The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities

OLIVIER NOURRY, Kyushu University, Japan
YUTARO KASHIWA, Nara Institute of Science and Technology, Japan
BIN LIN, Radboud University, The Netherlands
GABRIELE BAVOTA, Università della Svizzera italiana, Switzerland
MICHELE LANZA, Università della Svizzera italiana, Switzerland
YASUTAKA KAMEI, Kyushu University, Japan

Fuzz testing, also known as fuzzing, is a software testing technique aimed at identifying software vulnerabilities. In recent decades, fuzzing has gained increasing popularity in the research community. However, existing studies led by fuzzing experts mainly focus on improving the coverage and performance of fuzzing techniques. That is, there is still a gap in empirical knowledge regarding fuzzing, especially about the challenges developers face when they adopt fuzzing. Understanding these challenges can provide valuable insights to both practitioners and researchers on how to further improve fuzzing processes and techniques.

We conducted a study to understand the challenges encountered by developers during fuzzing. More specifically, we first manually analyzed 829 randomly sampled fuzzing-related GitHub issues and constructed a taxonomy consisting of 39 types of challenges (22 related to the fuzzing process itself, 17 related to using external fuzzing providers). We then surveyed 106 fuzzing practitioners to verify the validity of our taxonomy and collected feedback on how the fuzzing process can be improved. Our taxonomy, accompanied with representative examples and highlighted implications, can serve as a reference point on how to better adopt fuzzing techniques for practitioners, and indicates potential directions researchers can work on toward better fuzzing approaches and practices.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **General and reference** → *Empirical studies*; Surveys and overviews.

Additional Key Words and Phrases: fuzzing, software testing, empirical software engineering

ACM Reference Format:
Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities. 1, 1 (July 2023), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given the complexity of modern software systems, a tremendous number of open-source software (OSS) libraries are adopted by developers to ease development tasks and enhance productivity.

Authors' addresses: Olivier Nourry, oliviern@posl.ait.kyushu-u.ac.jp, Kyushu University, Japan; Yutaro Kashiwa, Nara Institute of Science and Technology, Japan, yutaro.kashiwa@is.naist.jp; Bin Lin, Radboud University, The Netherlands, bin.lin@ru.nl; Gabriele Bavota, Università della Svizzera italiana, Switzerland, gabriele.bavota@usi.ch; Michele Lanza, Università della Svizzera italiana, Switzerland, michele.lanza@usi.ch; Yasutaka Kamei, kamei@ait.kyushu-u.ac.jp, Kyushu University, Japan.

XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, recently disclosed issues such as the heartbleed vulnerability [12] and the log4j vulnerability [2] harshly reminded the open source community how even a single bug can propagate through the internet and impact millions. A breach of security of this magnitude can easily compromise the trust from the enterprise sector in OSS library adoption as indicated by the 2022 Anaconda State of Data Science report [1]: 51% of the respondents stated that the fear of vulnerabilities is the main cause preventing their organization from leveraging open source, while 25% admit to scaling back their use of OSS as a direct result of the log4j vulnerability. Therefore, it is critical to ensure that OSS libraries are secure and fault-proof.

In this context, developers around the world have now incorporated extensive testing and bug detection activities into their software development cycle. However, testing activities can be complex and time-consuming in nature, resulting in industry practices shifting toward test automation and more sophisticated testing infrastructure [57]. In particular, one type of automated testing techniques called fuzz testing (or fuzzing) [63] has been gaining popularity. Its goal is to reveal software defects and vulnerabilities by injecting invalid, unexpected, or random inputs to software.

While a lot of research has been done to improve fuzzing techniques [10, 13, 71, 72, 83], there is currently a lack of knowledge about fuzzing from the perspective of developers. More specifically, developers can face many complex challenges trying to conduct their fuzzing activities depending on the scope and the nature of the software. For example, developers may not know what type of fuzzer best suits their codebase, and fuzzers can also require specific configurations based on what kind of software is being fuzzed. Fuzzers are also notorious for requiring substantial processing power and memory, which makes fuzzing on a personal device difficult; especially if the fuzz targets are very large. To alleviate these performance issues, developers often turn to external sources to fuzz their software. The OSS-Fuzz initiative by Google [27], for example, continuously fuzzes hundreds of projects considered key/important for the OSS ecosystem with the goal of finding critical vulnerabilities before the ecosystem is affected.

Our goal is to shed light on what challenges developers face when fuzzing their software. We focus on developers who adopt fuzzing in their testing practices instead of those who create or maintain fuzzing techniques. Having such knowledge can not only help us to identify existing issues of using fuzzers, but also highlight the potential directions researchers and practitioners can investigate for better fuzzing tools.

To the best of our knowledge, no comprehensive empirical study has been conducted in this manner. The most relevant work is the one by Boehme *et al.* [7], which lists general challenges regarding fuzzing techniques after seminar discussions. However, these results were produced by brainstorming instead of being extracted from evidence, leading to coarse-grained challenges, which did not put fuzzing practitioners in the center of attention.

We build our theories from two sources: 1) issues discussed on GitHub and 2) developers' opinions collected through a survey. More specifically, we first retrieved 3,721 GitHub issues from fuzzing communities. We sampled 829 issues and manually analyzed developers' discussions to identify the specific challenges faced by developers. Then, a card-sorting approach [76] was followed to define a taxonomy consisting of challenge categories. In total, we obtained 22 categories of challenges regarding fuzzing tools and practices themselves, and 17 categories of challenges regarding fuzzing service providers. After that, we used an online survey to collect developers' feedback regarding the challenges we identified, and check whether any challenges were missed. We distributed the survey to developers who are active in OSS projects which adopted fuzzing, and collected a total of 106 valid responses. These responses were used to examine our taxonomy, which also brought some new insights regarding fuzzing tools and practices.

The main contributions of our paper are as follows.

- We are the first to establish a comprehensive taxonomy of challenges developers encounter during fuzzing activities and verify it with developers who have relevant expertise.
- Through both qualitative and quantitative approaches, we analyze the prevelance of the challenges presented in the taxonomy.
- We provide potential solutions to the challenges presented, accompanied with developers' insightful feedback, which shed light on future research directions.

2 STUDY DESIGN

The *goal* of our study is to investigate the challenges developers encountered during fuzzing. The *context* consists of (i) 829 manually analyzed GitHub issues from 129 fuzzing-related repositories and (ii) 106 responses collected from an online survey. The study aims to answer the following research question:

What are the challenges practitioners face when fuzzing their software?

We focus on developers and researchers who are involved in fuzzing activities, and aim to collect concrete challenges they face in practice. To answer our research question, we follow two steps. First, we conducted a manual analysis on 829 GitHub issues and formed a taxonomy of challenges developers face during fuzzing. We then distributed a survey to fuzzing practitioners to get their feedback on our taxonomy. We also asked them directly what challenges they have faced while fuzzing, and to tell us what areas of fuzzing they think should be improved.

2.1 Manual Analysis on GitHub Issues

To understand what challenges developers face during fuzzing, we investigated discussions in GitHub issues and extracted relevant information. The GitHub issues were extracted through two sources:

Issues from the OSS-Fuzz repository. OSS-Fuzz[1] is a service by Google that provides continuous fuzzing for OSS projects which either have a significant user base or are critical to the global IT infrastructure. It supports several popular fuzzing engines (*e.g.,* `libFuzzer`[2], `AFL++`[3]) and multiple programming languages (*e.g.,* C/C++, Rust, Go). We collected all the GitHub issues on the OSS-Fuzz repository through the GitHub API, and obtained 1,766 issues in total.

Issues from projects participating in Google's OSS-Fuzz initiative. By the time we conducted this study, over 500[4] projects had been fuzzed by OSS-Fuzz according to the official website.[5] At the time of data collection, 361 projects were actively being fuzzed (the projects were in the "projects" directory of OSS-Fuzz GitHub repository). Since these projects themselves are not focusing on fuzzing techniques, the discussions in their issue trackers might contain several discussion threads outside of our interest. To isolate fuzzing-related issues from other issues within each GitHub repository, we only collected issues whose title contains "afl" or "fuzz". These two keywords were chosen because "fuzz" allows us to catch any variation of the word such as "fuzzing" and "fuzzers", while "afl" is one of the most widely used fuzzers[6] [80]. This process led to 2,562 issues.

A manual inspection of dozens of issues revealed two mains problems within our dataset: First, many issues were merely reporting bugs found by fuzzers; second, the issues in the repositories where fuzzing tools are developed are mostly related to development challenges rather than fuzzing challenges while also containing the keywords we were looking for. Since our goal is to identify

[1]https://github.com/google/oss-fuzz
[2]https://llvm.org/docs/LibFuzzer.html
[3]https://aflplus.plus/
[4]The number is dynamic and keeps increasing.
[5]https://google.github.io/oss-fuzz/
[6]While the fullname of "afl" is "American Fuzzy Lop", in practice, developers tend to use its abbreviated version.

the challenges of using fuzzers, we performed extra filtering by removing the GitHub issues whose titles indicated they were automatically generated (*e.g.,* OSS-Fuzz issue 41533 [7]).

We then manually removed a few repositories which we found to be fuzzing tool development repositories, to reduce the number of false positives we would encounter during our manual labeling process (GitHub issues related to fuzzing tool development instead of a fuzzing challenge).

At the end of the process, we ended up with 3,721 unique GitHub issues from these two sources. While we acknowledge that fuzzing-related discussions also widely exist in other projects not sampled in this study, we believe the large number of included projects is representative, and our project selection approach largely reduced the cost of searching due to the fact that fuzzing is a very specific task and only counts for a small fraction of all the issues.

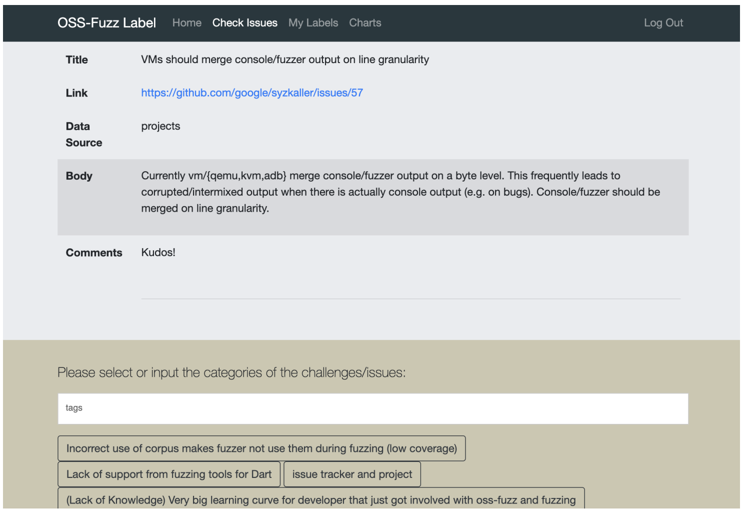
2.1.1 Manual Identification of Fuzzing Challenges. The 3,721 GitHub issues collected were used for our qualitative analysis: our goal was to extract fuzzing challenges from issue content and developer discussions. Not all the issues contain discussions related to fuzzing challenges, and some are not even related to fuzzing at all. However, these issues would be filtered out during our manual inspection.

Given the large number of issues and limited human resources, it was not realistic to inspect all the issues. Therefore, we extracted a randomly-stratified sample of 829 issues from our two data sources. The strata represented the data source, *i.e.,* issues were sampled proportionally to the number of collected issues from the OSS-Fuzz repository and the number of collected issues from projects participating in Google's OSS-Fuzz initiative. More specifically, we applied a sample size formula for an unknown population [17] which guarantees a confidence level of 95% with a significance interval (margin of error) of ±3%. This resulted in 829 issues in total, including 390 from the official OSS-Fuzz GitHub repository and 439 extracted from our list of projects participating in Google's OSS-Fuzz initiative.

We followed the card sorting procedure [76] to create a taxonomy of challenges developers face during fuzzing. In our case, multiple people formed the theory collectively without any predefined categories. Each issue was inspected independently by two authors and subsequently annotated. As the authors are located in different locations, to keep the process organized and manageable, we created a web app to assist with the annotation process (Figure 1).

The web app shows the title, link, data source, body and comments of the issue. An annotator then assigns challenge labels to the issue. One issue can be assigned multiple labels if different challenges are found. In case no challenge is presented, the annotator labels it as a "false positive" and continues with the next issue. During the annotation process, to keep a consistent taxonomy, annotators are allowed to reuse labels created before. However, we do ask annotators to clearly state the essense of the challenge. For example, when the fuzzer has compatibility issues, the label needs to include the information about which environment is incompatible with the fuzzer. When two issues encounter the exactly same challenge, annotators are encouraged to use the existing label over creating a new label. Due to the level of details added to the labels, however, it is likely that the authors may not find an exact match in the pool of existing labels. If none of the current labels perfectly fits, annotators can create a new label, and the created label is automatically added to the shared set and become visible for others to use from the time being. The web app also keeps track of which issues have already been labeled by each of the annotators, and allows annotators to modify their own labels. The assigned labels of an issue from one annotator is hidden from other annotators throughout the whole process so that annotators will not be biased when deciding the labels.

[7]https://github.com/google/gopacket/issues/983



Title: VMs should merge console/fuzzer output on line granularity

Link: https://github.com/google/syzkaller/issues/57

Data Source: projects

Body: Currently vm/{qemu,kvm,adb} merge console/fuzzer output on a byte level. This frequently leads to corrupted/intermixed output when there is actually console output (e.g. on bugs). Console/fuzzer should be merged on line granularity.

Comments: Kudos!

Please select or input the categories of the challenges/issues:

tags

Incorrect use of corpus makes fuzzer not use them during fuzzing (low coverage)

Lack of support from fuzzing tools for Dart

issue tracker and project

(Lack of Knowledge) Very big learning curve for developer that just got involved with oss-fuzz and fuzzing

Fig. 1. Web App used for inspecting issues

Three authors independently annotated the selected issues. The web app automatically assigned two annotators to each issue. To ensure that assigned labels are consistent, we used 80 issues as our trial batch. After annotating them, the three authors looked into the labels together and discussed how to assign better labels. The three authors then labeled the remaining 749 issues. To lower fatigue, conflicts were resolved after every 100 annotated issues. After each iteration, all three authors held an online meeting and went through the labels together. Any conflict was discussed until an agreement was reached. If an issue was ambiguous (not clear if the developers involved were actually facing challenges, or simply reporting an update or creating an issue for logging purposes) the issue was discarded.

At the end of the process, we could filter out 563 false positives. For the remaining 266 issues, we initially obtained 521 labels, among which 448 were unique. While each issue was annotated by two authors, some issues are annotated as false positives by one author, therefore they only received one valid label. During the conflict resolution, three authors agreed that these issues should be relevant and re-assigned labels to them after discussion. After resolving all the conflicts and merging the two labels for each issue, we obtained 262 unique labels (170 fuzzing process + 92 fuzzing provider).

2.1.2 Creating the First Taxonomy. The three authors who participated in the annotating process worked together to conduct the card sorting [76] on the 262 labels of challenges. The goal was to merge labels with similar meanings and categorize these labels. The general categories were based on the ISO/IEC 25,010 product quality model [58]. Extra categories were added when necessary. The card sorting process was done in iterations where the three authors inspected the labels sequentially. Any disagreement about the categorization was discussed after the inspection until consensus was reached. All annotators were also involved with refining the categories of fuzzing challenges. In the end, 22 challenges related to fuzzing processes and 17 challenges related to using a fuzzing service provider were obtained.

2.2 Survey with Developers

To verify the validity of our taxonomy and get feedback from developers involved in fuzzing activities, we conducted a survey which served three purposes: (1) to understand how often developers encounter the fuzzing challenges we observed during our investigation; (2) to know from the developers whether we missed any challenges during our manual investigation; (3) to ask the developers directly which areas of fuzzing need improvement and how the software engineering community could potentially help address these issues. Since our focus is to study challenges directly related to fuzzing, we did not include challenges related to using fuzzing service providers in our survey.

2.2.1 Survey Design. The first part of the survey consisted of generic questions about the background and experience of the respondents. In the second part of the survey we asked developers how often they encounter each type of challenge defined in our taxonomy. The survey provides multiple choice answers for each of these challenges previously identified following a 5-level Likert scale answer (*i.e.,* "Always", "Often", "Sometimes", "Rarely" and "Never") [14]. An "I don't know" option was also provided. The final part of the survey was made up of free answer questions to get feedback from developers directly. To verify the integrity of our taxonomy, we asked the survey respondents to write down if they have encountered any other challenges not mentioned in the survey. Finally, we asked developers to comment on how the fuzzing process could be improved.

2.2.2 Survey Participants. Given the very specific topic investigated in the paper, it was important to reach the right survey respondents. We identified our targets through two sources: (1) We selected developers who participated in issue discussions of the official OSS-Fuzz GitHub repository. Since the whole repository is about fuzzing, we knew that the participants to this repository were either directly or indirectly conducting fuzzing activities; (2) We collected developers who are involved in the GitHub projects whose topic labels include "fuzzing" via the GitHub search feature. In this way we could reach a wider variety of developers who might have different fuzzing experience.

We sent the survey to 1,545 fuzzing practitioners and made it available for 25 days. Out of 1,545 potential respondents, we got 106 valid (complete) responses. This gives us a 6.86% answer rate, on par with most software engineering surveys which typically get a response rate of around 5% [61]. Most of our respondents identified as developers (42%), followed by technical leads (25%) and experts working in the security field (14%). The remaining respondents identified as either researchers, bug hunters, engineering managers or did not share their profession.

Table 1. Years of development and fuzzing experience of the survey respondents.

	<3 years	3-5 years	6-10 years	>10 years
Professional software development experience	13	15	27	51
Fuzzing experience	47	40	13	6

Table 1 shows the experience of survey respondents in software development and fuzzing. The whole experience spectrum is covered, however, the experience distributions demonstrate a totally different trend: 74% of respondents have over 5 years of experience in software development, while the number drops to 18% for fuzzing experience. The reason might be that the popularity of fuzzing has greatly increased in OSS communities only within recent years and experienced developers are just starting to get involved with fuzzing.

3 RESULTS

During our manual analysis, two types of challenges emerged: one is related to fuzzing itself, the other boils down to the issues brought by using third-party fuzzing service providers (in our case, OSS-Fuzz). We discuss these two types of challenges in Section 3.1 and Section 3.2, respectively. To clearly present our results and highlight the insights, we use the following notations:

- ℹ: provides an example of GitHub issues which fit into the current category of fuzzing challenges.
- 🧍: illustrates respondents' experience or opinions related to the challenge. Each respondent is represented as R#.
- 💡: indicates potential solutions or directions to investigate for addressing some of the challenges.

3.1 Challenges from Fuzzing Itself

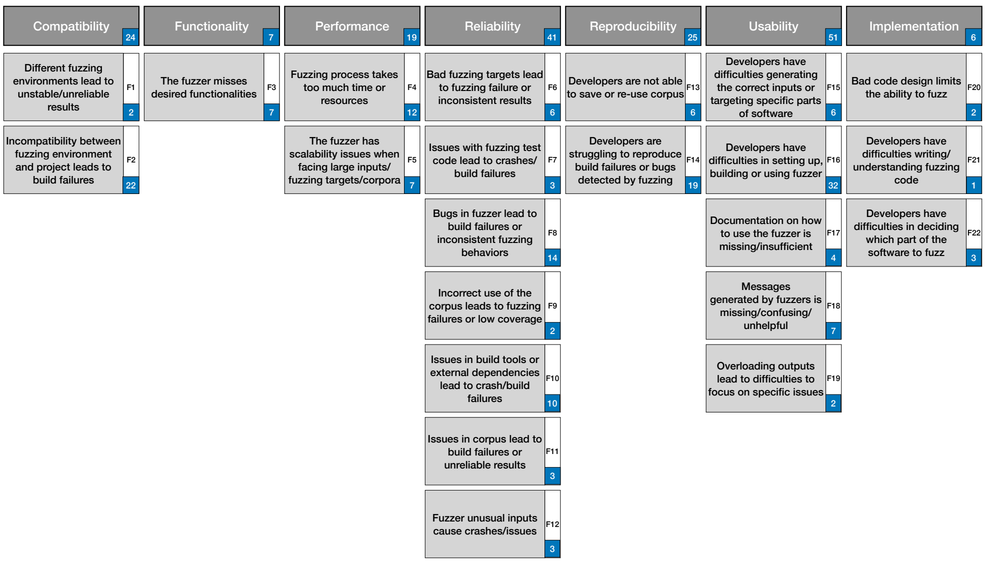

Compatibility 24
Different fuzzing environments lead to unstable/unreliable results F1 2
Incompatibility between fuzzing environment and project leads to build failures F2 22
Functionality 7
The fuzzer misses desired functionalities F3 7
Performance 19
Fuzzing process takes too much time or resources F4 12
The fuzzer has scalability issues when facing large inputs/ fuzzing targets/corpora F5 7
Reliability 41
Bad fuzzing targets lead to fuzzing failure or inconsistent results F6 6
Issues with fuzzing test code lead to crashes/ build failures F7 3
Bugs in fuzzer lead to build failures or inconsistent fuzzing behaviors F8 14
Incorrect use of the corpus leads to fuzzing failures or low coverage F9 2
Issues in build tools or external dependencies lead to crash/build failures F10 10
Issues in corpus lead to build failures or unreliable results F11 3
Fuzzer unusual inputs cause crashes/issues F12 3
Reproducibility 25
Developers are not able to save or re-use corpus F13 6
Developers are struggling to reproduce build failures or bugs detected by fuzzing F14 19
Usability 51
Developers have difficulties generating the correct inputs or targeting specific parts of software F15 6
Developers have difficulties in setting up, building or using fuzzer F16 32
Documentation on how to use the fuzzer is missing/insufficient F17 4
Messages generated by fuzzers is missing/confusing/ unhelpful F18 7
Overloading outputs lead to difficulties to focus on specific issues F19 2
Implementation 6
Bad code design limits the ability to fuzz F20 2
Developers have difficulties writing/ understanding fuzzing code F21 1
Developers have difficulties in deciding which part of the software to fuzz F22 3

Fig. 2. Taxonomy of challenges from the fuzzing process itself.

At the end of the annotating process, we identified 171 GitHub issues containing challenges related to fuzzing itself, out of the 829 initially sampled issues. These 171 issues span over 22 distinct sub-categories defined during the label refining process and constitute our taxonomy of fuzzing challenges. Figure 2 shows the complete taxonomy of these 22 types of challenges, which belong to seven large categories including compatibility, functionality, performance, reliability, reproducibility, usability, and implementation. The numbers in blue squares attached to categories indicate how many times the corresponding types of challenges occur during our manual analysis. As mentioned in Section 2, the categories are derived from the ISO/IEC 25,010 product quality model [58]. We added the one extra category of "implementation" deemed necessary and not fitting within one of the existing categories. Below we describe these categories in detail and provide both real-world examples and insights gained from the survey done with developers conducting fuzzing activities.

3.1.1 Compatibility Challenges. These arise when fuzzers are not compatible with certain elements of the environment they are being executed in.

Different fuzzing environments lead to unstable/unreliable results (F1). Developers often run their fuzzing tests on different machines to save resources. When these machines have different environments, the consistency of the fuzzing output can be affected or disrupted.

> ℹ For example, in issue #3347 of the scapy project [74], a fuzzing developer finds out that the outcome of fuzzing the software is not consistent across runs due to different python versions being used.

Incompatibility between fuzzing environment and project leads to build failures (F2). In this category, we find environment challenges which completely prevent developers from fuzzing unlike in F1 where developers can fuzz but the results are not reliable. Versioning issues, for instance, were one of the most common recurring themes while conducting the manual investigation. Compiler updates and versioning, specifically, can break fuzzing builds and require complex manual intervention from the developers to fix the problem. Several fuzzers only working on specific compilers and compiler versions also led to multiple fuzzing build failures. R16 and R18 of our survey both indicated having issues regarding fuzzing on multiple platforms. This type of issue seems to be frequently encountered by developers and was present in nearly 13% of all issues related to fuzzing challenges.

> ℹ As shown in issue #19557 from the bitcoin GitHub repository [4], the operating system running on the machine can be a limiting factor as many fuzzers are either platform dependent or must be compiled differently on different operating systems.
> R16: "*[One challenge is the] cross-platform use of fuzzers, e.g., if a development team primarily targets windows, the available fuzzing tools for windows are very lacking and hard to use.*"

Compatibility suggestion(s). *Integrating fuzzing with the environment like the GO language has done (as suggested by R31) or integrating fuzzing with testing frameworks (as suggested by R15) could be possible approaches to address environment issues and platform dependence.*

3.1.2 Functionality Challenges. These are encountered by developers due to the limitations of fuzzers and their features.

The fuzzer misses desired functionalities/features (F3). Some fuzzers lack features that prevent developers from fuzzing their system properly. Since systems vary, specialized fuzzing tools can be required for different types of software. Several respondents (R18, R27, R28, R34) mentioned that they had a need for fuzzers that support the latest tools and techniques. Another developer (R13) also mentioned that they would like it if fuzzers could provide fuzzing suggestions such as recommending areas of the source code which have not been covered by existing fuzzing code.

> ℹ In issue #1675 of the official OSS-Fuzz GitHub repository [38], a developer wishes to fuzz memory allocations but only finds out that this functionality was not yet supported by the fuzzers at the time.
> R12: "*With afl, I end up having to start from scratch every cycle. It would be nice if it were easier to minimize while I'm fixing issues and then have a fast start based on the previous cycle. Plus I sometimes have to recompile to work with other tools, like valgrind or gdb.*"
> R44: "*Runtime target selection for coverage guided tools would be a great improvement.*"

Functionality suggestion(s). *As developers often need to implement some needed functions for fuzzing themselves, it would be helpful to create an index to facilitate the reuse of the implemented functionalities.*

3.1.3 Performance Challenges. This category of fuzzing challenges includes all obstacles related to poor performance of fuzzers or fuzzing practices.

Fuzzing process takes too much time or resources (F4). Since fuzzers require significant memory and CPU consumption, it is not uncommon that developers mention that their fuzzer is taking all of their local machine's memory and prevents them from doing other work.

> ℹ Developers of the "wasmtime" project discussed the problem in a GitHub issue [11] that their fuzzers are taking too much time to compile and run, making it not worth the value gained by fuzzing.
> R13: "*Allocating time and resources for continued fuzzing is difficult; ideally tooling would make it easy for e.g., all developers on a project to contribute to fuzzing at a low-level, rather than occupying dedicated machines for brief but intense periods of fuzzing [...].*"

The fuzzer has scalability issues when facing large inputs/fuzzing targets/corpora (F5). Scalability concerns have been another recurring theme while investigating GitHub issues. These issues can arise from multiple different sources including the target being fuzzed, the corpora of old crashes, or the list of valid inputs used for future mutations.

> ℹ In issue #20088 of the bitcoin project [5], developers wonder how to scale their fuzzing as the number of fuzz targets, CPU time and disk space all grow over time. As the developer also mentions, keeping good coverage is another challenge to consider when scaling up.

Performance suggestion(s). Using an external agent to fuzz your software such as OSS-Fuzz or integrating the fuzzer(s) to a CI tool can alleviate some of the issues related to fuzzers using too much memory.

3.1.4 Reliability Challenges. They affect the reliability of the fuzzing process and its results. Obstacles causing abnormal or unexpected behaviors during the fuzzing process are categorized in one of the challenges included in this category.

Bad fuzzing targets lead to fuzzing failure or inconsistent results (F6) and Issues with fuzzing test code lead to crashes/build failures (F7). In these two categories, we find a wide range of cases where the fuzzing target itself or the fuzzing code caused problems for developers trying to fuzz. Multiple respondents (R4, R25, R43, R44) mentioned the difficulties of writing good fuzz targets.

> ℹ The fuzz target being too small can cause problems for the fuzzer(s) and result in breaking the fuzzing build, as shown in issue #815 of the OSS-Fuzz repository [34].
> R4: "*Writing a good fuzz target is hard, because it requires domain knowledge.*"
> R25: "*Writing a harness for a closed-source software (or library, for example: .so or .dll) is very challenging.*"

Bugs in the fuzzer lead to build failures or abnormal/inconsistent fuzzing behaviors (F8).

A common challenge we found in the reliability category is that developers could not fuzz due to an issue with the fuzzer itself. This subcategory had a wide range of problems ranging from fuzzers not being able to detect known positive cases, to generating false positives or reporting the wrong type of failure. In many cases, bugs found in the fuzzer crashed fuzzing tests or broke project builds.

> ℹ Developers have had issues where AFL would not be able to compile properly. In such cases, developers ask the fuzzer developers to fix the issue via GitHub issues, as shown in issue #6871 of the OSS-Fuzz repository [48].

Incorrect use of the corpus leads to fuzzing failures or low coverage (F9). This subcategory encompasses cases where developers could not properly set up or use the fuzzing corpora. In such cases, developers sometimes turned to GitHub to get help from more experienced fuzzing developers.

> ℹ In issue #638 of the OSS-Fuzz repository, the developer was not providing the corpus correctly to his fuzzer which made the fuzzer not detect/use it and therefore have poor overall coverage [33].

Issues with the build tools or external dependencies lead to crash/build failures (F10). Fuzzers are often used in combinations with other development tools. In some cases, these external tools can negatively affect the fuzzing process and lead to problems. The sources of crashes or failure are very diverse, which can be, for instance, container tools, compiling tools, or remote storage.

> ℹ For example, developers have had issues building and running their fuzzers in Docker containers due to their Docker image growing too big in size over time, as shown in issue #3548 of the OSS-Fuzz repository [45].

Issues in the corpus lead to build failures or unreliable results (F11). Some developers encountered unreadable or bad data in their corpus which needed debugging and also caused their build to fail.

> ℹ For example, when a corpus contains unreadable or corrupted files, the entire build can fail as a result of a bad corpus as shown in issue #7138 of the OSS-Fuzz repository [50].

Fuzzer unusual inputs cause crashes/issues (F12). Fuzzers can sometimes generate bad inputs which result in more serious crashes. Because these crashes can stop the fuzzing process entirely, developers have discussed possible ways to either handle these problematic inputs or try to avoid them entirely.

> ℹ For example, a developer describes issues he has had with inputs crashing his fuzz target in issue #1622 of the syzkaller repository [54]. In some cases his computer crashed or was rebooted while fuzzing his system.

💡 **Reliability suggestion(s).** To avoid the issues caused by incorrect use of fuzzers and faulty implementations, fuzzing tool developers might need to consider making the tools “simple-to-write”, as suggested by R43. Meanwhile, a plugin or intelligent assistant could hint the existing issues and potential misuse to speed up the fuzzing process and prevent developers from wasting efforts debugging.

3.1.5 Reproducibility Challenges. Challenges related to the reproducibility of the fuzzing process and its results.

Developers are not able to save or re-use corpus (F13). This is another challenge related to corpus management. In this category, we find instances where developers were unable to reuse their corpus after running fuzzers.

> ℹ For example, in issue #52292 of the Go project [24], one developer described that he was not able to reuse the corpus generated by his fuzzer and asks Golang developers how to save the corpus.

Developers are struggling to reproduce build failures or bugs detected by fuzzing (F14). This subcategory is one of the most mentioned types of challenges across all subcategories. We found that developers often encounter flaky crashes while conducting our investigation. In this subcategory, we also find instances where fuzzer logs were not clear or specific enough for the developers to locate or reproduce the issue. Multiple survey respondents also used the free text section to mention that reproducing crashes is a common problem when fuzzing. This type of challenge was the third most common type after F16 and F2 and accounted for almost 11% of our fuzzing challenge issues.

> Most of the GitHub issues labeled in this category were about developers finding and discussing flaky fuzzing crashes. For example, issues #2635 [41] and #6147 [47] of the OSS-Fuzz repository are common examples of flaky crashes caused by flaky memory errors.
> R18: "*Replicating crashes is a major issue.*"
> R63: "*Well I already mentioned in the survey, I would like to emphasize once again that the most difficult part of fuzzing is to reproduce failures/bugs*"

Reproducibility suggestion(s). As suggested by R19, when an issue is detected, it would be helpful for replication to "*see the minimal standalone code example with minimal dependencies to recreate it*". Extracting the minimal standalone code examples are apparently not an easy task, which requires further research and development.

3.1.6 Usability Challenges. This category encompasses all challenges related to the usability of fuzzers, the fuzzing outputs, and the external tools used during fuzzing activities.

Developers have difficulties generating the correct inputs or targeting specific parts of software (F15). Some software and APIs require specific inputs and must pass strict pre-condition checks which pose a challenge for fuzzers trying to generate new inputs. Developers discuss with fellow fuzzing practitioners and contributors to figure out how to generate inputs to fuzz specific parts of the system. Multiple survey respondents mentioned the issue of fuzzers going down "the wrong path" and finding only surface level memory corruption because of it.

> Issue #213 in the libcoap project [69] is a representative example of developers trying to fuzz one of their APIs but are not sure how to proceed.
> R44: "*Fuzzing finds low priority edge cases that have no priority for developers but also block the fuzzing process (in particular pathological OOMs are an issue) [...]. With a huge codebase, a specific fuzzer can easily be mislead into the 'wrong' target code.*"

Developers have difficulties in setting up, building or using fuzzer (F16). While conducting our manual investigation, we found this challenge to be one of the most commonly recurring topics. It encompasses all challenges related to configuring or using a fuzzer. The correct configuration of fuzzing seems to be very challenging, and a wrong configuration can have a wide range of impacts such as breaking the build, not fuzzing the right fuzz target, causing the fuzzer to use too much memory, and many other issues. This was the most commonly encountered problem by practitioners and account for 18% of all issues related to fuzzing challenges.

> In issue #32443 of the ClickHouse project [16], the fuzzer checks out to the wrong commit because the contributor did not properly set it up in the configuration file.
> R26: "*Fuzzing involves getting build systems to work. Build systems are the devil.*".

Documentation on how to use the fuzzer is missing/insufficient (F17). Configuring a fuzzer can be complex and developer often rely on the fuzzer documentation to correctly guide them through the process.

However, there have been multiple instances where developers needed to use specific features of the fuzzers but the arguments managing these features were not documented. In these instances, developers of the fuzzing tool usually had to tell the user about the existence of these arguments in GitHub issues directly. A few survey respondents also reported not having proper documentation for the fuzzers they were using.

For example, in issue #7948 of the solidity project [21], a developer asks for better documentation on how to build fuzzers via docker.
R28: *“specific fuzzer component’s documentation is not provided”.*
R103: *“More transparency and documentation on how the fuzzers concretely work would help the developers know what to do when the behavior is not as expected”.*

Messages/information generated by fuzzers is missing/confusing/unhelpful (F18). Logs and outputs generated by fuzzers are often useful to keep track of what and how well the fuzzer process is doing. However, not all the logs and outputs do their job properly. Some information in the logs and outputs may be incomplete and ambiguous, which provides little help to developers for quickly understanding what is happening and sometimes even misleads them.

In issue #48179 of the go project [23], the fuzzer finds a crashing output but does not report it.

Overloading outputs lead to difficulties to focus on specific issues (F19). During fuzzing, both fuzzers and other involved tools might generate a large amount of outputs. It is sometimes challenging for developers to quickly locate the information they need, spending time disentangling the mixed outputs from various sources.

In issue #57 of the Syzkaller project [53], a developer mentioned that the output of his fuzzer frequently got mixed with other console outputs when running inside a VM.
R3: *“Dealing with the volume of findings can be challenging.”*
R18: *“[...] triaging them (crashes) is another big issue.”*

Usability suggestion(s). There are various ways to increase the usability of fuzzers. For example, integrating fuzzers to standard unit test frameworks (R15) and development environment “like Go has done” (R31) could lower the barrier of entry and increase its adoption. R15 also gave a scenario or such integration: “*for instance, adding a unit test, marking one parameter as fuzzable. the framework could then run that ‘unit test’ with a fuzzer”*. Logging issues are also heavily studied by researchers. More clear and structural messages and outputs would help developers easily navigate through large chunks of information. Configurable logging systems which allows disabling certain logs and outputs could also reduce the workload to identify the relevant parts of information.

3.1.7 Implementation Challenges. The implementation challenges are related to the adaptation of an existing codebase or developing new functionalities for fuzzing purposes.

Bad code design limits the ability to fuzz (F20). In some cases, the architecture of the source code can limit or affect the ability to fuzz the software. More than one survey respondent reported having faced a situation where isolating a specific piece of code for fuzzing was challenging.

In issue #20232 of the Bitcoin project [6], there was a case where a fuzz target was using a global variable defined in the source code. This global variable was however scheduled to be deglobalized as part of a refactoring task and would therefore break some of the fuzz targets.
R43: *“[There is a] difficulty fuzzing internal boundaries in software. [...] Reason: monolithic design, assumptions about intended behavior.”*
R44: *“[...] Insufficient segmentation/separation of the target code (cannot easily carve out parts to test, many tests must be done on the full system) [...]”.*

Developers have difficulties writing/understanding fuzzing code (F21). Fuzzing can be very challenging for developers starting their fuzzing journey. In such cases, beginners sometimes ask for help from more experienced practitioners on GitHub to get started with writing code for their fuzzers.

In issue #1398 of the scapy project [73], a less experienced developer asks help from more experienced fuzzing developers to help him fix bugs in his code so that he can contribute to the public repository .

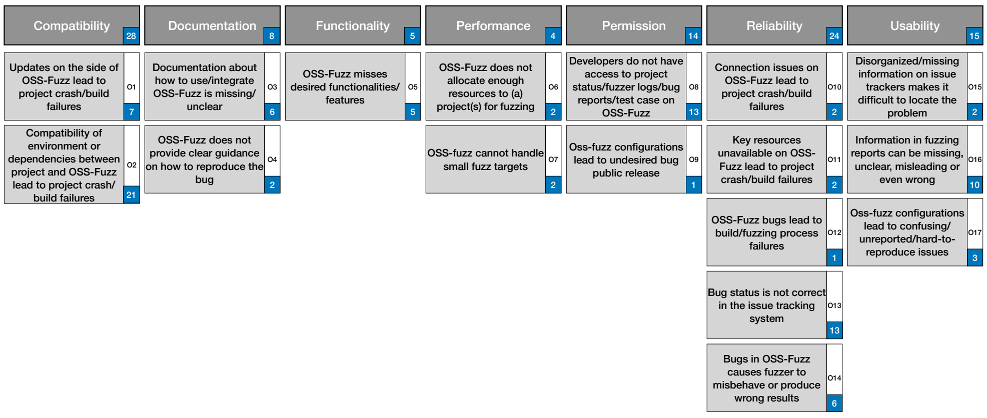
Developers have difficulties in deciding which part of the software to fuzz (F22). Mature repositories often contain very large codebases and legacy code. Contributors of a project therefore need to discuss together on GitHub about which parts of the software need to be covered by the fuzzers due to the loss of knowledge on certain parts of the source code. For projects where multiple parallel builds are being supported, developers also need to decide which builds/releases need to be fuzzed and which ones do not.

In issue #6201 of the Solidity project [20], developers discuss if specific directories should be fuzzed considering that nobody had proper knowledge of this part codebase at the time, and this part of the code might be phased out in the future.

Implementation suggestion(s). To help developers onboard, R58 mentioned that "*guidance on which parts of the software to fuzz would be very helpful.*" R13 also echoed, "*[...] the tooling could suggest interfaces that could be fuzzed; for example, places in the code where network input is read but which haven't already been covered by the existing fuzzing code.*" Researchers and developers could invest more efforts in creating intelligent agents for fuzzing recommendations.

3.2 Challenges from Fuzzing Service Providers

After our manual analysis on fuzzing-related GitHub issues, we identified 97 issues which are related to the challenges developers encounter due to third-party fuzzing service providers, in our case, mainly OSS-Fuzz. Similarly, we extracted 17 different types of challenges from these issues (Figure 3), which can be classified into seven large categories: compatibility, documentation, functionality, performance, permission, reliability, and usability. The numbers in blue squares indicate the occurrences of challenges during our manual analysis.



Compatibility 28
Updates on the side of OSS-Fuzz lead to project crash/build failures O1 7
Compatibility of environment or dependencies between project and OSS-Fuzz lead to project crash/ build failures O2 21
Documentation 8
Documentation about how to use/integrate OSS-Fuzz is missing/ unclear O3 6
OSS-Fuzz does not provide clear guidance on how to reproduce the bug O4 2
Functionality 5
OSS-Fuzz misses desired functionalities/ features O5 5
Performance 4
OSS-Fuzz does not allocate enough resources to (a) project(s) for fuzzing O6 2
OSS-fuzz cannot handle small fuzz targets O7 2
Permission 14
Developers do not have access to project status/fuzzer logs/bug reports/test case on OSS-Fuzz O8 13
Oss-fuzz configurations lead to undesired bug public release O9 1
Reliability 24
Connection issues on OSS-Fuzz lead to project crash/build failures O10 2
Key resources unavailable on OSS-Fuzz lead to project crash/build failures O11 2
OSS-Fuzz bugs lead to build/fuzzing process failures O12 1
Bug status is not correct in the issue tracking system O13 13
Bugs in OSS-Fuzz causes fuzzer to misbehave or produce wrong results O14 6
Usability 15
Disorganized/missing information on issue trackers makes it difficult to locate the problem O15 2
Information in fuzzing reports can be missing, unclear, misleading or even wrong O16 10
Oss-fuzz configurations lead to confusing/ unreported/hard-to-reproduce issues O17 3

Fig. 3. Taxonomy of challenges from fuzzing service providers.

While challenges related to fuzzing service providers are not the main focus of our study and not included in the survey to developers active in fuzzing, we still present each type of challenge with an example, so that the readers better understand the context and the issues developers might have when using a third-party fuzzing service (*i.e.,* OSS-Fuzz). Some of these challenges might share certain similarities with challenges in section 3.1, with the main difference being that here the issues are caused by fuzzing service providers and the fuzzing process works properly on local machines.

3.2.1 Compatibility Challenges. Challenges related to making a project compatible with OSS-Fuzz.

Updates on the side of OSS-Fuzz lead to project crash/build failures (O1). OSS-Fuzz is still under development, thus the users sometimes need to commit or push updates to the platform to maintain the compatibility of their project or support new features. The updates done by OSS-Fuzz can however have unintended impacts on projects being fuzzed. In some cases, new updates can lead to build failures for some projects, which cause them to stop being fuzzed.

> ℹ In issue #153 of the OSS-Fuzz [29] repository, a project developer finds that his build suddenly fails after an argument was renamed by OSS-Fuzz.

Compatibility of environment or dependencies between project and OSS-Fuzz lead to project crash/build failures (O2). In this category, we find instances where a project cannot be built or fuzzed due to mismatching fuzzing environments between OSS-Fuzz and the project. Most issues in this category happen when a project is able to conduct its normal fuzzing activities on a contributor's local machine or within a CI environment but not on OSS-Fuzz. We also found this to bethe most frequent provider challenge during our manual analysis, accounting for close to 21% of all the relevant issues related to external fuzzing providers.

> ℹ In issue #823 of the OSS-Fuzz repository [35], a project was using the latest features offered by a new compiler version which were not yet supported by the OSS-Fuzz environment.

3.2.2 Documentation Challenges. In this category, we identify cases where developers could not find a solution to their issue by consulting the OSS-Fuzz documentation.

Documentation about how to use/integrate OSS-Fuzz is missing/unclear (O3). Efforts are required by project developers to integrate their fuzzing infrastructure with OSS-Fuzz [26]. Some developers rely on the official documentation of OSS-Fuzz to guide them through the process but the documentation might be incomplete or unclear in some cases.

> ℹ In issue #2094 of the OSS-Fuzz repository [39], a developer trying to integrate his project using AFL to OSS-Fuzz mentioned the lack of relevant examples on AFL in OSS-Fuzz's documentation.

OSS-Fuzz does not provide clear guidance on how to reproduce the bug (O4). Developers can sometimes have difficulties trying to reproduce bugs found by OSS-Fuzz fuzzers. In these cases, developers turn to the official OSS-Fuzz documentation to get some guidance about the proper methodology to reproduce a bug. Unfortunately, the official documentation sometimes proves to be not robust enough to assist developers trying to reproduce bugs found by OSS-Fuzz.

> ℹ In issue #1344 [37], a developer asked for guidance from OSS-Fuzz developers due to not being able to reproduce the bug using only OSS-Fuzz's documentation.

3.2.3 Functionality Challenges. In this category, we identify challenges related to the lack of support from OSS-Fuzz as a service for features that developers need to fuzz their software.

OSS-Fuzz misses desired functionalities/features (O5). The OSS-Fuzz platform can sometimes lack features either by design decisions or because the platform is still under development.

This category encompasses issues where project developers needed a specific feature which was not supported by OSS-Fuzz to conduct their fuzzing activities.

> ℹ *For example, in issue #2558 [40], a developer asked OSS-Fuzz developers to build support for multiple sanitizers when reproducing bugs since it takes a lot of time to rebuild the project multiple times using different sanitizers to reproduce issues.*

3.2.4 Performance Challenges. This category encompasses performance challenges for fuzzing activities on OSS-Fuzz.

OSS-Fuzz does not allocate enough resources to (a) project(s) for fuzzing (O6). The OSS-Fuzz team takes care of assigning CPU resources and disk space to each project, with the purpose of ensuring that all projects regardless of their size can be fuzzed. When not enough resources are dedicated to a project, the project might not have all of its fuzz targets fuzzed or not be continuously fuzzed.

> ℹ In issue #3014 [42], a developer finds out that little CPU time was dedicated to their project and wonders why their project is not being fuzzed continuously by OSS-Fuzz.

OSS-Fuzz cannot handle small fuzz targets (O7). When a project's build is failing or is not being fuzzed, OSS-Fuzz developers will generally investigate the cause of the issues. In some cases, the OSS-Fuzz team found that the root of the problem came from the fuzz targets being too small for OSS-Fuzz fuzzers to fuzz.

> ℹ In issue #1333 of the OSS-Fuzz repository [36], an OSS-Fuzz developer found that one of the project's builds is failing due to a bad fuzz target. After further investigation with the project's developers, they found that it was due to one of the fuzz targets being too small.

3.2.5 Permission Challenges. We present challenges project developers face due to OSS-Fuzz permission-related issues.

Developers do not have access to project status/fuzzer logs/bug reports/test case on OSS-Fuzz (O8). Several fuzzing logs/reports and newly discovered bugs are only available to be seen by project maintainers before being released to the general public. Project developers are sometimes not registered as project maintainers or cannot access the logs and must ask OSS-Fuzz developers to grant them the permissions they need to access the logs for bug fixing. This type of challenge is also widely encountered by developers, and accounts for 13% of all the manually analyzed issues related to fuzzing providers.

> ℹ In issue #243 [30] of the OSS-Fuzz repository, a developer found out that he was not able to access one of fuzzer UIs when using an alternative email address from the one he uses for OSS-Fuzz.

OSS-Fuzz configurations lead to undesired bug public release (O9). OSS-Fuzz has a clear bug disclosure policy as described in its official documentation [25]. In some cases, the strict bug disclosure policy of OSS-Fuzz might not align with project developers' interests as they might require more time before public disclosure of a bug or vulnerability.

> ℹ In issue #244 [31], a developer asked OSS-Fuzz developers to change their policy for public disclosure of bugs for their project so that the issues would only be disclosed once the fixes are pushed to their project's stable branch.

3.2.6 Reliability Challenges. This category encompasses all challenges related to the reliability of the OSS-Fuzz platform or the environment and services it provides.

Connection issues on OSS-Fuzz lead to project crash/build failures (O10). OSS-Fuzz must deal with several external dependencies such as dependencies downloaded during a project's compilation or dependencies in the form of external tools used by the platform. All of these dependencies can be subject to connection issues. In some cases, these issues can impact fuzzing activities and cause unintended failures.

> ℹ In issue #7581 [52], a developer reported having a lot of build failures in OSS-Fuzz due to connection failures.

Key resources unavailable on OSS-Fuzz lead to project crash/build failures (O11). Multiple dependencies and files can be required during a project's compilation or during fuzzing. Any of these resources missing or being unavailable will typically cause problems such as breaking a project's build.

> ℹ For example, in issue #7198 [51], a developer reported that his build was failing due to corpus files not being available on OSS-Fuzz.

OSS-Fuzz bugs lead to build/fuzzing process failures (O12). OSS-Fuzz takes care of a lot of different tasks such as managing fuzzers, executing fuzzers, and managing each project's corpus. When one of the platform's features has bugs, it can lead to various issues preventing a project from being fuzzed properly.

> ℹ In issue #3082 [43], a bug on OSS-Fuzz caused problems with corpus pruning tasks and broke the coverage build.

Bug status is not correct in the issue tracking system (O13). A recurrent topic in the OSS-Fuzz GitHub repository is issue management in the bug tracker. When project developers already fixed a bug or are in the process of fixing a bug, there can be inconsistencies between the actual status of the bug and the bug status being shown in the bug tracker. This challenge is also fairly common and is present in 13% of all the manually analyzed issues related to fuzzing providers.

> ℹ In issue 5526 [28], a developer reported that multiple issues that were previously reproducible and subsequently fixed where still being labeled as "flaky" on the OSS-Fuzz bug tracker.

Bugs in OSS-Fuzz causes fuzzer to misbehave or produce wrong results (O14). OSS-Fuzz developers need to properly configure their fuzzers for each project fuzzed by OSS-Fuzz. Even though the OSS-Fuzz team consists of fuzzing experts, errors in the configuration can happen and impact the validity of the fuzzing outputs. The various bots and automation tools used by OSS-Fuzz to fuzz a project can also contain bugs resulting in undesired fuzzing results/outputs.

> ℹ While discussing how OSS-Fuzz should handle timeouts and OOM errors in issue #3432 [46], a developer mentioned that he had instances where the CIFuzz jobs timed out due to infinite loops without reporting any problem.

3.2.7 Usability Challenges. In this category, we describe obstacles related to the usability of OSS-Fuzz as a fuzzing provider for project developers.

Disorganized/missing information on issue trackers makes it difficult to locate the problem (O15). OSS-Fuzz currently uses the Chromium bug tracker Monorail [15] to track bugs found by OSS-Fuzz. Using bug trackers can be a challenge for developers who are not familiar with the UI, the naming conventions, or the templates used by a bug tracker to describe a bug.

> ℹ In issue #273 [32], developers discussed about information that should be included in a bug report in the issue tracker. In the current version, it was unclear which revisions were still affected by the bug.

Information in fuzzing reports can be missing, unclear, misleading or even wrong (O16). Project developers use fuzzing logs and reports for debugging or monitoring fuzzing activities and fuzzing coverage for their projects. Logs and reports provided by OSS-Fuzz can however sometimes be incorrect, missing or unclear as to what is the problem. This can make it difficult for developers trying to debug their project or trying to get an overview of their project's coverage from OSS-Fuzz reports.

> ℹ For example, in issue #6885 [49], a developer had a problem where the OSS-Fuzz fuzzer found a bug but the reproducer test case was either not generated or not uploaded. The logs generated by the fuzzer also contained misleading information about what happened to the crash report.

OSS-Fuzz configurations lead to confusing / unreported / hard-to-reproduce issues (O17). There can be several differences in how a project's fuzzers are configured versus how OSS-Fuzz developers configure their fuzzers. Different configurations can determine how strict fuzzers are in terms of which errors can be ignored versus which errors need to be reported. These differences in configuration can lead to confusion for project developers who might want to address any error regardless of their importance. For instance, a project developer might want to fix a smaller type of error such as integer overflow whereas OSS-Fuzz might choose to silence integer overflow by default and therefore not report any error as shown in issue #3227 of the OSS-Fuzz repository [44].

> ℹ In issue #6587 [22], a developer discovered that some bugs found by the fuzzer were ignored and not treated as errors.

3.3 Survey Result Analysis

3.3.1 Frequency of Fuzzing Challenge Occurrences in the Survey. Figure 4 shows the responses of survey participants when asked about the frequency at which they encounter each challenge in our taxonomy. The most common challenge for fuzzing developers is by far to generate the right inputs to fuzz the system (F15) with 15% of the survey respondents stating that this is a constant challenge and 40% stating that they often encounter issues trying to generate inputs for their system. This is not surprising since software systems can require strict pre-conditions to accept an input and can also have paths in the source code that are difficult to reach for fuzzers. Challenges linked directly to the fuzzers also seem to be very common: over 70% of our respondents have had issues where the fuzzer was missing certain features to fuzz the system properly. Practitioners trying to scale their fuzzers was also the challenge with the highest amount of "Always" with 17% of the respondents stating that scaling their fuzzers was always a challenge and 21% stating that it's often a challenge.

Another trend worth noting is that 60% of our respondents have had issues with fuzzing breaking their system's build (F2, F10, F16). We also find that the most common causes of build breakages (F1, F10, F16) are all related in some way to the configuration of the fuzzer(s). Fuzzers often require precise configuration based on a variety of factors such as the hardware they are running on, the version of the compiler, the programming language and many others. Since most of the respondents surveyed had less than 5 years of fuzzing experience (as shown in Table 1), it is not surprising that configuration challenges were quite common for developers still developing their fuzzing expertise.

With over 60% of our respondents stating that they have faced issues with fuzzing taking too much time and resources, our survey also confirms the need for ways to fuzz systems externally. Both initiatives such as OSS-Fuzz which can fuzz a project on Google's servers and CI tools which can support continuous fuzzing can help alleviate the hardware and human workload associated with fuzzing. In the free text section of the survey, resource consumption issues, the need for better tooling, and the lack of support for continuous fuzzing from CI tools were all challenges mentioned by the respondents.

3.3.2 Frequency Comparison Between Survey and Manual Analysis. From our manual analysis (Figure 2), difficulties in setting up, building or using fuzzer (F16), incompatibility between fuzzing

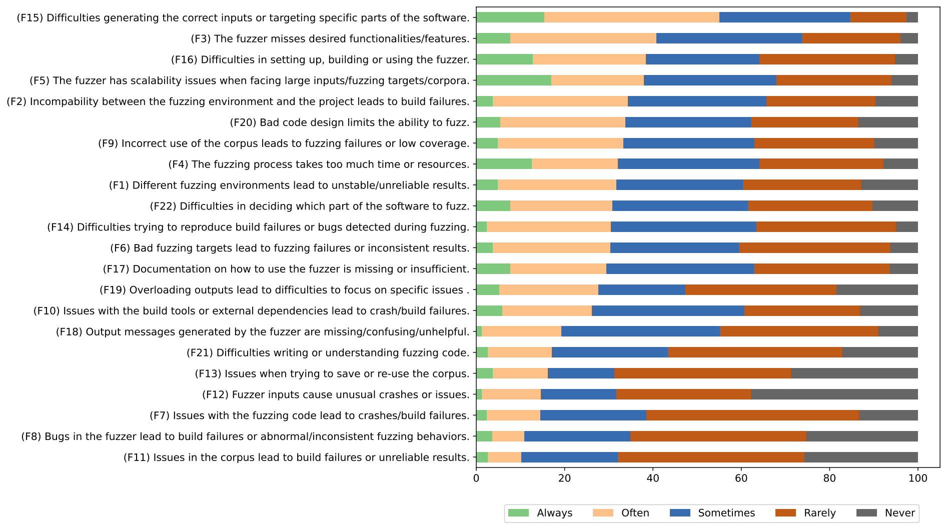


Fig. 4. Frequency at which the respondents encounter each challenge in percentage.

environment and project (F2), and difficulties in reproducing build failures or bugs (F14) are the most common challenges, which have much more occurrences than other issues and account for around 18%, 12%, and 11% of all issues, respectively. This is to a certain extent supported by our survey results. The difficulties of fuzzer usage (F16) and compatibility issues (F2) are among top 5 most common challenges according to survey responses (Figure 4). While the reproducibility issue is not at the top spot, it is still fairly common (over a quarter of respondents indicate that they always or often encounter this issue). It is worth noting that our manual analysis and survey mainly focus on the prevalence of the challenges instead of the difficulties. In fact, Respondent R63 specifically emphasizes that *"the most difficult part of fuzzing is to reproduce failures/bugs"*.

There are also some differences regarding the challenge frequency ranks in manually analyzed issues and survey responses. For example, in the survey, difficulties in generating the correct inputs or targeting specific parts of software (F15) are considered the most prevalent challenge by developers in the survey, while in the manual analysis, this category of challenge only accounts for 3.5% of all the issues and ranks at 9th in terms of frequency. Similarly, many developers complain that bad code design limits their ability to fuzz software systems (F20) in the survey, while only 2 manually analyzed issues mention this challenge. One reason for these differences can be attributed to the nature of these two sources. On GitHub issues, developers tend to discuss straightforward technical problems (*e.g.*, build failure, setup troubleshooting), while those issues which do not directly lead to failures are less likely to be mentioned (F15 and F20). We believe the survey results can well complement the manual issue analysis and help readers to have a more complete perspective on these issues.

3.3.3 Impact of Developer Experience on the Results. Developers with different levels of fuzzing experience might perceive challenges differently. To investigate the extent to which this happens, we looked at how the survey responses change based on the fuzzing experience. We split the survey participants into four groups: "beginner" (<3 years of fuzzing experience), "intermediate" (3-5 years of experience), "experienced" (6-10 years of experience), and "expert" (>10 years of fuzzing experience). The graphs of frequency at which the respondents of different groups encounter each challenge can be found in our replication package [68]. Due to the low number (6) of experts participating in the survey, below we only discuss the similarities and differences among the remaining three groups.

Regardless of experience, missing functionalities of fuzzers (F3) and difficulties in generating correct inputs or targeting specific parts of software (F15) are the most common issues for all the groups. Meanwhile, failures caused by problematic data in the corpus (F11), crashes caused by generated inputs (F12), and the difficulties of corpus reuse (F13) are the least frequent encountered challenges across all respondents.

In spite of these similarities, there are also some notable differences. Insufficient documentation (F17) is considered a common issue only by the "beginner" group, and "beginners" have much more difficulties in fuzzer setup and usage (F16) compared to other groups. One possible reason is that fuzzing has a steep learning curve and better on-boarding support would be highly appreciated by "beginners". On the contrary, fuzzing failure or inconsistent results caused by bad fuzzing targets (F6) are rather common for "intermediate" and "experienced" groups, while it is not a dominant challenge for the "beginner" group. One possible reason for this difference is that developers with more fuzzing experience have higher chances to fuzz more complex software systems. Further studies are needed to confirm this hypothesis.

3.3.4 Feedback from Practitioners on Current Fuzzing Solutions and the Field of Fuzzing. In the survey, some respondents provided feedback on challenges and potential solutions in the free-text questions. These opinions mainly focus on three aspects: applicability of new tools, performance and resource issues, and barrier of entry.

Applicability of new tools. According to R36 and R90, while academia tries to address some of these challenges by developing new fuzzing tools, these tools often fall short of features, replicability, or documentation. Moreover, they often do not reach production. Similarly, regarding the applicability of these tools, R65 suggests that simple fuzzing tools focusing on small specific issues should be developed first instead of trying to make them as generic as possible: *"There's an effort currently to create generic fuzzers that touch upon a lot of categories. I'd focus on a smaller, maybe niche area, and create pretty good fuzzing primitives that can be then put together on a per-project basis."*

Performance and resource issues. The issue of performance and resources during fuzzing is also quite prevalent in our survey analysis. Currently, one widely adopted solution is to use an external provider (*e.g.,* OSS-Fuzz) that takes over the workload. However, as R69 however points out, there are currently very limited options available to practitioners who want to use an external provider: it is *"hard to set up on larger scale by yourself (no widely used Kubernetes setups, GitLab gates their fuzzing feature behind their most expensive tier, GitHub doesn't even have something like this, Google operates OssFuzz as SaaS)".*

While the availability of external providers might need further improvement, overall, using an external service provider does seem to alleviate performance problems, and OSS-Fuzz generally seems to be an initiative appreciated by practitioners. For example, respondent R85 says: *"My overall experience with ossfuzz is quite good. Having something that is managed [...] and that only requires my attention when bugs are filed makes fuzzing [...] a reasonably good [experience]".* R67 further supports this sentiment and mentions the benefit of not needing to set up fuzzing infrastructure from scratch: *"Our project's fuzzing is hosted by OSS-Fuzz (Google-maintained infrastructure), which helps a lot on many of the issues mentioned in this survey. Setting up a fuzzing infrastructure from scratch must be a daunting task.".* As R75 points out, however, some improvements to the platform are still needed to help practitioners with the entire fuzzing process: *"We need anything platforms* *that assist us in the whole fuzzing lifecycle: setup and continuously run fuzzing, alert about findings, rerun fuzzing when fix is available".*

Barrier of entry. Another recurrent aspect is about the adoption of fuzzing itself. As of now, the barrier of entry to fuzzing seems quite high, which limits the adoption of fuzzing as a standard practice. Survey respondent R15, who has 6-10 years' experience, suggests integrating fuzzing directly into unit test frameworks to *"lower the barrier for developers to join in on fuzzing, instead of having dedicated persons writing fuzzers".* Respondent R13 with 6-10 years' experience and respondent R90 with over 10 years' experience further supports this argument by stating that tooling should *"make it easy for all developers on a project to contribute to fuzzing at a low-level"* and *"typical developer, doesn't have the skill set to analyze their code for interesting parts in the context of fuzzing. So fuzzing engineers end up doing a lot of work on analysing the target code and behavior".*

One possible reason for the current high barrier of entry for fuzzing is tooling related to fuzzing practices. As respondent R12 with over 10 years of fuzzing experience suggests, *"[fuzzing needs] more porcelain tooling, we just get the plumbing today and I use ad hoc support scripts now".* Even experienced fuzzing practitioners are relying on custom made scripts instead of having proper tooling to help them conduct fuzzing activities. It is highly likely that writing these "support scripts" require deep domain knowledge which beginners might do not have. It is therefore important to develop standardized tools so that the "quality of life" improvements brought by custom scripts can be generalized across the whole field via tooling.

4 THREATS TO VALIDITY

Threats to construct validity concern the relationship between theory and observation. In this study, to collect GitHub issues related to fuzzing activities for our manual analysis, we used the keywords "afl" and "fuzz" for issue searching. We acknowledge that some relevant issues, which are not described with specific keywords, might be filtered out. In our survey, we offered 5-level Likert scale options (*i.e.,* "Always", "Often", "Sometimes", "Rarely" and "Never") to survey participants when answering how common each type of challenge is. While this is a standard setting in surveys, these options are still subject to different interpretations, which might lead to biases in the results. As it is impractical to strictly define these options by quantification given the various sizes and scales of software systems developers are working on, we believe they can still accurately capture the relative prevalence of different challenges, which well serves the purpose of this study.

Threats to internal validity concern factors internal to our study that could have influenced our results. In our study, it is mainly related to the possible subjectiveness during our manual analysis and taxonomy creation. The annotation of challenges contained in GitHub issues is a subjective process based on the author's understanding of the problem and, despite multiple years of programming experience, the authors might not have all the relevant domain knowledge of the fuzzing tools used. To mitigate the impact of the potential biases, three authors independently investigated and classified each issue without consulting each other until the end of an iteration in which conflicts were resolved via unanimous agreement.

Threats to external validity concern the generalizability of our findings. The repositories we used are all open source projects fuzzed by Google's OSS-Fuzz. Other projects using different fuzzing providers or not using any fuzzing provider may encounter additional fuzzing challenges not present in projects participating in OSS-Fuzz. Also, although OSS-Fuzz supports state-of-the-art fuzzers [27], there are dozens of other publicly available or even self-created fuzzing tools used by developers, which may lead to their own set of challenges. On a similar note, OSS-Fuzz currently supports only certain programming languages. We may therefore miss out some challenges related to fuzzing projects written in an unsupported languages such as web applications developed in JavaScript and NodeJS.

5 RELATED WORK

5.1 Studies on Fuzzing Techniques

Developing fuzzing techniques is non-trivial, as different software systems require different types of inputs which must satisfy pre-conditions before the software accepts them. Developing specialized fuzzing tools to fuzz all sorts of software and hardware such as APIs [55] or microcontrollers [62] has therefore been an active field of research for many years.

Many researchers are actively developing new fuzzing techniques. In an ideal scenario, a fuzzer would be able to have perfect coverage by generating inputs that can trigger every possible edge of a codebase. To improve the discovery of new paths in the source code during fuzzing, some studies have proposed new ways to generate inputs and improve the coverage of fuzzers. Menendez *et al.* [65] developed the HashFuzz technique in order to have more diversity of covered paths in the test sets generated by fuzzers. Using HashFuzz, they find that fuzzers achieve better coverage and observe an increase of 28% to 97% in unique crash detection. Nguyen *et al.* [67] developed the BeDivFuzz feedback-driven fuzzing technique which takes into account how many times each branch of the source code was executed. Using BeDivFuzz, Nguyen *et al.* were able to achieve better branch coverage diversity by preventing a fuzzer from generating inputs that always target the same branches of code. Padhye *et al.* [70] proposed the FuzzFactory framework, which improves coverage-guided fuzzing by saving specific inputs generated during fuzzing so that the generated inputs can easily be reused to augment the coverage and generate more complex inputs.

Other studies have looked into fuzzing techniques from a different perspective more than coverage. The idea of mixing a black-box and a white-box (or coverage-based) fuzzing approaches has led to the design of graybox fuzzing and its first implementation AFLGo [9]. Graybox fuzzing aims to balance between getting better input generation than purely random inputs without requiring complete knowledge of the codebase. Since AFLGo, other researchers have also tried to improve graybox fuzzing by proposing their own tools and variations of the underlying input generation algorithm [19, 66]. Zhou *et al.* [82] proposed a new approach VisFuzz, which leverages extracted call graphs of the codebase to guide test engineers to locate blocking constraints and design better inputs for these constraints. VisFuzz can achieve coverage improvements between 10% and 150%.

Researchers have also tried to tackle the challenge of fuzzing domain-specific software and improve the fuzzing effectiveness with semi-automated techniques which require limited human intervention. For example, Shoshitaishvili *et al.* [75] proposed an approach which can automatically extract the parts of software and assign them to developers with proper expertise for vulnerability analysis. Their results demonstrate that humans, even without intensive training, can significantly improve the performance of fuzzing-based vulnerability analysis techniques. Aschermann *et al.* [3] adopted an annotation-based approach, which allows developers to annotate the source code to guide fuzzers during the execution. Their implementation is able to resolve complex patterns (*e.g.*, hash map lookups), play games and find novel security issues via fuzzing.

With the increased usage of deep learning based techniques in recent years, multiple fuzzing tools focusing on finding bugs in deep learning frameworks have been developed. Wei *et al.* [77] developed `FreeFuzz`, which aims at fuzzing deep learning libraries via mining open source software repositories. Gu *et al.* [56] developed `Muffin`, which can generate deep learning models via fuzzing in order to exercise and test deep learning libraries. With the advent of self-driving and image processing techniques, a new fuzzing tool aimed at fuzzing image processing deep learning libraries has also been proposed by Zhang *et al.* [81].

5.2 Studies on Fuzzing Challenges

Prior to our work, several empirical studies have been conducted to understand the fuzzing process and shed light on certain fuzzing challenges. Yun *et al.* [79] conducted a thorough survey of fuzzing in the field of embedded systems. In the paper, they list the current tools and techniques used to fuzz embedded systems and discuss future paths to improving fuzzing techniques for embedded systems. Ding *et al.* [18] conducted an empirical study using OSS-Fuzz data on bug fixing. In their study, they document what types of bugs are found by continuously fuzzing via OSS-Fuzz and the time required to detect bugs via OSS-Fuzz. They also investigate which bugs are prioritized for bug fixing by developers.

Böhme *et al.* conducted another empirical study on fuzzing where they measured the decreasing return of investment over time of fuzzing as more time is needed to find bugs over time [8]. In their study, Böhme *et al.* find that increasing the number of machines dedicated to fuzzing is neither sufficient nor a good approach to finding more vulnerabilities over time. Instead, they state that developing smarter and more efficient fuzzers is the key to finding more vulnerabilities and improving performances. The same phenomenon is observed in another study led by Klooster *et al.* [60] where they find that shorter sessions of fuzzing of 15 minutes can be sufficient to uncover important bugs while being less resource intensive than long fuzzing sessions. These studies provide hints on how to address the challenge of performance issues, which were well documented in our taxonomy (F4).

The adoption of fuzzing as a practice is another topic that has been gaining interest for practitioners and researchers. Kelly *et al.* [59] tried to alleviate the efforts needed to integrate fuzzing to a codebase by automatically generating fuzz targets. Using their approach, they were able to quickly generate dozens of fuzz targets without additional developer efforts. Their approach can be used to tackle usabiliity issues (F15 in our taxonomy) to a certain extent. Liang *et al.* [64] conducted a case study at Huawei where engineers tried to start fuzz testing one of their libraries. In their study, they presented the main obstacles faced by Huawei engineers and described how they overcame some of the fuzzing challenges the team encountered. The authors also discussed some of the lessons learned from the team after integrating fuzzing into their internal systems. They found that the strict environmental constraints expected by fuzzers pose a problem for industrial environments which might be using different operating systems, compilers and hardware. This finding is in line with our compatibility challenges (F1 and F2). They also found that there is a lack of support from build systems and that training is required before software testers/developers become sufficiently knowledgeable about fuzzing to properly integrate fuzzing into a software system, which is highly relevant to one of the usability issues presented in our taxonomy (F16).

In a previous Shonan meeting [7], fuzzing experts gathered and discussed the current state of fuzzing. During this meeting, they summarized some of the challenges of fuzzing and the limitations of current fuzzers for identifying relevant bugs. They also discussed the current theories being worked on in the field of fuzzing. Finally, the meeting participants conducted a small survey with 24 respondents and found that improving automation is the most important challenge in the field of fuzzing. Yan *et al.* [78] reached a similar conclusion when they investigated the role of humans in fuzzing activities by categorizing popular fuzzing tools into three different types: human-out-of-the-loop, human-on-the-loop, and human-in-the-loop. In their paper, they discussed about possible directions to improve fuzzing tools based on human-machine collaboration and hypothesized that automation must be improved pre-fuzzing to make fuzzers easier to use and post-fuzzing to analyze the vulnerabilities found. These two studies have a different focus regarding the challenges, that is, they look into the challenges of improving existing fuzzing techniques instead of the challenges developers face during fuzzing activities (*e.g.,* how to improve automation of fuzzer vs. what issues developers face during fuzzing automation).

Our taxonomy and survey complement some of the prior empirical work conducted in the field of fuzzing. Multiple of our respondents' answers align with the results obtained by previous studies. For example, some of our survey respondents mentioned that bug triaging and bug fixing are challenges that need improvement in fuzzing (F19) which align with the findings in Ding *et al.*' study [18] showing that fuzzers are finding thousands of bugs. Several respondents also mentioned the lack of specialized features and the inability of fuzzers to fuzz in a structure-aware manner (F3). This is further confirmed by Böhme *et al.* in their work on the performance challenge related to fuzzing [8] where they recommend to develop better fuzzing tools and techniques rather than to add more machines dedicated to fuzzing.

To the best of our knowledge, our study is the first to systematically derive the categories of challenges developers face during fuzzing activities, by both qualitative and quantitative means. The presented taxonomy was created based on evidence embedded in issue discussions and further confirmed by developers with relevant expertise, which by far gives the most complete overview regarding this issue. On one hand, this study confirms fragmented and specific challenges which can be seen in related work; On the other hand, the presented challenges contain many new findings which were not already disclosed by previous studies (*e.g.,* documentation issues, misleading messages, fuzzing provider challenges). We therefore hope that our survey and our manual investigation can serve as a reference points for fuzzing practitioners and researchers looking to develop better fuzzing tools and approaches.

6 CONCLUSIONS

As software systems are growing larger and becoming more complex, the number of software vulnerabilities will only continue to increase over time. Consequently, the adoption of fuzzing as a means to detect software vulnerabilities should become more common over time as well. Industrial practitioners have already started integrating fuzzing as part of their internal testing practices as shown by the case study led by Liang *et al.* [64].

To provide practitioners and researchers with insights on fuzzing improvements, we conducted a study to analyze the challenges associated with fuzzing activities. We constructed a taxonomy of fuzzing challenges and validated it by surveying fuzzing practitioners, collecting their feedback, and asking them directly what challenges they have themselves faced while fuzzing their software. Our taxonomy, accompanied with real-world examples and potential solutions for addressing the challenges, can support both practitioners and researchers. For software testers and software developers looking to conduct fuzzing activities, our taxonomy allows them to be aware of the challenges they might face while fuzzing their software.

Using our taxonomy as a reference point, we hope potential fuzzing adopters can plan ahead of time on how to handle the challenges highlighted in our study and, consequently, smoothly integrate fuzzing with their existing testing infrastructure. For practitioners who are already fuzzing their software and are looking to use an external provider such as OSS-Fuzz to reduce fuzzing costs or scale up their fuzzing, we also highlight some of the challenges they might encounter which are not present on their own machines.

For researchers, our taxonomy highlights areas of the field which need improvements according to fuzzing practitioners. Our survey respondents' answers reveal the need for more specialized fuzzing tools. Besides, intelligent agents integrated in developers' workflow would also help practitioners better adopt fuzzing tools, which can be investigated more by researchers. Bug triaging and integration of fuzzers with CI systems, for instance, is another promising direction to look into.

Finally, our study provide empirical researchers another stepping stone to improve the current empirical body of knowledge with respect to the field of fuzzing.

Our future work will focus on two directions. First, we would like to expand our research scope and investigate challenges of fuzzing projects developed in a programming language less prominent in the fuzzing field. Second, we would like to analyze fuzzing challenges in detail and try to propose approaches to address some of the current issues. Given the various insights disclosed in this study, we believe there is plenty of room for improving fuzzing techniques and practices, which will in the end benefit both industry and academia.

REPLICATION

To facilitate future work, we have made available online the anonymized survey responses and the result of the manual labeling process in our replication package [68].

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of JSPS and SNSF for the project SENSOR (No. 183587, JPJSJRP20191502), JSPS for the KAKENHI grants (JP21H04877, JP22K18630, JP21K17725), JST for the PRESTO grant (JPMJPR22P3), Kyushu University for the Leading Human Resource Development Fellowship, and the Inamori Research Institute for Science (Kyoto, Japan) for supporting Yasutaka Kamei via the InaRIS Fellowship.

REFERENCES

[1] Anaconda. 2022. State of Data Science Report 2022. https://www.anaconda.com/state-of-data-science-report-2022
[2] Oxford Analytica. 2021. Apache software flaw could result in major breaches. *Emerald Expert Briefings* oxan-es (2021).
[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP'20)*. 1597–1612.
[4] bitcoin/bitcoin. 2020. Issue #19557. https://github.com/bitcoin/bitcoin/issues/19557. Last accessed on 07-12-2022.
[5] bitcoin/bitcoin. 2020. Issue #20088. https://github.com/bitcoin/bitcoin/issues/20088. Last accessed on 07-12-2022.
[6] bitcoin/bitcoin. 2020. Issue #20232. https://github.com/bitcoin/bitcoin/issues/20232. Last accessed on 07-12-2022.
[7] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021), 79–86.
[8] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'20)*. 713–724.
[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2329–2344.
[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. 1032–1043.
[11] bytecodealliance/wasmtime. 2019. Issue #459. https://github.com/bytecodealliance/wasmtime/issues/459. Last accessed on 07-12-2022.
[12] Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. 2014. Heartbleed 101. *IEEE Security & Privacy* 12, 4 (2014), 63–67.
[13] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15)*. 725–741.
[14] Peter M Chisnall. 1993. Questionnaire design, interviewing and attitude measurement. *Journal of the Market Research Society* 35, 4 (1993), 392–393.
[15] Chronium. [n. d.]. Monorail, Chronium. https://bugs.chromium.org/. Last accessed on 07-12-2022.
[16] ClickHouse/ClickHouse. 2021. Issue #32443. https://github.com/ClickHouse/ClickHouse/issues/32443. Last accessed on 07-12-2022.
[17] Wayne W Daniel and Chad L Cross. 2018. *Biostatistics: a foundation for analysis in the health sciences.* Wiley.
[18] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. https://arxiv.org/abs/2103.11518
[19] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 2440–2451.
[20] ethereum/solidity. 2019. Issue #6201. https://github.com/ethereum/solidity/issues/6201. Last accessed on 07-12-2022.

[21] ethereum/solidity. 2019. Issue #7948. https://github.com/ethereum/solidity/issues/7948. Last accessed on 07-12-2022.

[22] ethereum/solidity. 2021. Issue #6587. https://github.com/ethereum/solidity/issues/6587. Last accessed on 07-12-2022.

[23] golang/go. 2021. Issue #48179. https://github.com/golang/go/issues/48179. Last accessed on 07-12-2022.

[24] golang/go. 2022. Issue #52292. https://github.com/golang/go/issues/52292. Last accessed on 07-12-2022.

[25] Google. [n. d.]. Geting Started, Bug disclosure guidelines, OSS-Fuzz. https://google.github.io/oss-fuzz/getting-started/bug-disclosure-guidelines/. Last accessed on 07-12-2022.

[26] google. [n. d.]. Geting Started, OSS-Fuzz. https://google.github.io/oss-fuzz/getting-started/. Last accessed on 07-12-2022.

[27] Google. [n. d.]. OSS-Fuzz. https://google.github.io/oss-fuzz/

[28] google/oss fuzz. [n. d.]. Issue #5526. https://github.com/google/oss-fuzz/issues/5526. Last accessed on 07-12-2022.

[29] google/oss fuzz. 2016. Issue #153. https://github.com/google/oss-fuzz/issues/153. Last accessed on 07-12-2022.

[30] google/oss fuzz. 2017. Issue #243. https://github.com/google/oss-fuzz/issues/243. Last accessed on 07-12-2022.

[31] google/oss fuzz. 2017. Issue #244. https://github.com/google/oss-fuzz/issues/244. Last accessed on 07-12-2022.

[32] google/oss fuzz. 2017. Issue #273. https://github.com/google/oss-fuzz/issues/273. Last accessed on 07-12-2022.

[33] google/oss fuzz. 2017. Issue #638. https://github.com/google/oss-fuzz/issues/638. Last accessed on 07-12-2022.

[34] google/oss fuzz. 2017. Issue #815. https://github.com/google/oss-fuzz/issues/815. Last accessed on 07-12-2022.

[35] google/oss fuzz. 2017. Issue #823. https://github.com/google/oss-fuzz/issues/823. Last accessed on 07-12-2022.

[36] google/oss fuzz. 2018. Issue #1333. https://github.com/google/oss-fuzz/issues/1333. Last accessed on 07-12-2022.

[37] google/oss fuzz. 2018. Issue #1344. https://github.com/google/oss-fuzz/issues/1344. Last accessed on 07-12-2022.

[38] google/oss fuzz. 2018. Issue #1675. https://github.com/google/oss-fuzz/issues/1675. Last accessed on 07-12-2022.

[39] google/oss fuzz. 2019. Issue #2094. https://github.com/google/oss-fuzz/issues/2094. Last accessed on 07-12-2022.

[40] google/oss fuzz. 2019. Issue #2558. https://github.com/google/oss-fuzz/issues/2558. Last accessed on 07-12-2022.

[41] google/oss fuzz. 2019. Issue #2635. https://github.com/google/oss-fuzz/issues/2635. Last accessed on 07-12-2022.

[42] google/oss fuzz. 2019. Issue #3014. https://github.com/google/oss-fuzz/issues/3014. Last accessed on 07-12-2022.

[43] google/oss fuzz. 2019. Issue #3082. https://github.com/google/oss-fuzz/issues/3082. Last accessed on 07-12-2022.

[44] google/oss fuzz. 2020. Issue #3227. https://github.com/google/oss-fuzz/issues/3227. Last accessed on 07-12-2022.

[45] google/oss fuzz. 2020. Issue #3548. https://github.com/google/oss-fuzz/issues/3548. Last accessed on 07-12-2022.

[46] google/oss fuzz. 2021. Issue #3432. https://github.com/google/oss-fuzz/issues/3432. Last accessed on 07-12-2022.

[47] google/oss fuzz. 2021. Issue #6147. https://github.com/google/oss-fuzz/issues/6147. Last accessed on 07-12-2022.

[48] google/oss fuzz. 2021. Issue #6871. https://github.com/google/oss-fuzz/issues/6871. Last accessed on 07-12-2022.

[49] google/oss fuzz. 2021. Issue #6885. https://github.com/google/oss-fuzz/issues/6885. Last accessed on 07-12-2022.

[50] google/oss fuzz. 2022. Issue #7138. https://github.com/google/oss-fuzz/issues/7138. Last accessed on 07-12-2022.

[51] google/oss fuzz. 2022. Issue #7198. https://github.com/google/oss-fuzz/issues/7198. Last accessed on 07-12-2022.

[52] google/oss fuzz. 2022. Issue #7581. https://github.com/google/oss-fuzz/issues/7581. Last accessed on 07-12-2022.

[53] google/syzkaller. 2016. Issue #57. https://github.com/google/syzkaller/issues/57. Last accessed on 07-12-2022.

[54] google/syzkaller. 2020. Issue #1622. https://github.com/google/syzkaller/issues/1622. Last accessed on 07-12-2022.

[55] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-Aware Dataflow Graphs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1070–1081.

[56] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1418–1430.

[57] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale. 2018. Software testing: Survey of the industry practices. In *Proceedings of the 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO'18)*. 1449–1454.

[58] ISO/IEC. [n. d.]. ISO 25000 Portal. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[59] Matthew Kelly, Christoph Treude, and Alex Murray. 2019. A Case Study on Automated Fuzz Target Generation for Large Codebases. In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'19)*. 1–6.

[60] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben ten Hove, and Marcel Böhme. 2022. Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines. https://arxiv.org/abs/2205.14964

[61] Timothy Lethbridge, Susan Sim, and Janice Singer. 2005. Studying Software Engineers: Data Collection Techniques for Software Field Studies. *Empirical Software Engineering* 10 (07 2005), 311–341.

[62] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. MAFL: Non-Intrusive Feedback-Driven Fuzzing for Microcontroller Firmware. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1–12.

[63] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[64] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. 562–566.

[65] Hector D. Menendez and David Clark. 2022. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3540–3553.

[66] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-Time Temporal Logic Guided Greybox Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1343–1355.

[67] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*.

[68] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Kamei Yasutaka. 2023. Replication package for the paper: The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities. https://github.com/posl/fuzzingChallengesReplicationPackage

[69] obgm/libcoap. 2018. Issue #213. https://github.com/obgm/libcoap/issues/213. Last accessed on 07-12-2022.

[70] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3 (2019), 29 pages.

[71] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP'18)*. 697–710.

[72] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. 861–875.

[73] secdev/scapy. 2018. Issue #1398. https://github.com/secdev/scapy/issues/1398. Last accessed on 07-12-2022.

[74] secdev/scapy. 2021. Issue #3347. https://github.com/secdev/scapy/issues/3347. Last accessed on 07-12-2022.

[75] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 347–362.

[76] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[77] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 995–1007.

[78] Qian Yan, Minhuan Huang, and Huayang Cao. 2022. A Survey of Human-machine Collaboration in Fuzzing. In *Proceedings of the 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC'22)*. 375–382.

[79] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. 2022. Fuzzing of Embedded Systems: A Survey. 55, 7 (2022).

[80] Michael Zalewski. [n. d.]. American fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/

[81] Pengcheng Zhang, Bin Ren, Hai Dong, and Qiyin Dai. 2022. CAGFuzz: Coverage-Guided Adversarial Generative Fuzzing Testing for Image-Based Deep Learning Systems. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4630–4646.

[82] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. 2019. VisFuzz: Understanding and Intervening Fuzzing with Interactive Visualization. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. 1078–1081.

[83] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. 2169–2182.