safety as the second most important property, followed by concurrency safety. Many other non-functional properties, such as execution speed, memory consumption and side-channel vulnerabilities were also deemed important by more than one-third of our participants. Contrary to the goals of our participants, in general, fuzzer and sanitizer support for finding non-crashing bugs is often limited, if it exists at all. The commonly used LLVM sanitizers, for example, can not identify liveness, performance, or side-channel vulnerabilities at all, and only find a subset of concurrency bugs (data-races and deadlocks) at the cost of extremely high memory and execution time overhead (often over 10x).

Many diverse fuzz-testing oracles, especially for functional correctness, are considered important by developers; better support for detecting these bug types may boost adoption of fuzzing tools.

We also asked developers *"What is important to gain from a fuzzing campaign (i.e. fuzzing a software application)?"* We did not observe any significant difference between the importance of the three possible outcomes we included in this question: obtaining a regression test suite, bug discovery, or exercising a particular code location. Additionally, we did not see a significant difference between the experienced and inexperienced participants in this case. Nevertheless, the majority of the participants indicated that all three of these three goals are important (31/44, 70%, $P < .01$). However, most state-of-the-art fuzzers do not provide any tools for understanding and maintaining such automatically generated test suites over time, other than test suite minimization tools.

Automated regression test suite generation is **as important as bug discovery** and **directed testing** to software practitioners.

Concrete Workflows

CI/CD has emerged as a key process in modern software development. We asked our participants whether they would include fuzzing in their CI/CD system and also *"How frequently would you run a fuzzer as part of your build integration or deployment*

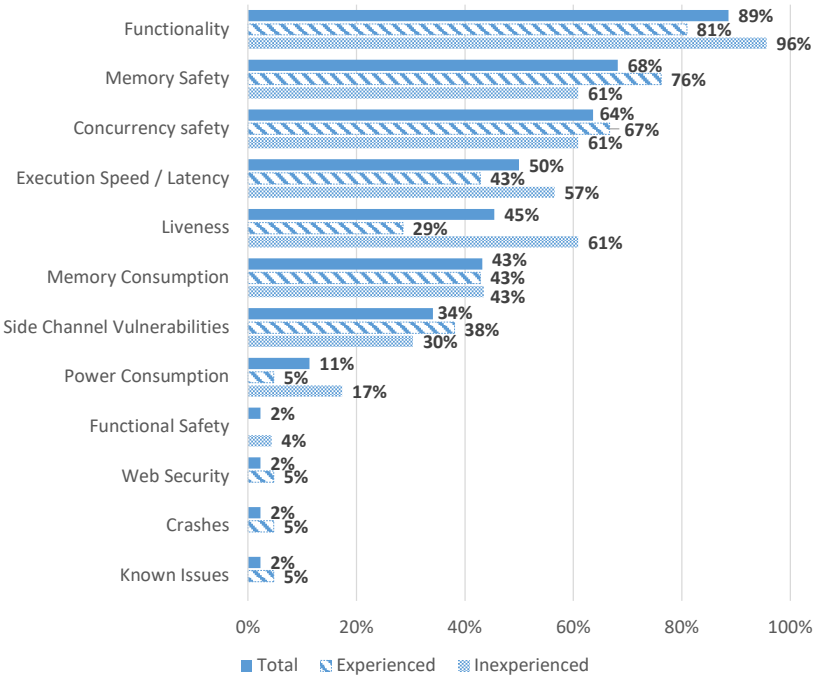


Fig. 3: Participants' opinions about the most important testing properties.

system?" to understand how fuzzing can be used in this context. We found that a large majority of our participants (33/44, 75%, $P < .001$) would add fuzzing to their CI/CD pipelines, predominantly as either per commit or in nightly administered fuzzing runs. We could not determine any significant difference between the responses of experienced and inexperienced participants for this topic.

We also queried about integrating fuzzing into the participants' local development workflows. Perhaps surprisingly, most participants (28/44, 64%, $P < .05$) would include fuzzing in their local development workflow. These practitioners would mostly run a fuzzer only as a local pre-commit or pre-merge step (19/28, 68%, $P < .05$), rather than more frequently (e.g., on each file-edit). Some experienced practitioners indicated that they would also periodically run a fuzzer multiple times before committing to the CI pipeline, while *no* inexperienced developers indicated they would do so (experienced: 5/21, 24%; inexperienced: 0/23, 0%; $P = .013 > .01$).

A majority of developers want to be able to run fuzzers as part of their local development workflow.

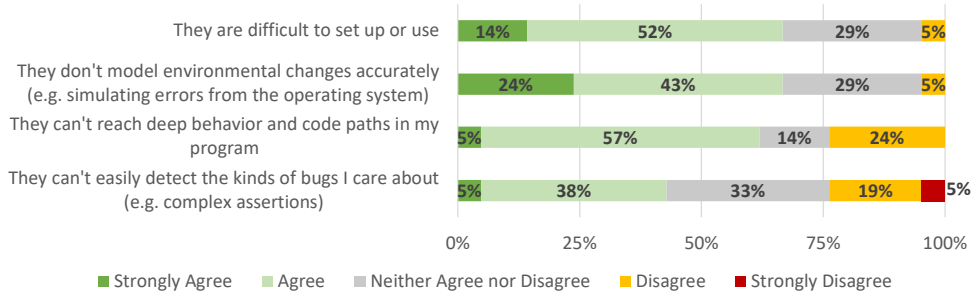


Fig. 4: Experienced participants' choices for reasons that prevent them from using fuzzing.

Infrastructure

We also asked practitioners *"At what scale would you conduct fuzzing campaigns?"*. A large number of our participants (36/44, 57%, $P = .146 > .05$), with a stronger majority of experienced practitioners (14/21, 67%, $P < .05$), would deploy fuzzers on a single machine with one or a small number of cores. This indicates that fuzzing research's emphasis on single or few-core performance is well-justified. Of the remaining participants, roughly an equal number indicated that they would use vertically (many core, single machine) or horizontally scaled (many machines) hardware for their fuzzing campaigns.

Obstacles in Fuzzing

While fuzzing has grown in prominence in recent years, the majority of our participants do not yet use them regularly in their development workflows. To understand why, we asked *"What prevents you from using fuzzers?"*. Most inexperienced participants (14/23, 61%) gave no indication for one or more parts of this question, and we saw no significant trend among the remaining few responses from this group. Therefore, we only discuss the responses from the experienced participants (Figure 4). In agreement with prior research (Nourry et al (2023)), one of the leading barriers to adoption of fuzzing among our experienced participants was usability (14/21, 67%, $P < .05$), Interestingly, modeling environmental changes was seen equally problematic (14/21, 67%, $P < .05$). Further, a large minority of our experienced participants saw reaching deep behavior and code paths as key obstacles as well.

4.2 Fuzzer Setup and Inputs (RQ2)

The next section of our survey investigated developers' tolerance for manual steps to start using a fuzzer, to better establish which steps should be automated or otherwise avoided in fuzzing workflows.

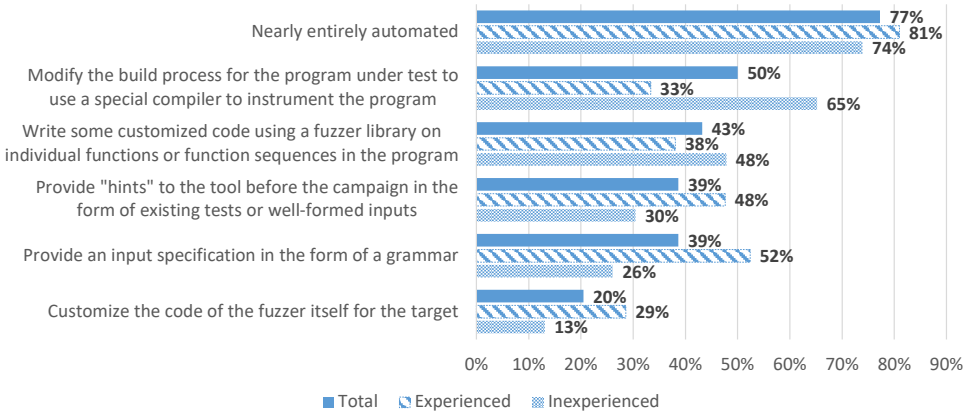


Fig. 5: Participants' willingness to configure and setup the fuzzer.

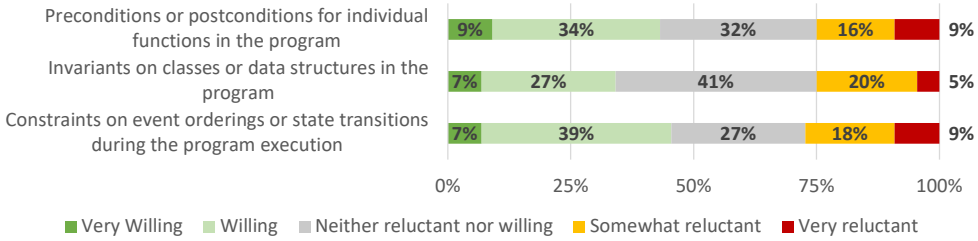


Fig. 6: Participants' willingness to write/provide additional specifications as inputs to fuzzer.

Fuzzer Setup

To understand users' tolerances to various common set-up tasks for fuzzing, we asked our participants *"What level of setup are you willing to do (if any) to configure a fuzzer on a given target?"* (Figure 5). While a small number of participants would only use a nearly completely automated, "push-button" fuzzer, most (38/44, $P < .001$) would be willing to do some amount of manual configuration. For the various tasks associated with the program under test –modifying the build-process, writing a fuzzing harness, providing initial test cases, or specifying an input grammar– a similar proportion of roughly 40% of users are willing to undertake them. Far fewer would consider modifying the fuzzer to conform to a particular target program. We note that this dichotomy may contrast with the fuzzing research community, who are likely more familiar with the fuzzing tools than the programs being tested. While, in general, we could not determine a statistically significant difference between the experienced and inexperienced participants for this topic, we still would like to highlight two aspects: Firstly, while most inexperienced practitioners (15/23, 65%, $P < .05$) would be willing to modify the build process, only a few experienced practitioners (7/21, 33%) would

do the same (difference across groups: $P = .035 > \alpha = .01$). Secondly, most inexperienced participants (17/23, 74%, $P < .01$) would not be willing to provide any input specification; however, the experienced participants were more divided on this question, with 11/21 being positive and 10/21 being negative about providing an input specification (difference across groups: $P = .074 > .alpha = 01$).

Interacting with the Fuzzer

A majority of our participants (25/44, 57%) want to interactively guide a fuzzer towards particular code locations. More interestingly, the same number of users would also like to interactively patch roadblocks that hinder the fuzzer from making progress. This could be facilitated by a visualization of roadblocks faced during a campaign, as recently proposed by (Yan et al (2023)).

Specifications

While the complete specifications needed for verification are often considered too heavy-weight for most developers, in the context of fuzzing, incomplete specifications can still be useful as test oracles. We asked our participants *"How willing are you to write the following kinds of specifications for a program to detect bugs while fuzzing?"* (Figure 6). To ensure participants who are not familiar with specifications were able to give accurate responses, we provided concrete example specifications for context. Of the three possible modes of specifications we included in our survey –temporal properties, data invariants, and pre/post-conditions– none would be willingly written by a majority of developers. Thus, to target the majority, fuzzer authors should continue to focus on using lightweight, implicit oracles (such as Address Sanitizer) rather than requiring users to manually write specifications. There is, however, a substantial portion of developers ($> \frac{1}{3}$) who *would* be willing to write specifications of all three possible modes of specifications we asked about.

The majority of software professionals would not provide even partial specifications to enhance fuzzing. However, there may still be opportunities to cater fuzzers to the $\frac{1}{3}$ minority who are willing to write pre/post-conditions, data invariants, and temporal properties.

4.3 Fuzzer Outputs and Artifacts (RQ3)

This section of our survey examined developer expectations for the outputs of a fuzzing campaign, which can help tool builders understand which output artifacts are most useful for their users.

Bug Relevance

Most of the inexperienced participants (17/23, 74%, $P < .01$) indicated that they cannot comment on the relevance of fuzzer-generated bug reports, so we only discuss responses from the experienced participants, who mostly (19/21, 90%, $P < .01$) commented on this question. In general, a reduction of false alarms in bug reports relative to static analysis tools is often mentioned as one of the primary strengths of fuzzing

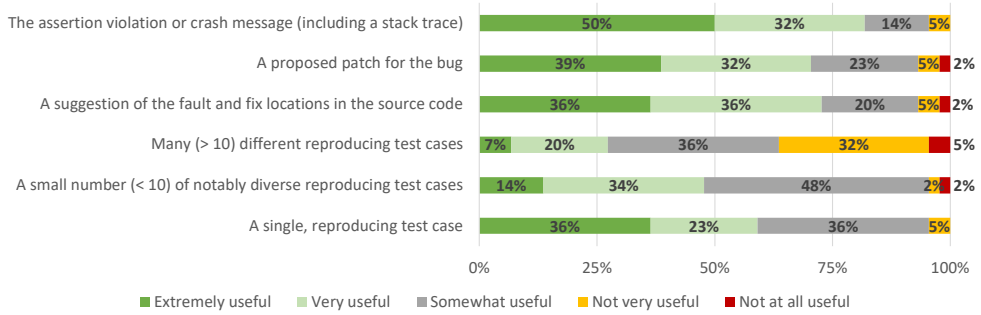


Fig. 7: Participants opinions about features in fuzzer-generated bug reports.

and other dynamic analysis techniques. However, a large group of our experienced respondents familiar with such reports (10/19, 53%) indicated that fuzzer-generated bug reports are either mostly irrelevant or at least contain a mix of irrelevant and relevant findings. Thus, while better positioned than static-analysis, fuzzing-generated reports still require triage in most cases. Automatically identifying false alarms in fuzzer-generated bug reports could be an impactful direction for future research.

Usefulness of Bug Reports

Figure 7 shows user opinions on the difficulty of fixing fuzzer-reported bugs relative to user-reported bugs. Overall, we cannot say that users find bugs reported by fuzzers are any easier or harder to fix than user-reported bugs. The majority of the inexperienced participants did not know (13/23, 57%), and the experienced participants had no clear trend towards a specific direction. After showing participants examples of fuzzer-generated bug reports, we also asked *"How useful do you think each feature of a fuzzer-generated bug report would be?"*. Unsurprisingly, most participants (36/44, 82%, $P < .001$) mention it is very useful to have the identified assertion violation or crash message, including a stack trace. A large majority also indicated that suggested fault and fix locations (32/44, 73%, $P < .001$) or a proposed patch (31/44, 70%, $P < .01$) for the bug would be beneficial. Interestingly, the responses from experienced and inexperienced practitioners were mainly consistent, indicating the relevance of the obtained feedback about fuzzer-generated bug reports for general software development.

Additional artifacts such as suggested fix locations or proposed patches are **more desirable than a reproducing test case** for automated bug reports.

Outputs

Participants mentioned the integration of fuzzers into their existing development environments, pipelines, and tools. For example, automated issue reporting with tools like Sonarqube and Jira. Others mentioned ideas to further improve the information in bug reports by adding the execution trace with the state values of the reproducing test case, the relevant bytes in the crashing input that differentiate (w.r.t. the execution behavior) from other similar but non-crashing inputs, runtime profiling (memory

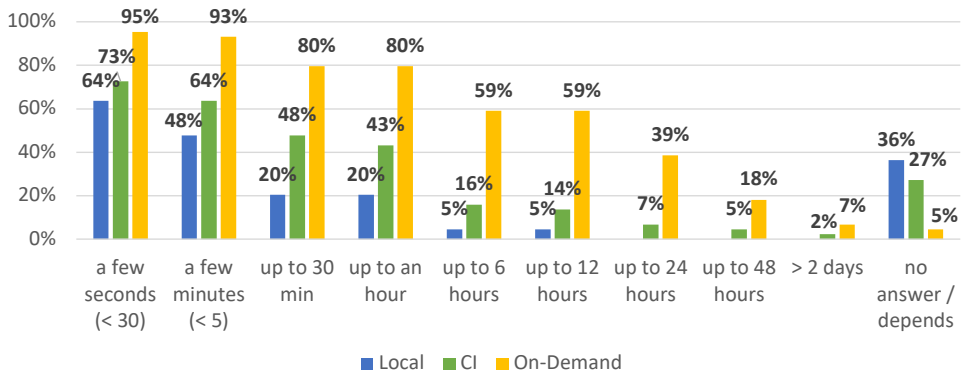


Fig. 8: Cumulative presentation of the expected fuzzing campaign lengths for the different workflow types.

usage and timing information) of each function that is executed along with the reproducing test case, and any additional information the system under test produced as part of the execution (logs, error messages, etc.).

Generated Test Cases

The readability and interpretability of fuzzer-generated test cases was regarded as very important by most participants (38/44, 86%, $P < .001$). Yet there are few tools available for understanding generated tests. Additionally, while most existing automated debugging techniques focus on minimization[5], minimality only ranked third in importance among our respondents.

Our participants (consistently across the two groups) prefer automatically generated test cases which have more human readable characters (20/44), followed by test cases which are similar to user-provided "well-formed" inputs (15/44). Future research can focus on maximizing the *interpretability* of generated test cases.

Interpretability **is more important than minimality** for fuzzer-generated test cases.

4.4 Fuzzer Evaluations (RQ4)

Developer expectations also provide important to establish evaluation criteria for fuzzing research, which is of great interest to the fuzzing research community (Böhme et al (2020); Liang et al (2018)).

Evaluation Configurations

Generally, fuzzer evaluation configurations should mirror real-world use cases, such as those identified in Section 4.1. We queried participants about the expected fuzz campaign lengths for three fuzzing workflows: on-demand audits, CI/CD fuzzing, and

[5] https://github.com/google/AFL/blob/master/afl-tmin.c

fuzzing in the local development cycle. Figure 8 shows their aggregate responses. While we observed that experienced practitioners could be willing to wait a bit longer, our analysis could not determine a significant difference between the two groups. Therefore Figure 8 shows the responses over all 44 participants. For an on-demand fuzzing campaign, 39% (17/44) of all participants would wait up to 24 hours, with a sharp drop to only 18% of participants willing to wait 48 hours or more. There are similarly sharp declines from 1 hour to 6 hours, and between 12 and 24 hours. Still, the general recommendation of 24 hours for fuzzing campaigns appears to be in-line with developer expectation for on-demand fuzzing campaigns.

In contrast, participants prefer shorter time bounds for CI fuzzing. Only a few of the participants (7/44, 16%) were willing to run a CI fuzzing job for more than one hour. During local development, even fewer of the participants would wait longer than an hour for a fuzzing run, while most would not wait more than five minutes.

For the strongly desired local and CI/CD fuzzing, the expected timeouts are substantially shorter than the usual evaluation timeouts suggested in the fuzzing literature (Klees et al (2018)). This inspired us to perform follow-up experimentation to assess current state-of-the-art fuzzers under these new expectations (see Section 5).

To meet developer expectations, evaluations of fuzzers should timeout at:
- 5 minutes, for local development
- 1 hour, for CI/CD fuzzing
- 24 hours, for on-demand audits

Evaluation Metrics

While bug-finding ability is a primary goal of any fuzzer, there are many ways in which this effectiveness can be measured. To understand which of these measures are preferred by practitioners, we asked *"When evaluating fuzzers on a benchmark suite, which metric is most important?"*. A strong plurality of our respondents (20/44, 45%) indicated that the number of *exploitable* bugs is the most important evaluation metric. 32% (14/44) of our participants indicated that the quantity of bugs found was most important, with relatively few remaining participants selecting other options. This top-2 ranking of the most important ranking for fuzzing evaluation was indeed consistent across experienced and inexperienced participants.

For our participants, the severity –followed by quantity– of bugs found are more important metrics than difficult-to-discover or rare bugs.

While users indicated that bug-finding ability, particularly for exploitable bugs is an important metric for fuzzers, ground-truth evaluations can be difficult to conduct due to the relative sparsity of bugs. To generate benchmarks for these evaluations, our participants mostly would trust artificial bugs that are automatically backported from a different version of the program under test (29/44, 66%, $P<.05$). Overall, our hypothesis testing did not conclude any significant difference between the two groups; however, we would still want to highlight the following observations: The majority of the experienced participants (15/21, 71%, $P < .05$) follow the trend of trusting automatically backported bugs most, followed by bugs generated based on known bug

patterns (9/21) and transplanted bugs from similar software (9/21). The inexperienced participants are less clear in their decision and equally often (14/23, 61%) mentioned backported bugs, bugs generated based on known bug patterns, and manually crafted bugs. Somewhat fewer would trust evaluations on bugs generated by simple random mutations, indicating that researchers may need to work to increase practitioner trust in techniques such as mutation testing (Just et al (2014)).

When ground-truth bugs are not available, several alternatives are trusted by a majority of participants, though bugs generated by random program mutations may need to bridge a larger trust gap with practitioners.

4.5 Summary

We summarize our findings from this user survey in the table below:

4.1: Workflows and Use Cases	
Types of Bugs	Diverse fuzz-testing oracles are desired
Testing Goals	Regression-test generation, bug discovery, and directed testing are equally important
Deployment Scenarios	Typical deployments of fuzzers are on a single-machine, with low parallelism
Workflows	Local Development and CI/CD workflows are desired for fuzzing
Obstacles	UI/UX is the greatest obstacle to adoption of fuzzing
4.2: Setup and Inputs	
Setup	Developers are somewhat willing to tolerate setup costs for fuzzing
Interaction	Users desire interactive workflows with fuzzers that can take guidance from a human
Specifications	Developers are mixed on writing specifications, but the majority are unwilling to do so
4.3: Outputs and Artifacts	
Bug Relevance	False positives and noise are still problematic for the deployment of fuzzers
Usefulness of Reports	Fuzzer-generated bug reports are about as useful as other bugs
Report Artifacts	Fault localization and proposed patches are more important than a reproducing test case
Other Outputs	Developers mentioned integrations with issue-tracking platforms as desired
Generated Test-cases	Interpretability of fuzzer-generated test cases is highly important
4.4: Evaluations	
Evaluation Configurations	We identify empirical thresholds for fuzzing cutoff times that meet developer expectations
Bug-Finding Evaluations	Severity of bugs found is the most important criteria for fuzzer evaluation
Alternative Metrics	Alternatives to ground truth bugs are generally trusted by developers

5 Evaluation of Fuzzers in a Commit Fuzzing Scenario

As continuous local and CI/CD fuzzing were both strongly desired workflows for our survey participants, we conducted an additional quantitative analysis to assess the capabilities of fuzzing tools in this context. We evaluate how commit, regression, directed, and general purpose fuzzers can find existing regression bugs in previous versions of real open-source software projects. We explore whether state-of-the-art fuzzing techniques can detect regression errors under strict configurations that match developer expectations and requirements. Specifically, we provide answers to the research question RQ5.

Fuzzer ↓	Program → / Issue # →	yara 48329	libtiff 58729	proj4 49256	zstd 57086	libxml2 57469	file 59438	haproxy 52049	libvpx 48816	usrsctp 47712	openvswitch 47112
AFL	t_{build} (s)	18.65	42.50	110.36	107.21	22.12	28.35	84.19	7.48	12.36	31.36
	P_{5m}-sat.	0.00	1.00	0.00	0.00	0.00	1.00	0.95	0.00	1.00	0.60
	P_{1h}-sat.	0.00	1.00	0.00	0.25	0.00	1.00	1.00	0.00	1.00	1.00
	P_{5m}-static	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.05
	P_{1h}-static	0.00	1.00	0.00	0.05	0.00	1.00	1.00	0.00	1.00	0.55
AFLchurn	t_{build} (s)	36.86	58.72	119.25	142.16	53.50	73.73	1168.09	12.03	-	150.93
	P_{5m}-sat.	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	-	0.90
	P_{1h}-sat.	0.00	1.00	0.00	0.20	0.00	1.00	1.00	0.00	-	1.00
	P_{5m}-static	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	-	0.05
	P_{1h}-static	0.00	1.00	0.00	0.00	0.00	1.00	1.00	0.00	-	0.40
AFLgo	t_{build} (s)	80.97	127.14	487.35	-	73.07	57.35	198.41	34.98	-	108.66
	P_{5m}-sat.	0.00	1.00	0.00	-	0.00	1.00	0.85	0.00	-	0.90
	P_{1h}-sat.	0.00	1.00	0.00	-	0.00	1.00	1.00	0.00	-	1.00
	P_{5m}-static	0.00	1.00	0.00	-	0.00	1.00	0.00	0.00	-	0.00
	P_{1h}-static	0.00	1.00	0.00	-	0.00	1.00	1.00	0.00	-	0.45
AFL++	t_{build} (s)	101.20	104.93	241.39	216.15	53.33	65.95	194.88	15.46	30.67	70.17
	P_{5m}-sat.	0.00	1.00	0.00	0.00	0.00	1.00	0.25	0.00	1.00	0.40
	P_{1h}-sat.	0.00	1.00	0.00	0.05	0.00	1.00	1.00	0.00	1.00	1.00
	P_{5m}-static	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.05
	P_{1h}-static	0.00	1.00	0.00	0.05	0.00	1.00	1.00	0.00	1.00	0.35
CIDfuzz	t_{build} (s)	-	-	-	-	923.58	80.65	-	-	-	-
	P_{5m}-sat.	-	-	-	-	0.00	1.00	-	-	-	-
	P_{1h}-sat.	-	-	-	-	0.00	1.00	-	-	-	-
	P_{5m}-static	-	-	-	-	0.00	1.00	-	-	-	-
	P_{1h}-static	-	-	-	-	0.00	1.00	-	-	-	-
Selectfuzz	t_{build} (s)	107.06	952.77	2670.94	-	133.40	67.30	219.56	102.30	-	207.92
	P_{5m}-sat.	0.00	0.00	0.00	-	0.00	1.00	0.10	0.00	-	0.55
	P_{1h}-sat.	0.00	1.00	0.00	-	0.00	1.00	1.00	0.00	-	1.00
	P_{5m}-static	0.00	0.00	0.00	-	0.00	1.00	0.00	0.00	-	0.05
	P_{1h}-static	0.00	1.00	0.00	-	0.00	1.00	1.00	0.00	-	0.30

Table 1: Probability of Bug Discovery for Commit-level Fuzzing of Open Source Programs with Time Limit *including build times* of 5 minutes (P_{5m}) and 1 hour (P_{1h}). Results shown for using a saturated (-sat.) or static (-static) corpus.

5.1 Experiment Setup

Fuzzers

For our evaluation, after a literature review in total we select six state-of-the-art fuzzers that met our criteria representing three different avenues explored in the literature, namely commit fuzzing (CIDFuzz by Zhang et al (2023)), regression fuzzing (AFLChurn by Zhu and Böhme (2021)), directed fuzzing (AFLGo by Böhme et al (2017) and SelectFuzz by Luo et al (2023)), and general fuzzing (AFL and AFL++ by Fioraldi et al (2020)). Our selection criteria for directed, commit and regression fuzzers was that they (1) were compatible and able to run in Fuzzbench (Metzman et al (2021)) infrastructure (2) that the code for the fuzzer was openly available and (3) that it was compatible with both C and C++ programs. We included all such fuzzers we identified that met this criteria. We selected AFL and AFL++ as the baseline comparisons for our evaluation as state-of-the-art general-purpose fuzzers because all fuzzers in our evaluation are based on the AFL fuzzing algorithm and implementation. We compare against AFLGo and SelectFuzz, which attempt to generate test inputs that reach target locations in a program. For the program locations, we use the changed lines in the buggy commit as targets. Finally, we choose AFLChurn and CIDFuzz as regression fuzzers specifically designed to detect bugs in a commit of a

software program. We configured each fuzzer to use default parameters in our benchmarking setup according to its respective documentation or openly available example configurations in a best-effort manner. Based on our survey results in Sections 4.1 and 4.2, fuzzer usability is the number one obstacle for software professionals to use fuzzers and the majority of developers are not willing to extensively customize fuzzer configurations to their systems; thus we believe this setup reflects how practitioners would utilize these tools in their own workflows.

Subject Programs

We randomly selected 10 bugs from OSS-Fuzz (Serebryany (2017)) that are (1) labeled as reproducible, (2) were able to be built and run in the FuzzBench infrastructure, and (3) where we were manually able to reproduce the bug and identify the bug-introducing commit. We sampled bugs starting from the front page of the bug tracker[6] at the time of experimentation, up to one regression bug per program that met criteria (1), (2), and (3), moving to the subsequent page if no bugs met our criteria. We sampled up to 10 bugs, with no more than one bug per project. As a result, all our subjects are C/C++ programs from open source projects integrated with the OSS-Fuzz infrastructure.

Fuzzing Setup

For our evaluation setup, we seek to emulate frequent, cumulative fuzzing on a given target in response to code changes, as in a CI/CD pipeline or in a local development workflow. Thus, we create a fuzzer-generated corpus on the program under test *before* the bug-introducing commit which we call the *saturated* corpus. This saturated corpus emulates a fuzzer-created test suite built up over many iterations of successive, change-based fuzzing campaigns during the lifetime of a project. To generate the saturated corpus, we fuzzed the pre-bug commit version of each program continuously for two weeks each with AFL and AFL++. We then aggregated all test cases saved by the two-week campaigns and used `afl-cmin`[7] to minimize them with respect to edge coverage. `afl-cmin` is a utility program that greedily constructs a subset of test cases to achieve the total coverage of all test cases. In our evaluation of fuzzers, we use this minimized corpus as the initial seeds for each program.

As an additional baseline, we also evaluated each fuzzer with a corpus comprised *only* of test cases provided in the project source repository. For each project, we used the OSS-fuzz configuration and project documentation to locate these test cases, taking only the test cases which were available prior to the bug introducing commit. We call this the *static* corpus.

Hardware

All experiments were conducted using the standard benchmarking service FuzzBench (Metzman et al (2021)) via Docker containers on a 112-core 2.70GHz 191G RAM Intel Xeon (Gold) machine.

[6] https://issues.oss-fuzz.com/issues
[7] https://github.com/google/AFL/blob/master/afl-cmin

Evaluation Metrics

We assess both the efficacy and efficiency of the fuzzers in finding regression errors. We manually analyzed the crash signatures recorded by Fuzzbench and determined if they corresponded to the vulnerability in question. Three of the authors independently checked the crash signatures to determine whether they corresponded to the vulnerability in question.

To analyze the results we use *survival analysis* because our bug-finding trials are *right-censored* data. Specifically, we use the Kaplan and Meier (1958) method. For some additional claims of statistical significance, we use Wald's t-test (Wald (1943)) on the bootstrapped Restricted Mean Survival Time (RMST) of the fitted Kaplan-Meier models. The RMST captures the area under the curve of a survival plot, thus a *higher* RMST indicates that a fuzzer is *less effective* because the bug "survives" longer.

Results Table Description

We collect results for each of the fuzzers over 20 repetitions and analyzed the efficacy of finding the target regression error. Table 1 summarizes the efficacy of each fuzzer for timeouts of 5 and 60 minutes (see Section 4.4) and the build time for each project. For each sub-column, P_{5m} and P_{1h} represent the probability of the fuzzer detecting the target regression errors in twenty trials for time duration of 5 minutes and 60 minutes, respectively. P_x-static represents these probabilities when using the static corpus, rather than the saturated corpus (P_x-sat.). Sub-column t_{build} depicts the average time in seconds needed for building the subject for each fuzzer over 20 repetitions. Note that t_{build} –which captures the time taken for each fuzzer's instrumentation and/or other inital program analyses– is *included* in the computation of P_{5m} and P_{1h}. In otherwords, a trial is considered successful if the fuzzer can both build, instrument and analyze the project *and* find the bug within the target timeframe.

5.2 (RQ5) Effectiveness of Existing Fuzzers

Of the ten bugs in our evaluation, four bugs –in `libvpx`, `proj4`, `yara` and `libxml2`– were not found by any of our evaluated fuzzers in the one-hour time limit.[8] However, out of the remaining six programs, we find that the static corpora *or* a saturated corpus of test cases generated by fuzzing *earlier* versions of the program was able to trigger the bug in three programs: `file`, `usrsctp` and `libtiff`. These bugs were triggered by existing inputs without *any* additional fuzzing on the bug-introducing commit itself.

With respect to our identified thresholds of *5 minutes* for local fuzzing and *1 hour* for CI/CD fuzzing, we see mixed results when using a saturated corpus. Table 1 shows the discovery rates for each fuzzer on each bug at each duration. Here, we observe that two of the bugs are discovered at least once within the first five minutes. Mostly, the probability of finding these bugs within 5 minutes is relatively high as well (e.g., geomean of 0.75 for AFL). At an hour, prospects improve somewhat for finding the bug in `zstd`, but still, no fuzzer can find the bug in the majority of trials. For the other programs in which bugs can be found, increasing the fuzzing campaign up to one hour nearly ensures the bug will be discovered.

[8] In preliminary testing, some fuzzers were able to find the bugs in `libxml2` and `yara` within one hour with very low probability using the saturated corpus, but this did not occur in our final evaluation with 20 trials.

Existing fuzzers can already have a substantial impact if integrated into change-based developer workflows, finding half of all bugs at least once within expected time frames for local development (5/10) and CI/CD fuzzing (6/10). Future research in will be needed to find the remaining bugs within these shorter time limits more consistently.

Comparison of Fuzzers

We show the Kaplan and Meier (1958) fitted survival plot for all programs where there are differences in effectiveness across fuzzers in Figure 10**.(middle)**. The survival plots show the probability that a bug has *not* been found by the corresponding fuzzer over time. Across these programs, we don't see any trend in terms of fuzzer superiority; no one fuzzer appears to be significantly better or worse than the others at finding this regression bug. We do observe that AFL consistently beats AFL++, despite the latter being a re-implementation of the former with additional features ($RMST_{5m}$: 226.85 $AFL{+}{+} \rightarrow$ 208.34 AFL, $p < 0.001$ and $RMST_{1h}$: 2099.02 $AFL{+}{+} \rightarrow$ 2026.42 AFL, $p < 0.001$). This is unusual, as AFL++ is generally meant to be an improved implementation of AFL, but not unheard of; AFL also beats AFL++ occasionally in public benchmarking results[9]. We could not determine clearly that directed or commit/regression fuzzers perform better in this context relative to the general-purpose fuzzers. This result is surprising, as directed and commit/regression fuzzers are optimized for reaching particular code target locations quickly. Unfortunately, all of the directed and regression fuzzers are incompatible with one or more of the programs in our evaluation set, especially in the case of CIDFuzz.[10] However, despite CIDFuzz compatibility issues, it does attain the statistically significant highest *code* coverage for the two programs it can successfully fuzz.

5.3 (RQ6) Impact of Additional Variables on Fuzzer Effectiveness

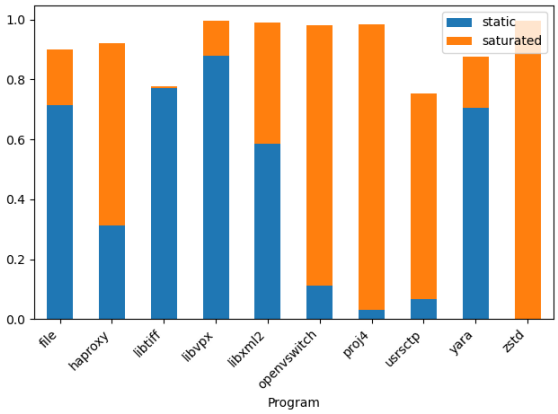
Comparison of Corpora

For this experiment we used two different starting corpora. One of which is saturated in that it is distilled from the historical corpus of OSS-Fuzz runs and thus is expected to be high coverage. In contrast, the static corpus consisted of existing test cases extracted from each target program's repository. Figure 9**.(left)** shows the LLVM edge coverage of each corpus, normalized to the maximum coverage observed during fuzzing.[11] Here we can see that the saturated corpus is indeed, very high coverage in all cases. However, the static corpus coverage varies widely, with some examples less than 10% of the maximum observed coverage and other examples having nearly the same coverage as the saturated corpus. Figure 9**.(right)**, we can see the distribution of corpus sizes across programs. Here again, we see a wide variance across projects, with

[9] https://www.fuzzbench.com/reports/2021-06-02/index.html#mbedtls_fuzz_dtlsclient-summary

[10] CIDFuzz's LLVM instrumentation pass causes segmentation faults on most programs. The build process for `zstd` is complex, and would need to be heavily modified to incorporate most AFLGo-based fuzzers' instrumentation steps. Similarly, `usrsctp` has a library versioning conflict with several AFLGo-based fuzzers, which results in a linker error on instrumentation.

[11] Note that for `zstd` the LLVM coverage calculator failed to generate a coverage report, likely due to its excessively large size. Thus we are missing the initial coverage for the static corpus of this target program.



Program	# Test Cases	
	Saturated	Static
file	386	74
grok	442	36
haproxy	87	40
libtiff	940	7367
libvpx	155	161
libxml2	377	40
openvswitch	2249	24
proj4	1555	133
usrsctp	25	24
yara	121	7
zstd	782	24223

Fig. 9: **(Left)** Initial Corpus Coverage Normalized to Maximum Observed Coverage per Program and **(Right)** Initial Corpus Size (Number of Test Cases)

some projects like `libvpx` having more test cases than the saturated corpus, despite having lower initial coverage.

Looking at Figure 10.**(top)** and **.(middle)**, we can compare the bug-finding efficiency of our fuzzers when using each corpora. Here we see that fuzzers using either corpus are able to find all three bugs in at least some cases, but, generally, the bugs are found more quickly and reliably with the saturated corpus. For example, on `openvswitch`, all fuzzers find the bug within one hour when using the saturated corpus, but none of the fuzzers can find the bug in the majority of trials when the static corpus is used instead. Similarly, both AFL and AFLGo are able to find the bug in `haproxy` within 5 minutes in the majority of trials with the saturated corpus, but *never* find the bug in less than 5 minutes using the saturated corpus. Across all benchmarks, there is a significant decrease in effectiveness when the static corpus is used instead of the saturated corpus ($RMST_{5m}$: 239.13 *static* $\rightarrow$ 206.80 *saturated*, $p < 0.001$ and $RMST_{1h}$: 2324.71 *static* $\rightarrow$ 1940.65 *saturated*, $p < 0.001$).

A high coverage corpus generated from cumulative fuzzing is generally more effective at finding new bugs that the static test cases included within source repositories for OSS-Fuzz. Practitioners should consider using an updated, cumulative corpus for CI/CD fuzzing. Future research in test case selection for local development and CI/CD fuzzing, such as (Canakci et al (2022)), could improve fuzzer efficacy.

Build Times

An often overlooked aspect in research papers is the *build time* overhead of a given tool. This number includes the time it takes a fuzzer to analyze and instrument a target program *before* it starts fuzzing. In the context of 24-hour fuzzing campaigns, this additional instrumentation and analysis time may be negligible. However, for local development or CI/CD fuzzing, with the total campaign length being only a few minutes to an hour, the additional build times can be substantial. Indeed, looking at

the columns t_{build} in Table 1, we can see a large difference between fuzzers. Selectfuzz takes *ten times* the build time of AFL on average! Even AFL++ or AFLGo takes 80 to 90 seconds longer than AFL to instrument programs on average; in the context of a local development fuzz campaign, this is equivalent to more than 20% of the total time budget.

We can see the impact on fuzzer effectiveness for our identified timing thresholds of 5 minutes and 1 hour in Figure 10. Comparing the **(middle)** and **(bottom)** rows, we can see the difference in survival plots for the saturated corpus are included and excluded, respectively. Here we can see a significant visual difference for some fuzzers: AFLchurn goes from being being among the worst fuzzers on the `haproxy` benchmark program with build times included **(middle)**, to being among the best if we ignore build times **(bottom)**. Indeed overall, we see a statistically significant drop in effectiveness across all fuzzers and benchmarks ($RMST_{5m}$: 234.31 *excluded* $\rightarrow$ 200.12 *included*, $p < 0.001$ and $RMST_{1h}$: 2154.33 *included* $\rightarrow$ 2076.55 *excluded*, $p < 0.001$).

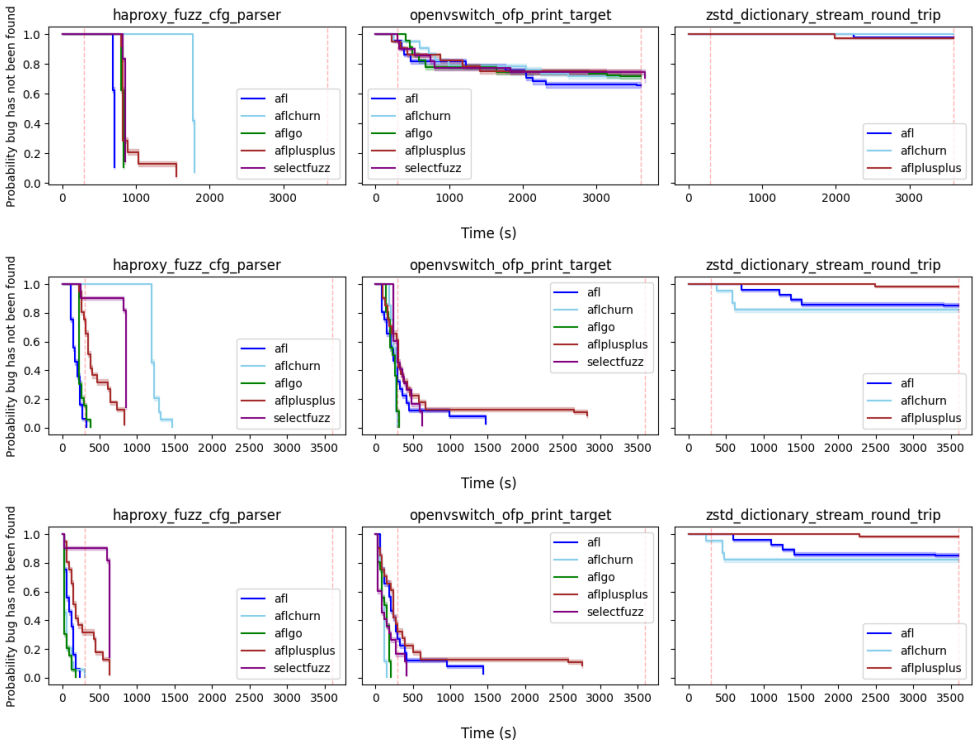


Fig. 10: Probability that the regression bug **is not found** by each fuzzer over time (lower is better). **(top)** using static corpus; **(middle)** using saturated corpus; **(bottom)** using the saturated corpus *excluding* build times

Build-time overhead introduced by fuzzers can have a significant impact in shorter campaigns; regression fuzzer developers should emphasize approaches which do not drastically increase build times for real-world software projects.

6 Threats to Validity

6.1 Developer Survey

Construct

Experimenter bias is always a threat to the validity of personal opinion surveys. We attempted to phrase all of our questions in neutral language, but it is possible that aspects of our survey instrument may have biased the results. To avoid observer bias, we designed the survey to collect no personally identifying information, giving our participants a trusted platform.

Internal

To ensure the clarity of our questions and the form's structure, we performed small-scale test runs with researchers who were not involved in the study.

The overarching goal of our study was to assess the expectations of software professionals for a usable fuzzing workflow; to ensure our conclusions reflect this, we blocked our data on fuzzing experience and discarded responses from academic researchers. We choose to include responses from those unfamiliar with fuzzing because we are explicitly interested in how these users see themselves using a fuzzing tool. While there is a risk that this population are not able to accurately answer questions, our target population is nonetheless highly technical (Section 3) and we provided a general summary of fuzzing concepts at the beginning of our survey form to ensure our participants had the requisite background information.

Conclusion

To reliably interpret all responses, particularly the open-text responses, we applied qualitative analysis coding with an agreement of at least two authors on all codes. For questions measuring agreement, we used the chi-square test of independence (Pearson (1900)) and the binomial test (Dodge (2008)), both well-established statistical tests in the realm of user surveys (Hilton et al (2017); Gorski et al (2018); Javed et al (2019); Votipka et al (2020); van der Linden et al (2020); Noller et al (2022)).

External

A potential threat to external validity is the limited sample size and scope of participants in our survey. By collecting responses using the snowball method (see Kitchenham and Pfleeger (2008)) from 44 participants across 23 companies and 11 countries, which we see as a sufficiently diverse set of participants, but we cannot guarantee the generalization of our results. We believe that we were able to reach an important sample cohort because they mostly identify as highly technical software developers who have only limited experience with fuzzing while being familiar with the concept (Section 3).

6.2 Commit Fuzzing

Construct

To ensure construct validity, we directly measure our criteria for fuzzer effectiveness –ability to find bugs associated with a code change– rather than using a proxy metric, such as code coverage. Because crashes from different bugs can look similar, three of the authors examined the crash signatures for each crash detected during fuzzing to determine whether they corresponded to the targeted bug.

Internal

To avoid selection bias in our choice of benchmarks, we chose them randomly from real-world bugs in the OSS-Fuzz (Serebryany (2017)) bug tracker. More specifically, we sampled randomly from the front page of the OSS-Fuzz bug tracker at the date of experimentation, filtering for confirmed and reproducible bugs. It is possible that there is a temporal bias, as our bugs all come from a similar timeframe, but we sampled only up to one bug per project to minimize this effect.

We attempted to avoid selection bias in our choice of fuzzers by using all openly available directed, commit and regression fuzzers that we were able to make compatible with the Fuzzbench benchmark suite. To identify these fuzzers, we searched for these keywords (along with fuzzing/fuzzer) in research databases such as Google Scholar and DBLP. We used all fuzzers we identified that met our criteria. We chose AFL and AFL++ as baselines to reduce the risk of cross-implementation differences in fuzzers, rather than fundamental algorithmic differences (e.g. directed vs. undirected fuzzing); all of the evaluated fuzzers' implementations are modifications or extensions of the AFL fuzzer.

Another possible threat to internal threat to validity is three of the bugs were originally discovered by AFL, which might give AFL an advantage in our experiment. However, for two of these three bugs (`yara` and `file`) we see no difference in fuzzer effectiveness. For the last bug found by AFL, `haproxy`, there are differences in performance, but all fuzzers except for one (Selectfuzz) find the bug in 5 minutes excluding build times. Additionally, because we evaluated *only* fuzzers based on the AFL fuzzer implementation, we believe this effect to be minimal. We leave a larger follow up study with additional bugs found by both AFL and LibFuzzer which could isolate this effect to future work.

Conclusion

In general, we believe our experimental setup is in accordance with the latest standards for fuzzer evaluations (Klees et al (2018); Kim et al (2024)).

To avoid issues with misinterpreting our results, we use statistical techniques that are well-established across multiple fields of research. As bug-finding benchmarks are *right-censored* data, we use survival analysis (Kaplan and Meier (1958)). To determine statistical significance between fuzzers on individual benchmarks for a single configuration, we use the confidence intervals given by our survival analysis (Kalbfleisch and Prentice (2002)), which can be seen in Figure 10. When comparing across benchmarks and fuzzers, we use the bootstrapped Restricted Mean Survival Time (Royston and

Parmar (2013)) and Wald's t-test (Wald (1943)). For bootstrapping, we use 100 iterations of case-resampling the individual trials of each experiment. For other claims of significance, we compare build times with Wald's t-test and code coverage for CID-Fuzz with Fuzzbench's built in Mann-Whitney U test (Mann and Whitney (1947); Wilcoxon (1945)).

To validate model assumptions, we check for normality visually using a QQ plot before applying Wald's t-test, available in our artifact (Section 8).

Notably, we choose not to use two common analyses for censored data –the log-rank test and Cox-regression– because our data appears to violate the proportional hazards assumption (Figure 10).

External

We follow the recommendation of Klees et al (2018) that ≥ 10 benchmark programs should be used for fuzzer evaluations. As with any empirical study, more subject programs and fuzzers would improve the generalizability of our results. However, verifying each bug's reproducibility, the introducing commit, integrating OSS-Fuzz benchmark programs, and generating a large initial corpus all take significant manual effort. During the course of this experiment, we attempted to set up a total of 25 OSS-fuzz programs using Fuzzbench's integration with OSS-Fuzz. However, of these programs and bugs, most could not be used: seven resulted in build errors that we were not able to debug, three had errors related to fuzzer instrumentation passes, three we could not reproduce reported bugs at the commits indicated in the bug report, and the remaining two had issues with several dependent libraries which also needed to be built as specific commits to reproduce past bugs. We believe that this study is still valuable at its current size, especially when viewed with complementary works such as Zhu and Böhme (2021); Zhang et al (2023); Kim et al (2024); Klooster et al (2023) and we ensured that each of our configurations had sufficient trials ($n = 20$) to generalize for the bugs and subject programs we evaluated.

7 Related Work

Fuzzing User Surveys

Böhme et al (2020) discuss the challenges in fuzzing identified by expert researchers and users at a recent Shonan meeting. Similarly, Nourry et al (2023) constructed an extensive taxonomy on the specific challenges faced by fuzzing experts. In this work, we identify some key challenges in fuzzing which largely agree with some findings in these prior publications. Namely, (1) that usability issues are the largest impediment to adoption among developers (Nourry et al (2023); Böhme et al (2020)) (2) that users of fuzzers prefer shorter time budgets than those typically seen in academic literature (Nourry et al (2023)), that (3) these users also want to be able to detect classes of bugs not easily detectable by existing fuzzers and sanitizers (Böhme et al (2020)), and that (4) most developers want human-in-the-loop, interactive features in their fuzzers (Böhme et al (2020)). However, we also provide several novel findings not found in these prior studies, specifically that a majority of software professionals want to run fuzzers as part of their local development workflow, that software professionals are

more amenable to modifying their systems to be fuzzed rather than adapting an existing fuzzer for their system, that interpretability is more important than minimality for generated test inputs, that many alternatives to ground-truth bugs are trusted by software professionals, that bug severity is more important than bug-quantity in fuzzer evaluations and additionally we establish novel, user-expectation based timing bounds for fuzzers in different workflows. For a comprehensive list, see Section 4.5. More generally, the questions from our study are not focused on challenges in *existing* fuzzer workflows, but rather on desired use cases to bring fuzzing to *new* users. Nearly all participants from Nourry et al (2023) and Böhme et al (2020) are deeply ingrained in existing workflows, having either attended a small conference for fuzzing researchers or participated in online discussions about OSS-Fuzz, whereas our users bring a fresh perspective on adaptations needed to drive adoption in fuzzing. Also, unlike these two works, we conduct a follow up study based on key outcomes from our survey results.

Empirical User Studies of Automated Test Generation Techniques

Other researchers have conducted fuzzer usability studies with students (Plöger et al (2021); Plöger et al (2023)) and in industrial contexts (Liang et al (2018)). We identified usability as the largest obstacle to fuzzer adoption. These studies are complementary to our survey in that they provide insights into the specific usability challenges that users encounter with fuzzers. Indeed, while a substantial portion of developers in our study (43%) indicated that they are willing to write a custom fuzzing harness, in practice, many users struggle with this task using existing fuzzing tools (Plöger et al (2021); Plöger et al (2023); Liang et al (2018)). Future research in automatic fuzz-harness generation (Kelly et al (2019); Babić et al (2019)) could mitigate these issues in practice. Likely a combination of improving these usability issues identified by other work and meeting expectations of software professionals will be needed to shift fuzzers left in development workflows.

Fraser et al (2015) conducted a user study for the closely related technique of automated *unit test* generation with EvoSuite (Fraser and Arcuri (2011)). They found that bug detection was *not* easier for developers using automatically generated unit tests, despite a significant increase in code coverage from these tests. Our survey has a similar goal, in that we are attempting to identify how to make automated testing tools (in this case, fuzzers) a useful component of the sofware lifecycle. We conduct an empirical evaluation as a follow up (Section 5). However, we did not evaluate the usability of fuzzers for developers (as has been discussed in aforementioned prior work), but rather the efficacy of available fuzzers for a novel application, namely commit fuzzing in CI/CD systems.

Regression and Unit Fuzzing

Dynamic analysis has emerged as one of the primary ways of combating regression bugs (Braz et al (2022); Memon et al (2017)), which may comprise as much as 77% of vulnerabilities in open source projects (Zhu and Böhme (2021)). As a result, many researchers and industry leaders have begun to investigate and use fuzzers in local and CI/CD settings. Google has created CIFuzz[12] for continuous integration fuzzing for

[12] https://google.github.io/oss-fuzz/getting-started/continuous-integration/

OSSFuzz and other projects. Other companies like Gitlab[13] and CodeIntelligence[14] have similar offerings. A case study of CI fuzzing in the Linux kernel has even been recently conducted (Shi et al (2019)). From academic research, regression fuzzers target recently changed code with modified basic-block distance-to-change based power schedules (Zhu and Böhme (2021); Zhang et al (2023)) or in combination with shadow symbolic execution (Noller et al (2020)). These tools are similar in concept and design to directed fuzzers (Böhme et al (2017); Luo et al (2023); Marinescu and Cadar (2013)), which target one or more code locations. Our follow-up study evaluates the current abilities of general purpose, directed and regression fuzzers, for commit-level fuzzing.

Fuzzer Evaluation Configurations

Past recommendations for evaluation configurations have primarily been derived from observations of existing fuzzer capabilities (Klees et al (2018); Herrera et al (2021)). In contrast, we ask developers a priori how much time they would be willing to allocate to fuzzing tools in various contexts. We believe that these user expectations can be combined with practical limitations to provide more representative guidelines for evaluations in the future. Several other evaluations of fuzzers in the context of regression or directed fuzzing have been conducted in parallel with this work. Kim and Hong (2023) curated a benchmark of OSS Fuzz reported bugs for evaluating regression fuzzers. However, they omit bugs that are found in less than three minutes. This choice is appropriate for determining which fuzzers are best at finding challenging regression bugs, but diminishes the ability of fuzzers of finding regression bugs quickly in general. Additionally, Kim and Hong use the seed corpus present in the version control history for the bug-inducing commit of the given program. We use this setup as our baseline *static* corpus, but also compare results to a *saturated* corpus up to the bug-inducing commit. Indeed, our results indicate that the setup used by Kim et al (2024) is *not optimal* for change-based fuzzing. Klooster et al (2023) also conduct an evaluation of fuzzers in the CI/CD context, similarly finding that many regression bugs can be found relatively quickly by existing fuzzers (< 15 minutes). However, unlike our follow-up study, their research focuses only on the capabilities of general-purpose grey-box fuzzers, omitting regression and directed fuzzers. Instead of using a saturated corpus on a single commit, they simulate a sequence of commits starting from the corpus of the first bug-inducing commit and augmenting it cumulatively. As neither approach is an exact replication of long-term continuous fuzzing, we believe that results from both works can be informative and complementary. Kim et al (2024) assessed the state of directed fuzzer *evaluations*, excluding general purpose and regression fuzzers, and do so in a more conventional context with 24 hour timeouts rather than CI/CD or local development workflows.

8 Perspectives

Fuzz testing has conventionally been used for finding security vulnerabilities in existing software systems. As such, it has been primarily employed on mature software

[13] https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing
[14] https://www.code-intelligence.com/cli-tool

systems or even vendor-provided code upon acquisition. The role of fuzz testing in software development has been relatively less examined. This is the outlook we examine in this paper. Our follow up study shows that, to some extent, developer expectations from our survey can already be met by existing technology; if left-shifted into development workflows, fuzzers *can* effectively find many regression errors *within* tolerable time limits. Beyond change-based fuzzing, the user survey we present in this paper is of general relevance. Our participants' answers provide many insights on the usage of fuzzing, whether conducted as part of the software development process or for hardening a mature or acquired software system.

The findings from our work indicate the need to connect fuzzers with new development tools. For the software engineering community, our work promotes the need to integrate fuzzing into future development environments and build workflows, with the goal of writing a functionally correct program. This would truly amount to a shift-left of fuzzing in software engineering workflows.

Declarations

Funding and/or Conflicts of interests/Competing interests. The authors declare that they have no conflict of interest. This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

Compliance with Ethical Standards. We obtained approval from the Institutional Review Board (IRB) of the National University of Singapore before executing our developer survey.

Data Availability. Our research artifact includes all data, statistics, and codes for our user study, as well as the data for our fuzzing experiments:

https://doi.org/10.6084/m9.figshare.24769719.v1.

References

Babić D, Bucur S, Chen Y, et al (2019) Fudge: Fuzz driver generation at scale. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 975–985, https://doi.org/10.1145/3338906.3340456

Beck (2002) Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., USA

Böhme M, Pham VT, Nguyen MD, et al (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications

Security. Association for Computing Machinery, New York, NY, USA, CCS '17, p 2329–2344, https://doi.org/10.1145/3133956.3134020

Böhme M, Cadar C, Roychoudhury A (2020) Fuzzing: Challenges and reflections. IEEE Software 38(3):79–86. https://doi.org/10.1109/MS.2020.3016773

Braz L, Fregnan E, Arora V, et al (2022) An exploratory study on regression vulnerabilities. In: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. Association for Computing Machinery, New York, NY, USA, ESEM '22, p 12–22, https://doi.org/10.1145/3544902.3546250

Calcagno C, Distefano D (2011) Infer: An automatic program verifier for memory safety of c programs. In: Bobaru M, Havelund K, Holzmann GJ, et al (eds) NASA Formal Methods. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 459–465, https://doi.org/10.1007/978-3-642-20398-5_33

Canakci S, Matyunin N, Graffi K, et al (2022) Targetfuzz: Using darts to guide directed greybox fuzzers. In: Proceedings of the 2022 ACM on Asia conference on computer and communications security, pp 561–573, https://doi.org/10.1145/3488932.3501276

Churchill D (2018) Keynotes. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pp 23–25, https://doi.org/10.1109/ICST.2018.00010

Daka E, Fraser G (2014) A survey on unit testing practices and problems. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, IEEE, pp 201–211, https://doi.org/https://doi.org/10.1109/ISSRE.2014.11

Distefano D, Fähndrich M, Logozzo F, et al (2019) Scaling static analyses at facebook. Commun ACM 62(8):62–70. https://doi.org/10.1145/3338112

Dodge Y (2008) Binomial Test, Springer New York, New York, NY, pp 47–49. https://doi.org/10.1007/978-0-387-32833-1_36

Fioraldi A, Maier D, Eißfeldt H, et al (2020) Afl++: Combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. USENIX Association, USA, WOOT'20, URL https://www.usenix.org/conference/woot20/presentation/fioraldi

Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp 416–419, https://doi.org/10.1145/2025113.2025179

Fraser G, Staats M, McMinn P, et al (2015) Does automated unit test generation really help software testers? a controlled empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(4):1–49. https://doi.org/https://doi.org/10.1145/2699688

Godefroid P (2020) Fuzzing: hack, art, and science. Commun ACM 63(2):70–76. https://doi.org/10.1145/3363824, URL https://doi.org/10.1145/3363824

Goodin D (2023) "Cisco buried the lede." <10,000 network devices backdoored through unpatched 0-day. https://arstechnica.com/security/2023/10/actively-exploited-cisco-0-day-with-maximum-10-severity-gives-full-network-control/, [Accessed 27-10-2023]

Gorski PL, Iacono LL, Wermke D, et al (2018) Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). USENIX Association, Baltimore, MD, pp 265–281, URL https://www.usenix.org/conference/soups2018/presentation/gorski

Herrera A, Gunadi H, Magrath S, et al (2021) Seed selection for successful fuzzing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, New York, NY, USA, ISSTA 2021, p 230–243, https://doi.org/10.1145/3460319.3464795

Hilton M, Nelson N, Tunnell T, et al (2017) Trade-offs in continuous integration: assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017, p 197–207, https://doi.org/10.1145/3106237.3106270

Javed Y, Sethi S, Jadoun A (2019) Alexa's voice recording behavior: A survey of user understanding and awareness. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. Association for Computing Machinery, New York, NY, USA, ARES '19, https://doi.org/10.1145/3339252.3340330

Jin M, Shahriar S, Tufano M, et al (2023) Inferfix: End-to-end program repair with llms. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2023, p 1646–1656, https://doi.org/10.1145/3611643.3613892

Just R, Jalali D, Inozemtseva L, et al (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, FSE 2014, p 654–665, https://doi.org/10.1145/2635868.2635929

Kalbfleisch JD, Prentice RL (2002) The statistical analysis of failure time data. John Wiley & Sons

Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. Journal of the American statistical association 53(282):457–481

Kelly M, Treude C, Murray A (2019) A case study on automated fuzz target generation for large codebases. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–6, https://doi.org/10.1109/ESEM.2019.8870150

Kim J, Hong S (2023) Poster: Bugoss: A regression bug benchmark for empirical study of regression fuzzing techniques. In: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp 470–473, https://doi.org/10.1109/ICST57152.2023.00053

Kim TE, Choi J, Im S, et al (2024) Evaluating directed fuzzers: Are we heading in the right direction? Proc ACM Softw Eng 1(FSE). https://doi.org/10.1145/3643741

Kitchenham BA, Pfleeger SL (2008) Personal Opinion Surveys, Springer London, London, pp 63–92. https://doi.org/10.1007/978-1-84800-044-5_3

Klees G, Ruef A, Cooper B, et al (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, CCS '18, p 2123–2138, https://doi.org/10.1145/3243734.3243804

Klooster T, Turkmen F, Broenink G, et al (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), pp 25–32, https://doi.org/10.1109/SBFT59156.2023.00015

Lane B (2020) Equifax expects to pay out another 100 million for data breach. URL https://www.housingwire.com/articles/equifax-expects-to-pay-out-another-100-million-for-data-breach/

Liang J, Wang M, Chen Y, et al (2018) Fuzz testing in practice: Obstacles and solutions. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 562–566, https://doi.org/10.1109/SANER.2018.8330260

van der Linden D, Anthonysamy P, Nuseibeh B, et al (2020) Schrödinger's security: opening the box on app developers' security rationale. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, ICSE '20, p 149–160, https://doi.org/10.1145/3377811.3380394, URL https://doi.org/10.1145/3377811.3380394

Luo C, Meng W, Li P (2023) Selectfuzz: Efficient directed fuzzing with selective path exploration. In: 2023 IEEE Symposium on Security and Privacy (SP), pp 2693–2707, https://doi.org/10.1109/SP46215.2023.10179296

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics 18(1):50–60. URL http://www.jstor.org/stable/2236101

Mansur MN, Christakis M, Wüstholz V, et al (2020) Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, p 701–712, https://doi.org/10.1145/3368089.3409763

Marinescu PD, Cadar C (2013) Katch: High-coverage testing of software patches. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, p 235–245, https://doi.org/10.1145/2491411.2491438

Memon A, Gao Z, Nguyen B, et al (2017) Taming google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp 233–242, https://doi.org/10.1109/ICSE-SEIP.2017.16

Menendez HD, Clark D (2022) Hashing fuzzing: Introducing input diversity to improve crash detection. IEEE Transactions on Software Engineering 48(9):3540–3553. https://doi.org/10.1109/TSE.2021.3100858

Meng R, Dong Z, Li J, et al (2022) Linear-time temporal logic guided greybox fuzzing. In: Proceedings of the 44th International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, ICSE '22, p 1343–1355, https://doi.org/10.1145/3510003.3510082

Metzman J, Szekeres L, Simon L, et al (2021) Fuzzbench: An open fuzzer benchmarking platform and service. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2021, p 1393–1403, https://doi.org/10.1145/3468264.3473932

Micco J (2018) Advances in continuous integration testing at google. URL https://research.google/pubs/pub46593

Miller BP, Fredriksen L, So B (1990) An empirical study of the reliability of unix utilities. Commun ACM 33(12):32–44. https://doi.org/10.1145/96267.96279

Newman LH (2021) 'the internet is on fire'. URL https://www.wired.com/story/log4j-flaw-hacking-internet/

Noller Y, Pasareanu C, Böhme M, et al (2020) Hydiff: Hybrid differential software analysis. In: Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering, ICSE 2020, pp 1273–1285, https://doi.org/10.1145/3377811.3380363

Noller Y, Shariffdeen R, Gao X, et al (2022) Trust enhancement issues in program repair. In: Proceedings of the 44th International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, ICSE '22, p 2228–2240, https://doi.org/10.1145/3510003.3510040

Nourry O, Kashiwa Y, Lin B, et al (2023) The human side of fuzzing: Challenges faced by developers during fuzzing activities. ACM Trans Softw Eng Methodol 33(1). https://doi.org/10.1145/3611668

Pearson K (1900) X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 50(302):157–175. https://doi.org/10.1080/14786440009463897

Pham VT, Böhme M, Roychoudhury A (2020) Aflnet: A greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp 460–465, https://doi.org/10.1109/ICST46399.2020.00062

Phan QS, Nguyen KH, Nguyen T (2023) The challenges of shift left static analysis. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp 340–342, https://doi.org/10.1109/ICSE-SEIP58684.2023.00036

Plöger S, Meier M, Smith M (2021) A qualitative usability evaluation of the clang static analyzer and libFuzzer with CS students and CTF players. In: Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021). USENIX Association, pp 553–572, URL https://www.usenix.org/conference/soups2021/presentation/ploger

Plöger S, Meier M, Smith M (2023) A usability evaluation of afl and libfuzzer with cs students. In: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery, New York, NY, USA, CHI '23, https://doi.org/10.1145/3544548.3581178

Royston P, Parmar MK (2013) Restricted mean survival time: an alternative to the hazard ratio for the design and analysis of randomized trials with a time-to-event outcome. BMC medical research methodology 13:1–15. https://doi.org/https://doi.org/10.1186/1471-2288-13-152

Runeson P (2006) A survey of unit testing practices. IEEE Software 23(4):22–29. https://doi.org/10.1109/MS.2006.91

Schreier M (2012) Qualitative content analysis in practice. Sage publications

Serebryany K (2017) OSS-Fuzz - google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC, URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

Shi H, Wang R, Fu Y, et al (2019) Industry practice of coverage-guided enterprise linux kernel fuzzing. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 986–995, https://doi.org/10.1145/3338906.3340460

Smith L (2001) Shift-Left Testing — drdobbs.com. https://www.drdobbs.com/shift-left-testing/184404768, [Accessed 25-10-2023]

Votipka D, Fulton KR, Parker J, et al (2020) Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, pp 109–126, URL https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

Wald A (1943) Tests of statistical hypotheses concerning several parameters when the number of observations is large. Transactions of the American Mathematical society 54(3):426–482

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics Bulletin 1(6):80–83

Winters T, Manshreck T, Wright H (2020) Software engineering at google: Lessons learned from programming over time. O'Reilly Media

Yan Q, Cao H, Lu S, et al (2023) Infuzz: An interactive tool for enhancing efficiency in fuzzing through visual bottleneck analysis (registered report). In: Proceedings of the 2nd International Fuzzing Workshop. Association for Computing Machinery, New York, NY, USA, FUZZING 2023, p 56–61, https://doi.org/10.1145/3605157.3605847, URL https://doi.org/10.1145/3605157.3605847

Zhang J, Cui Z, Chen X, et al (2023) Cidfuzz: Fuzz testing for continuous integration. IET Software 17(3):301–315. https://doi.org/10.1049/sfw2.12125

Zhu X, Böhme M (2021) Regression greybox fuzzing. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, CCS '21, p 2169–2182, https://doi.org/10.1145/3460120.3484596, URL https://doi.org/10.1145/3460120.3484596