

A Study of Software Quality Maintenance in Open-Source Software Projects

ヌリ, オリビエ

<https://hdl.handle.net/2324/7329505>

出版情報 : Kyushu University, 2024, 博士 (工学), 課程博士
バージョン :
権利関係 :



A Study of Software Quality Maintenance in Open-Source Software Projects

Olivier Nourry

June 2024

Department of Information Science and Technology

Abstract

Software quality is an important indicator to ensure that a software system is secure and can be maintained over time by developers. Poor software quality can cause a system to become bloated, too complex to use, or unsafe (i.e., it contains vulnerabilities). For open-source projects, specifically, it is important to conduct software quality maintenance so that new developers can contribute to the project and dependent projects in the open-source ecosystem do not get negatively impacted.

One way to ensure that software quality maintenance is conducted is by adopting proper software engineering practices. This includes (but is not limited to) using the right development tools, adopting an efficient release process, integrating software maintenance as part of the standard development process, and much more.

In this thesis, we study how software engineering practices can impact software quality maintenance activities by investigating real world practices in OSS (Open-Source Software) projects. More specifically, we aim to investigate if and how do coarse-grained/high-level and fine-grained/low-level software engineering practices affect software maintenance activities. In this context we define fine-grained software engineering practices as matters that affect daily software development activities. This includes what development tools are used daily by developers, or how often developers conduct maintenance activities such as software testing, bug fixing, or performance optimization. Conversely, we consider coarse-grained software engineering practices as matters that are more generic or big picture and which affect the overall development process rather than daily activities. For example, planning the content of each release, or gathering software requirements from stakeholders are both aspects of software engineering that are only conducted when they are required.

The first (coarse-grained) software engineering practice we investigated was

the release cycle. With modern software requiring constant changes, release cycles have progressively gotten shorter over time. Since core features are usually the primary focus during software development, our goal was to find out if these shorter releases came at the cost of software quality maintenance. For this study, we therefore selected three large OSS projects that have had experience with shortening their release cycles and investigated if their refactoring activities were impacted by the shorter releases. Since refactoring is one of the most common ways to maintain code quality over time, we used refactoring as a proxy metric to measure if software quality maintenance had been affected by the shorter releases. From this investigation, we were able to find out how the change in release cycle affected refactoring activities in software projects. We were also able to provide some recommendations for developers on how to keep up with software maintenance while following shorter release cycles.

We continued our investigation of software engineering practices by looking at a fine-grained aspect of software quality maintenance: software testing. In recent years, the OSS ecosystem has been heavily impacted by vulnerabilities spreading throughout the ecosystem and causing billions of dollars in damage. As a result, a vulnerability detection technique called fuzz testing (or fuzzing) has seen a surge in open-source software. In 2022, the Go programming language released native support for fuzz testing. Due to the importance of finding vulnerabilities in open-source software, we investigated if adding fuzz testing into the standard Go library motivated developers to fuzz their software systems. We data mined 11,341 open-source Go projects on GitHub and measured how much fuzz testing is being conducted in OSS projects. We then compared fuzz testing activities with other testing activities to see if fuzz testing was adopted as a standard testing practice. From this investigation, we were able to establish a list of recommendations for developers such as how fuzzing is best used and what functions of their software are critical to fuzz.

From there, we decided to investigate deeper and conduct another study on fuzz testing specifically. Since fuzz testing is much more complex to set up and

conduct than other testing strategies such as unit testing, we aimed to find out if the barrier of entry to fuzz testing was too high. We therefore investigated OSS projects to determine what current challenges are preventing open-source developers from conducting fuzz testing. We conducted a manual analysis of fuzzing related GitHub issues in OSS projects and established a clear taxonomy of challenges related to fuzz testing. We then surveyed fuzzing experts that conduct fuzz testing in OSS projects to get their opinion on how the adoption fuzzing can be improved. From this investigation, we delivered a clear overview of the current state of fuzz testing and a list of fuzzing related challenges that need to be addressed to lower the barrier of entry to fuzz testing. As an extension of our work, we selected the most important challenge limiting the adoption of fuzzing (build related issues) which we believe could be automated with proper software engineering tools. From this experiment we show that 1) it is possible to develop automation tools to lower the barrier of entry of fuzzing, and 2) developing software engineering tools can greatly abstract the complexity of software quality maintenance tasks such as fuzz testing.

The main contributions of this thesis are as follow. We give an overall overview of current software maintenance practices in OSS projects. We then show how both high-level and low-level modern software engineering practices can affect software maintenance. We then give a detailed taxonomy of challenges to show how software maintenance practices can be improved. Finally, we provide a proof of concept that shows it is possible to use automation to keep up with software maintenance during modern software development.

Acknowledgement

Related Publications

This thesis encompasses the findings in the following publications:

- Olivier Nourry, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Does shortening the release cycle affect refactoring activities: A case study of the JDT Core, Platform SWT, and UI projects, *Information and Software Technology* 2021.
- Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza and Yasutaka Kamei, AIP: Scalable and Reproducible Execution Traces in Energy Studies on Mobile Devices, *2022 IEEE International Conference on Software Maintenance and Evolution*.
- Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei, The Human Side of Fuzzing: Challenges Faced by Developers during Fuzzing Activities, *ACM Transactions on Software Engineering and Methodology* 2023.

The extended work section contains the results of the following study which is currently under major revision in *ACM Transactions on Software Engineering and Methodology* 2024:

- Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, Honglin Shu, Yasutaka Kamei, My fuzzers won't build: An empirical study of fuzzing build failures.

Contents

Acknowledgement	iv
Related Publications	i
1 Introduction	1
1 Motivation and Background	1
2 Thesis Overview	2
2 Does Shortening the Release Cycle Affect Refactoring Activities: A Case Study of the JDT Core, Platform SWT, and UI projects	6
1 Introduction	6
2 Background	8
2.1 Refactoring	8
2.2 Short Releases	10
3 Study Design	11
3.1 Research Questions	11
3.2 Project Selection	12
3.3 Data Extraction	13
3.4 Filtering	14
4 Case Study Results	16
4.1 (RQ1) Do Developers Still Refactor as Much?	16
4.2 (RQ2) Are Refactoring Efforts Affected by Shorter Releases?	18
4.3 (RQ3) Do Shorter Releases Impact What Type of Refactoring is Applied to the Code?	21

4.4	(RQ4) Does Shortening the Releases Affect the Human Resources Dedicated to Refactoring Activities and their Refactoring Workload?	29
5	Discussion	32
5.1	Evolution of Refactoring Activities	32
5.2	Refactoring and Software Quality	33
5.3	Summary and Implications	34
5.4	Threats to Validity	37
6	Chapter Summary	38
3	Exploring the Adoption of Fuzz Testing in Open-Source Software: A Case Study of the Go Community	40
1	Introduction	40
2	Background and Related Work	42
2.1	Background	42
2.2	Related work	44
3	Studied Data	46
3.1	Selection of Target Projects	46
3.2	Locating Code to be Analyzed	48
4	Preliminary Analysis	49
4.1	Do Developers Use Fuzz Testing? (PQ1)	49
4.2	Is Fuzzing Test Code More Prone to Change Than Non-Fuzzing Test Code? (PQ2)	51
5	Is Fuzz Testing Development Conducted As Part of the Standard Testing Process? (RQ1)	54
6	Who Conducts Fuzz Testing in Open-Source Projects? (RQ2)	57
7	Discussion	61
7.1	What Projects Are Making The Most Use of Fuzz Testing?	61
7.2	What Are the Main Uses of Fuzzing?	63
7.3	Summary and Takeaway	64
8	Threats To Validity	65
9	Chapter Summary	66

4	The Human Side of Fuzzing: Challenges Faced by Developers during Fuzzing Activities	67
1	Introduction	67
2	Study Design	69
2.1	Manual Analysis on GitHub Issues	70
2.2	Survey with Developers	74
3	Results	75
3.1	Challenges from Fuzzing Itself	76
3.2	Challenges from Fuzzing Service Providers	86
3.3	Survey Result Analysis	93
4	Threats to Validity	98
5	Related Work	99
5.1	Studies on Fuzzing Techniques	99
5.2	Studies on Fuzzing Challenges	101
6	Chapter Summary	103
5	Automating Software Quality Maintenance for Better Efficiency and Reduced Complexity	105
1	Introduction	105
2	Background	108
2.1	Fuzzing Process	108
2.2	OSS-Fuzz Infrastructure	110
3	Quantitative Analysis	114
3.1	Quantitative dataset	114
3.2	(RQ1) How often do fuzzing builds fail?	115
3.3	(RQ2) How long does it take to fix failing fuzzing builds?	118
4	Qualitative Analysis	121
4.1	Qualitative dataset	121
4.2	Qualitative analysis methodology	122
4.3	(RQ3) What are the root causes of fuzzing build failures?	124
4.4	Comparison with previous studies	130

4.5	Implications	131
5	Discussion	132
5.1	Why do projects abandon their fuzzing build and how do we prevent it from happening?	132
5.2	Improving fuzzing using other fields' breakthroughs.	134
5.3	Modeling build failures	135
6	Threats to validity	139
6.1	Threats to internal validity	139
6.2	Threats to construct validity	140
6.3	Threats to external validity	140
7	Related Work	140
7.1	Studies on build failures	140
7.2	Empirical studies on fuzzing	142
8	Chapter Summary	143
6	Conclusion	145
	Rereferences	146

List of Figures

2.1	An overview of the data extraction procedure	11
2.2	Range of our dataset	14
2.3	Number of refactoring commits per pattern during both release periods	23
2.4	Evolution of refactoring activities over time	32
3.1	Overview of the data collection process	47
3.2	Number of non-fuzz tests (left) and fuzz tests (right) per project containing at least one fuzz test (log scaled)	50
3.3	Number of commits modifying non-fuzzing test functions and number of com- mits modifying fuzzing test functions per month since fuzz tests were added to the standard library.	53
3.4	Distribution of confidence values across studied projects.	57
3.5	Percentage of testing commits that modify fuzzing code for developers con- ducting fuzzing activities.	60
4.1	Web App used for inspecting issues	72
4.2	Taxonomy of challenges from the fuzzing process itself.	76
4.3	Taxonomy of challenges from fuzzing service providers.	87
4.4	Frequency at which the respondents encounter each challenge in percentage. .	94
5.1	OSS-Fuzz architecture	111
5.2	OSS-Fuzz public panel to OSS-Fuzz projects' fuzzing builds status	113
5.3	Approach to calculate the time to fix a build failure in a failing build sequence.	118
5.4	Time to fix a fuzzing build failure in hours	120

5.5	Number of subsequent failing builds	120
5.6	Embedded log distribution using t-SNE	136

List of Tables

2.1	Summary of the dataset	15
2.2	Weekly refactoring activities of each project during TR and SR	18
2.3	Refactoring Churn in LoC during TR and SR	20
2.4	Top 10 most common refactoring patterns in the JDT.CORE project during each release period	24
2.5	Top 10 most common refactoring patterns in the PLATFORM.SWT project during each release period	25
2.6	Top 10 most common refactoring patterns in the PLATFORM.UI project dur- ing each release period	25
2.7	Ranking and workload variation between TR and SR	28
2.8	Developer resources during TR and SR	30
2.9	Refactoring workload contribution of developers during TR and SR	31
2.10	p -values for the number of refactoring commits during early versus late TR and early versus late SR	33
2.11	Impact of shorter releases on software development tasks	36
3.1	Association rule calculations	54
3.2	Median association rule values calculated across our dataset	56
3.3	Median number of developers participating to standard testing activites, fuzzing activities, and both types of testing activities across all projects.	59
3.4	Functions and functionalities that are being fuzzed in open-source GO projects.	61
3.5	Top ten projects with the highest number of fuzz functions in our dataset. . .	63

4.1	Years of development and fuzzing experience of the survey respondents.	75
5.1	Number of logs remaining after each step of the filtering process	115
5.2	Statistical data of how many builds fail in each project (in absolute number of builds) and across all projects (in percentage).	116
5.3	(Left) Cumulative percentage of initial build failures fixed within the specified time interval. (Right) Percentage of initial build failures with their corresponding subsequent number of build failures.	120
5.4	Root causes of fuzzing build failures observed during the manual analysis of failing fuzzing build logs. The number in parentheses () shows how many instances the annotators found during the manual labeling process.	125
5.5	Random Forest Classifier 10-Fold cross validation results	137

Chapter 1

Introduction

1 Motivation and Background

Software quality maintenance is the process of modifying, or updating part of a software system with the goal of maintaining or improving software quality over time. Software quality maintenance activities include, but are not limited to, software testing, code refactoring, and bug fixing. With modern software development evolving with the aim of providing as many new features as possible to users as quickly as possible, keeping up with software quality maintenance is an increasing challenge for developers.

Companies now release new versions and features faster than ever before to customers. For example, in a 2017 engineering blog,¹ a facebook engineer described Facebook’s continuous integration and continuous delivery processes and revealed that the Android code alone is built between 50,000 and 60,000 every single day to keep up with new changes.

The unprecident size and complexity of modern software systems also poses new challenges to developers conducting software quality maintenance. For instance, Google’s Chrome project had less than 2 million lines of code when released in 2008 versus more than 37 million lines of code as of 2024.² While this exponential growth brings with it many new features for the users, the challenge of maintaining such large systems has also become increasingly more complex and time consuming for developers.

¹<https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/>

²https://openhub.net/p/chrome/analyses/latest/languages_summary, Last Accessed: Jun 2024

For such large systems, a lack of proper maintenance can cause them to become bloated, too complex to use, or unsafe (i.e., it contains vulnerabilities). For open-source projects, specifically, conducting proper software quality maintenance is essential to reduce the chance of having vulnerabilities that can spread throughout the open-source ecosystem in the codebase. As shown by the Log4j incident in 2021 [8], a single vulnerability in a large open-source project can spread worldwide and cause billions of dollars in damage.

With the new constraints brought by modern software development (e.g., shortened release cycles, growth in size and complexity, etc.) it is therefore relevant to 1) investigate if developers are able to keep up with software quality maintenance during the development process and 2) provide developers better software quality maintenance tools and strategies to keep up with modern software development. This thesis uses highlights several challenges of conducting software quality maintenance activities in open-source projects, assesses a solution proposed by the open-source community to facilitate software testing activities, and we propose a proof of concept to partially automate software quality maintenance activities.

2 Thesis Overview

In this section, we provide an overview of the thesis and its main contributions to the field of software engineering.

- We demonstrate how software engineering and development practices can directly impact software quality maintenance and development activities.
- We conduct a case study of the Go community to investigate if developers make use of software quality maintenance features that are added to the standard library.
- We conduct the first study investigating fuzzing practices to find areas that could benefit from better software engineering. Our work establishes foundational work in the field of fuzzing to start conducting empirical studies and improve fuzzing practices. Our taxonomy of challenges presents the most accurate and up-to-date overview of the current factors that makes fuzz testing difficult to conduct. Our work provides the research community the necessary insights into the field of fuzzing to help lower the barrier of entry to fuzzing and allow developers to conduct software quality maintenance

for vulnerability detection.

- We conduct an experiment to explore the potential of using use machine learning to automate software quality maintenance activities. We provide a proof of concept and show that it is possible to develop automation tools that reduce the complexity of software quality maintenance activities.

Challenge I:

Investigate how modern software development strategies impacts software quality maintenance activities

In Chapter 2, we first investigate how software engineering practices can impact software quality and activities aimed at maintaining or improving software quality. For our case study, we specifically investigate how changing the software release cycle can impact refactoring activities. We select three projects from the Eclipse foundation that reduced their software release cycle from one release every year to one release every four months. We then use the RefactoringMiner tool to find every refactoring applied in the three studied projects. We finally investigate how the shorter release strategy impacted refactoring activities by comparing the amount of refactoring applied, the efforts dedicated to refactoring activities, and the type of refactorings applied during both the yearly release period and the quarterly release period.

Our results show that shortening the release cycles caused developers to increasingly apply simple refactoring patterns over complex refactorings. Specifically, patterns that have full tool support such as “Renaming” types of refactorings (e.g., rename a function and all instances where it’s called) greatly increased in popularity/usage after shortening the release cycles. From our three studied projects, we find that only one was able to sustain its refactoring activities even with shorter releases. Our investigation reveal that it’s also the only project where the number of developers conducting refactoring activities increased (instead of decreased) after adopting quarterly releases.

Challenge II:

Explore the effectiveness of adding software quality maintenance features to the standard library on software quality maintenance activities.

In Chapter 3, we perform a large empirical study to understand the impact of adding software quality maintenance features to a language’s standard library. For our case study, we use the recent (2022) addition of fuzz testing into the official Go standard library to investigate if the use of fuzz testing increased as a result of this feature addition. We therefore investigate fuzz testing activities in 11,341 open-source Go projects on GitHub.

Our results reveal that only a small minority of open-source projects make use of native Go fuzzing. Moreover, we find that developers mostly rely on already established testing practices such as unit testing. By analyzing commit histories of projects, we find that very few developers are contributing to fuzzing efforts which suggests that fuzz testing might still be too complex to conduct for average developers.

By performing a qualitative analysis, we then find that the projects using fuzzing the most are projects for which security is of most importance (e.g., network-related, financial-related, etc.). Finally, we find that these projects use fuzz testing sparingly to test the most critical/important functions of their codebase.

Using our results, we propose a set of recommendations on how to best use fuzzing for developers who wish to use fuzz testing during their software quality maintenance activities. Finally, this study shows that adding software quality maintenance features to the standard library is not sufficient by itself to promote the use of new software quality maintenance features and that more work needs to be done to help developers conduct their software quality maintenance.

Challenge III:

Investigate what are the current barriers preventing developers from conducting software quality maintenance for their software systems.

In Chapter 4, we perform a large manual analysis to investigate the current factors limiting the adoption of fuzzing in 829 GitHub issues. We further validate our results by conducting

a survey with over 100 fuzzing experts to get their personal experience on what makes fuzz testing challenging to conduct and what areas of the fuzzing process requires improvements. As of 2024, fuzzing is the current state of the art approach to detect software vulnerabilities in the source code (i.e., detecting poor software quality). This work aims to highlight key areas of fuzzing that can be improved to enable non-experts to start fuzzing their source code and consequently improve their software quality.

Our investigation reveal 39 challenges related to conducting fuzzing activities. Our survey with developers reveal that fuzzing suffers from a high barrier of entry making it difficult for non-experts to get involved with fuzzing activities altogether. This work lays down the foundational work for future software engineering research to address fuzzing's high barrier of entry via the development of better tools or the implementation of standardized fuzzing practices.

Chapter 2

Does Shortening the Release Cycle Affect Refactoring Activities: A Case Study of the JDT Core, Platform SWT, and UI projects

1 Introduction

Refactoring in software systems has been investigated extensively throughout the years. Multiple aspects such as the impacts on code quality [120] and testing activities [144] have been investigated in order to determine how important it is to refactor the code in a software development project. Past studies [122,172,192] have concluded that refactoring is beneficial to a software development project and that it is worth dedicating some development time to refactor the code.

Modern-day software companies, however, have started changing the way they develop software systems. Over the past few years, large companies such as Google, Facebook, Firefox, and Netflix have all decided to adopt a short release cycle model for various reasons such as keeping up with customer expectations or keeping up with competitors [187].

This short release cycle model has been referred to as rapid releases (RR), or short releases (SR), and the old yearly release model as traditional releases (TR). Several studies have been

conducted in order to better understand the impacts of shorter releases by comparing various metrics and activities before and after shortening the release cycles. For example, a previous study [144] showed that shorter releases' builds contain more code commits than traditional releases' builds. Another study [120] also found that short release cycles had a significant impact on the number of files being affected per commit. The code quality, number of bugs, integration, and testing in shorter releases have also all been subject to investigation in past studies [38, 120, 144] but to the best of our knowledge, the impacts of shortening the release cycles on refactoring activities have not yet been investigated. In this chapter, we therefore conduct a study to further contribute to the empirical knowledge related to the impacts of shortening the release cycles on software development activities.

In this case study, we investigate if shortening the release cycles had any impacts on refactoring activities in three software development projects. Due to the possible negative impacts of not refactoring, it is important for potential short release adopters to be aware of the possible downsides related to refactoring activities that shorter releases can bring to a project. We examine four aspects of refactoring activities: the amount of refactoring done, the efforts devoted to refactoring, the type of refactoring applied in the code, and the human resources dedicated to refactoring. These aspects allow us to draw an overall picture of refactoring during short releases and traditional releases in order to determine if the refactoring activities are impacted after adopting a shorter release cycle.

Study Organization: Section 2 introduces previous studies involving refactoring and short releases. Section 3 describes our dataset selection, its extraction and the filtering process, and Section 4 presents our results. In Section 5, we summarize our findings, discuss their implications and address some of the threats to the validity of this study. We conclude in Section 6 where we briefly summarize our findings and make our final comments on refactoring and short releases.

2 Background

2.1 Refactoring

Refactoring has been widely defined as changes that improve code quality but preserve the software system's external behavior [57]. Thanks to the improvement of refactoring detection algorithms [44, 123, 189, 195, 209] that automatically identify refactoring activities from source code histories, many researchers have conducted empirical studies [14, 124, 162, 172, 174, 188, 199] on software evolution and software refactoring.

Refactoring and Software Quality. In a previous study [128], Lacerda et al. performed a systematic literature review and concluded that refactoring has a strong relationship with code quality attributes such as understandability, reusability, and maintainability. During their literature review, they also identified that code quality is one of the relationship linking code smells and refactoring. Code quality is therefore a good representation of how well refactoring can remove code smells.

While leading a case study [159] on the impacts of refactoring on software quality and productivity in an agile environment, Moser et al. also found empirical evidence that refactoring improves code quality factors, reduces code complexity and coupling, and improves cohesion. Their results also showed empirical evidence that refactoring increases development productivity. In another study [160], Moser et al. also proposed a new methodology to assess if refactoring improves quality. Using their new proposed approach, they conducted a case study in a near industrial environment and found that refactoring has a positive effect on reusability and promotes ad-hoc reuse of object-oriented classes.

Kim et al. [122] found that there is an increase in the number of bug fixes after API-level refactorings while investigating the role of refactorings in three evolving open source projects. They also found that less time is required to fix bugs after API-level refactorings are applied to the code.

While refactoring has many benefits, refactoring code changes carry the same risks associated with any other codebase change such as the introduction of new defects as shown in a previous study [45] that found that refactoring can account for a significant part of API breaking changes.

Refactoring automation. To mitigate the risks of introducing new defects, it has been suggested that refactoring tools should be used whenever possible. Multiple automated tools have been developed over time to ease the process of refactoring for developers. These tools were designed to help developers in several ways ranging from automated refactoring suggestion [103], to automated refactoring validation that checks if a manual refactoring was applied correctly [61] and all the way to fully automated refactoring applications [205]. Popular IDEs such as IntelliJ IDEA and Eclipse have also added refactoring support by adding their own built-in refactoring tools to their software. These tools generally cover a wide range of refactoring patterns and can help developers save time when refactoring their code.

Xing and Stroulia [210] studied the proportion of code modifications for refactoring changes and which types are the most frequent types of refactorings being applied in the Eclipse JDT project. They found that about 70% of structural changes may be due to refactorings code changes and that 60% of refactorings can be automatically applied by a refactoring tool if the refactoring tool can gather the information relevant to the refactoring being applied.

Although refactoring tools are becoming more commonly used, Silva et al. [188] found that many developers still do not trust automated refactoring for complex cases or believe that automation does not need to be used in trivial cases. A previous study led by Brett et al. [40] also found that refactoring tools themselves can contain multiple bugs. It is therefore very important that refactoring is done properly in order to avoid introducing more defects. Because each release is usually longer when using a traditional release cycle model, the evolution of the software system is well controlled due to developers planning ahead of time how much time is to be allocated to each refactoring task. New release methodologies however reduce that planning time a lot by shortening release cycles [154]. This study differs from previous refactoring and software evolution studies by taking into account this new shorter releases approach to software development.

2.2 Short Releases

Many software development projects have been switching from a traditional release cycle to a shorter release cycle to deliver new features faster (a few weeks to a few months to ship a major release) rather than providing them yearly. This satisfies users and also allows developers to receive quick feedback from users [121]. For instance, the Firefox and Eclipse projects shortened the release intervals for major/main releases, respectively. Specifically, the Firefox project started releasing a product in 6 weeks rather than yearly. From Eclipse 4.9, the Eclipse project abandoned service releases (i.e., patch releases) and replaced all the service releases with main releases.

Many studies have examined the impacts of shortening the release cycles on development activities (e.g., testing, bug-fixing, and integrating) to find that shortening release intervals affects development activities [15, 35, 38, 118, 121, 127, 144, 155]. For bug-fixing activities, the average bug-fixing time is reduced by adopting shorter releases [15]. As for testing activities, shorter releases narrow the scope of the tests and increase the number of test executions [144].

Research motivation. In the field of the consumer market, Tonietto et al. [194] reported about code behaviour changes when developers are under time pressure. When time is limited, people tend to perform fewer tasks and are less likely to complete time-consuming tasks because of the perception of having less time than they actually have when compared to an unbounded time [194]. In this way, while having limited time, developers change the scope of their tasks and the amount of effort put on specific tasks in order to optimize their activities.

Refactoring, for instance, is not trivial work when it must be completed in a short amount of time. In fact, past studies have found that certain types of refactoring can be quite challenging even without time limitations. For example, Garrido et al. [60] have previously studied the challenges of refactoring a C program file that uses preprocessor directives. Vassallo et al. [200] also investigated the advantages and barriers of continuous refactoring in CI and found out that *the main barrier to continuous refactoring was the lack of time*. The same study also states that this is even more true in agile projects because there is more emphasis on new changes. We use this finding by Vassallo et al. to motivate our study of the impacts of shorter releases on refactoring practices.

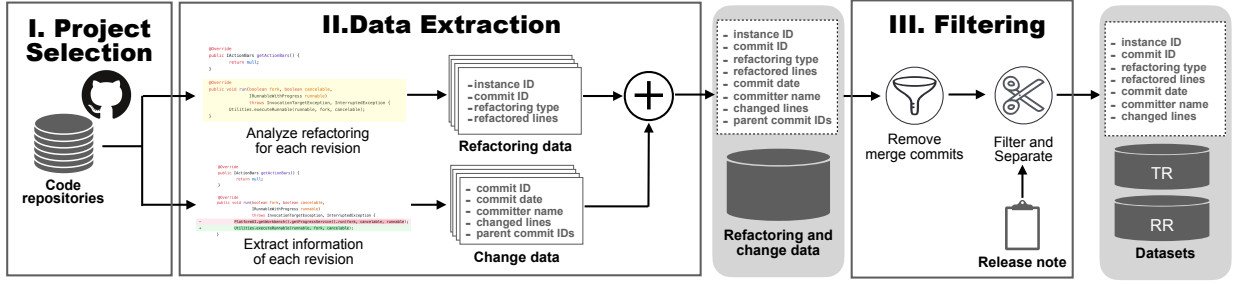


Figure 2.1: An overview of the data extraction procedure

To the best of our knowledge, our work is the first to examine the impact of switching from traditional yearly releases to shorter releases on refactoring activities. Since refactoring plays a crucial role in software development, this case study provides further insight for projects planning to transition to shorter releases. In this study, we focus on the process change and compare the amounts, efforts, contents, and human resources of refactorings between traditional releases and shorter releases in order to investigate if shortening the release cycles had any impacts on refactoring activities.

3 Study Design

This section explains the design of our case study in order to address our research questions introduced in Section 3.1. To measure the impacts of switching to a shorter release cycle on refactoring activities, we compare two datasets derived from traditional yearly releases and shorter quarterly releases. Figure 2.1 depicts an overview of the data extraction process. We first apply a refactoring analysis tool to each revision in code repositories. Next, we measure the relevant metrics and merge them with the outputs of the refactoring tool. Finally, the merged outputs are divided into two datasets based on whether the commit date is before or after the cycle change date.

3.1 Research Questions

In this study, we structure our investigation around the following four research questions.

RQ₁: Do developers still refactor as much? (Amount)

RQ₂: Are refactoring efforts affected by shorter releases? (Effort)

RQ₃: Do shorter releases impact what type of refactoring is applied to the code? (Contents)

RQ₄: Does shortening the release cycles affect the human resources dedicated to refactoring activities and their refactoring workload? (Human-resource)

In previous studies [38,120,144] that investigate the impacts of shorter releases on various development tasks, three main aspects have been used to measure the impacts: the frequency of the task (RQ1), the characteristics of the task (RQ3) and the human resources dedicated to the task (RQ4). In addition to investigating these three main aspects, in RQ2, we diverge from the main aspects used by previous studies in order to investigate the impacts of shorter releases on refactoring efforts by measuring the amplitude of the refactoring changes. In this study, we design our research questions in a way that will help paint a broad picture of what impacts, if any, switching to shorter releases has on refactoring activities. We also try to align our analysis with previous studies so that we can compare our results afterwards. We compare the results of our empirical study with previous studies in Table 2.11 in the discussion section to get an overall view of the possible impacts of shortening the releases on development activities.

3.2 Project Selection

To address our research questions, we perform a case study analyzing refactoring in traditional yearly releases and shorter quarterly releases. The projects used in this study were picked because they satisfied the following three criteria:

Criterion₁: **Experience in shortening release cycles.** We need to compare the amount, effort, contents and human-resource of refactoring between traditional and shorter releases. Therefore, our studied systems must have development experience following both a traditional release cycle and a shorter release cycle.

Criterion₂: **Java System.** Since we need to analyze refactoring activities, we use a state of the art refactoring analysis tool that can detect various refactoring types (See Section 3.3). However, this tool has the limitation of only being able to analyze Java code. Thus, our studied systems must be written in Java.

Criterion₃: **Large-size repositories** In order to acquire reliable results, we need many instances of refactoring. Our studied systems must have at least 1,000 commits during the traditional releases period and a significant number of commits during the shorter releases period in their master branch.

Because short release cycles are not a widely used approach yet in open-source software development, the number of open source projects that have developed for a significant period of time following both yearly traditional releases and shorter releases is limited. Considering this limiting factor and our three selection criteria, we decided to pick three Java projects from the Eclipse Foundation which are known to be participating in the simultaneous release initiative. These projects are the JDT.CORE, PLATFORM.UI, and PLATFORM.SWT projects, which are mature (over 10 years of development) and still active.

The Eclipse Foundation, which owns these projects, follows a simultaneous release scheme to deliver products in sync with all the projects in Eclipse. Figure 2.2 depicts the transition of Eclipse’s release cycle. So far, the release cycle has undergone changes twice before the adoption of their current release model. Until Eclipse version 4.5, Eclipse delivered yearly new features and bug-fix in the main release in June and then provided patch releases twice in September and February in order to fix bugs found in the main release. In Eclipse 4.6, they added a new patch release (named service release) in December and moved the February patch release to March making all releases released quarterly. Furthermore, from Eclipse 4.8, a shorter release cycle was adopted which abolished patch releases and made every quarterly release into a main release.

3.3 Data Extraction

In order to mine the refactoring patterns, we used the state-of-the-art refactoring analysis tool developed by Tsantalis et al. called RefactoringMiner [195]. RefactoringMiner can detect up to 40 different types of refactoring patterns (as of February 1, 2020)¹. We used this tool due to its state of the art performance (high precision and high recall [195]) as well as the large diversity of patterns that the tool can detect. The tool can detect simple patterns such as

¹We used build “8ccec9050ad6dcc4d575cce3713f5283d5988c5d” since RefactoringMiner is still being developed.

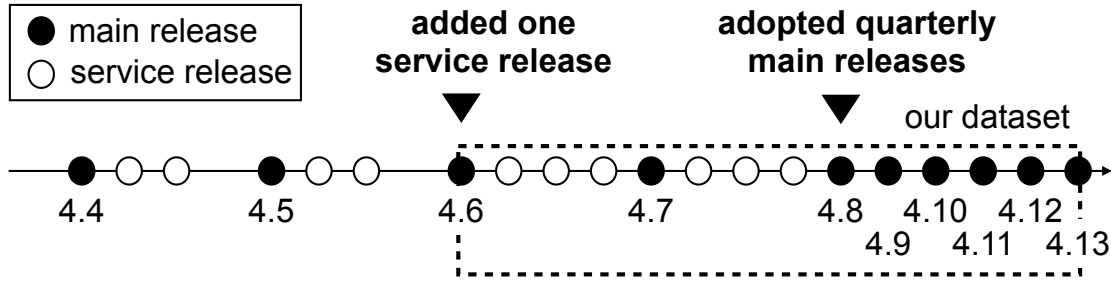


Figure 2.2: Range of our dataset

renaming and moving pieces of code or pushing down and pulling up methods. It is also able to detect more time consuming or complex patterns such as extracting a superclass or merge and split types of refactorings such as split parameter or merge attribute. After cloning the repositories of the studied projects, we ran the RefactoringMiner tool on every commit. The output of each commit contains the information of multiple refactoring instances, including the refactoring types and the refactored lines in the source code.

Additionally, we extracted the basic information of each commit such as commit date, parent commit ids, changed lines, and committer name. We used the commit date and the parent commit ids for our filtering process and to create our datasets. Then, we utilized the changed lines and the committer name for RQ2 and RQ4, respectively. Finally, we merged the basic information with the output of the RefactoringMiner, based on the commit id.

3.4 Filtering

Merge commits. Before making datasets, we need to remove commits with duplicated refactoring instances. Code repositories contain merge commits, which will cause refactoring instances to be redundantly detected even though the same snippet has previously been marked as refactored in the parent commits. Therefore, we eliminate commits with multiple parent commits from our dataset.

Commits during early development periods. Since the goal of this study is to find the impacts of shortening the release cycle, in order to have a fair comparison between the traditional releases data and short releases data, we selected a subset of the traditional releases data that was as close as possible to the switch to shorter releases. Using this subset, we can ensure that we are not comparing early development commits and contributions that are

Table 2.1: Summary of the dataset

Project	Release	Commits	Refactoring Instances	Refactoring Commits	% Refactoring Commits
jdt.core	TR	1,306	3,435	354	27.10%
	SR	618	816	146	23.60%
platform.swt	TR	1,877	3,466	168	9.00%
	SR	1,243	1,409	115	9.30%
platform.ui	TR	1,475	1,914	270	18.30%
	SR	1,340	3,182	224	16.70%

aimed at developing the core of the application with recent contributions aimed at software maintenance and update [120]. Thus, we do not use the data prior to Eclipse 4.6 (June 22nd, 2016). We would like to note that the yearly release cycle (main release) was broken on Eclipse 4.8 (June 27th, 2018) where quarterly releases were adopted (See Figure 2.2). All commits authored prior to June 27th, 2018 (Eclipse 4.8) were categorized as traditional release commits and all commits done on June 27th, 2018, and after were categorized as short release commits. Although the release periods have a different length, our data is normalized so that our results are not affected by it. In RQ1 and RQ2, we compare weekly median data to normalize between the two release periods, in RQ3 we normalize using a ranking-based approach, and finally in RQ4 we use percentages and medians to normalize our data.

The main idea behind this choice is that the project would have the same maturity and frequency of code contribution from the contributors if no release cycle change happened. Taking a subset that was too early in the project's life would result in comparing traditional releases commits aimed at developing the core of the application against shorter releases commits which are more oriented towards maintenance and feature additions since the project is much older at that point.

Dataset Summary: The final dataset, as shown in Table 2.1 is made of traditional releases data and short releases data extending over a similar length of time. Henceforth, traditional releases refer to the time period between July 22nd, 2016, and June 27th, 2018, and short releases refer to the time period after June 27th, 2018.

4 Case Study Results

4.1 (RQ1) Do Developers Still Refactor as Much?

Motivation. Due to the shorter releases, developers do not have as much time to dedicate to tasks not related to feature development and maintenance such as testing [47, 48, 113, 202] or refactoring. For example, Vassallo et al. [200] found that lack of time is the main barrier to refactoring while investigating continuous refactoring in CI. We therefore investigate if adopting shorter release cycles affects the overall project’s refactoring activity by comparing how much refactoring was being done during both release periods.

Approach. Refactoring is not usually done as an everyday task but rather when it is needed to implement new features or when the code becomes unreadable [124]. For these reasons, we decided to aggregate our data to study refactoring activity on a weekly basis.

For RQ1, we focus our results around the median weekly values to compare the refactoring done during the traditional releases period and the shorter releases period. To extract the overall refactoring activity in our chosen projects, we use the following two metrics:

Refactoring Instances: We use refactoring instances as our main metric to see exactly how much refactoring was applied to the code during each period. The number of instances directly reflects how many times the code was changed for refactoring purposes.

Refactoring Commits: We use refactoring commits as a secondary metric to study refactoring activities. The number of refactoring commits gives us an insight as to how often the code is refactored.

We start our investigation by counting the number of refactoring instances per week in each project and extracting the weekly median number of refactoring instances. We then compare the median values between traditional releases and short releases to see if shorter releases impacted how much refactoring is applied to the code. We then proceed to look at refactoring commits to see if there is more weekly refactoring activities after shortening the releases. We extract the weekly median number of commits during the traditional releases period and the shorter releases period and then compare the weekly median percentage of refactoring commits during both periods.

We finally use a Mann-Whitney test for statistical tests and Cliff's delta for effect size, which are non-parametric, to verify if there is a statistically significant difference in the number of refactoring instances and refactoring commits between the traditional releases and the shorter releases. We select these non-parametric tests because refactoring activities sometimes show outliers and are not assured to follow a normal distribution.

Findings 1. We observe a decreased amount of refactoring with a lower number of refactoring instances applied in the JDT.CORE project after shortening the release cycle

As shown in Table 2.2, we find that the weekly median number of refactoring instances is very similar during both the traditional releases and the shortened releases for the two platform projects. The JDT.CORE project's median refactoring instances per week, however, is twice as much during the traditional releases period than the short releases period. Although both the platform projects show a small increase in the median number of weekly refactoring instances, a Mann-Whitney test on the weekly number of instances for each project confirms that out of the three projects, the JDT.CORE project is the only one showing a statistically significant difference in the number of refactoring instances per week ($p\text{-value} < 0.01$). It is also the only project with a non-negligible effect size for the number of refactoring instances per week. The two platform projects were found to have no statistically significant difference in the weekly number of instances.

Findings 2. We find a decrease in refactoring activities with a lower number of weekly refactoring commits in the JDT.CORE project after adopting shorter releases.

The refactoring commits data shows less of a difference than the refactoring instances data when looking at the weekly median number of refactoring commits. Out of the three projects, we find that only the JDT.CORE project has a difference in the weekly median number of refactoring commits. We however find that this measure does not tell the whole story; when looking at the percentage of weekly commits that contain refactoring changes in Table 2.2, we find that there was an overall decrease in median percentage of refactoring commits of almost 6% in the JDT.CORE project after adopting shorter releases. Although the decrease is much smaller than the one found in the JDT.CORE project, both platform

Table 2.2: Weekly refactoring activities of each project during TR and SR

Project	Release	Refactoring instances			Refactoring commits			
		Median #	SS	ES	Median #	Median %	SS	ES
jdt.core	TR	10.5	0.002	d=0.28	3.0	25.00%	0.002	d=0.28
	SR	5.0	p<0.01		2.0	19.09%	p<0.01	
platform.swt	TR	2.0	0.523	d=-0.06	1.0	7.14%	0.867	d=-0.01
	SR	2.5	p>0.01		1.0	7.02%	p>0.01	
platform.ui	TR	5.0	0.4882	d=-0.06	2.0	16.67%	0.469	d=-0.06
	SR	6.0	p>0.01		2.0	14.29%	p>0.01	

*SS: statistical significance ES: effect size

projects also show a decrease in the median percentage of refactoring commits.

Using a Mann-Whitney test and Cliff's delta, we find that our results remain consistent with our previous observation; only the JDT.CORE project has a statistically significant difference in weekly refactoring commits and a non-negligible effect size. The PLATFORM.SWT and PLATFORM.UI projects do not show any statistically significant difference in the number of refactoring commits between the traditional releases and the shorter releases.

From our investigation, we conclude that adopting shorter releases may have affected the amount of refactoring done and the refactoring activities of the JDT.CORE project. We found no clear positive or negative impacts of shortening the release cycle in the platform projects.

4.2 (RQ2) Are Refactoring Efforts Affected by Shorter Releases?

Motivation. In some cases, we find commits with multiple refactoring instances affecting only a few lines of code. In other cases, we find single commits with only one instance that required changing hundreds of lines of code. Because of these different cases, we cannot determine how much effort is put into refactoring using only the number of refactoring commits and instances.

Since past studies have found that refactoring is more often done manually than using refactoring tools [124, 162, 188], we need to take into account that some refactorings require significantly more time and code changes while others can be done very quickly and easily. In

this RQ, we aim to find out how much refactoring efforts developers dedicate to refactoring by comparing the refactoring code churn of refactoring commits during both the traditional releases and the shorter releases.

Approach. Because code churn has been used several times as a proxy measure of effort or as part of a subset of multiple metrics used to measure effort in past studies [114, 115, 149], we decided to use the refactoring code churn as our metric to estimate how much effort is put by developers into refactoring during the yearly releases and the quarterly releases.

The first thing we did was to aggregate all files that had been refactored at least once. We then looked at the Git history to extract every code change and code churn from this subset of files. For every refactoring instance in our dataset, we used the output of the RefactoringMiner to find the specific source code lines involved with refactoring.

In some cases such as refactorings that move source code folders, we found that the RefactoringMiner associated some code changes with a specific refactoring instance when the changes were actually caused by a different refactoring instance. This meant that some lines of code involved with refactoring were counted twice when summing up code churn for all the refactoring instances. To avoid counting the lines of code multiple times for the same code changes, we compared the code change history with the lines flagged by the RefactoringMiner to make sure each line was only counted once.

To calculate the refactoring code churn for a specific file, we therefore compared the code change history of the file with the source code lines that were flagged by the RefactoringMiner as being involved with refactoring. We discarded all code changes that were not directly caused by refactoring in order to only keep the history of refactoring code changes for that file. We repeated this process for every file in our subset to get our final dataset for RQ2.

After extracting every file's individual refactoring code churn, we aggregated our data by project in order to measure the change in refactoring effort on a project basis. We then calculated the total refactoring churn involved during the traditional releases and the shortened releases for each project. Similarly to our approach in RQ1, we then split our data on a weekly basis. We use the median of weekly refactoring code churn to get an overview of the difference in refactoring efforts over time for each project. We then look at the weekly refactoring code churn distribution to see exactly how refactoring efforts changed between

Table 2.3: Refactoring Churn in LoC during TR and SR

Project	Release	Total Refactoring Churn	Weekly Refactoring Churn Median
jdt.core	TR	18,498	46.5
	SR	10,973	20.5
platform.swt	TR	35,131	11
	SR	4,993	5.5
platform.ui	TR	6,207	26
	SR	45,657	20.5

the yearly releases and the quarterly releases for each project.

Although multiple studies [124,162,188] find that most refactoring is being done manually, some of these studies also find that a small subset of patterns associated with “Renaming” types of refactoring are more often applied using tools than manually. Due to the high likelihood of tool usage, we excluded “Renaming” types of refactoring from RQ2’s analysis.

Findings 3. The median weekly refactoring churn of the JDT. CORE project shows a statistically significant difference between the traditional releases and the shorter releases. We proceed to look at the weekly data starting with the weekly median refactoring churn as shown in Table 2.3. Surprisingly, we find that the highest gap in median weekly refactoring, with more than double the weekly refactoring churn during traditional releases, is from the JDT.CORE project which had the lowest difference in total refactoring churn. The PLATFORM.SWT shows a small decrease in weekly refactoring churn median and the PLATFORM.UI project shows no impact at all.

A Mann-Whitney test on the weekly refactoring code churn confirms our findings by revealing no statistically significant difference in the two platform projects. The JDT.CORE project, unlike the platform projects, was found to have a statistically significant difference in weekly refactoring churn.

From our investigation, we conclude that adopting shorter releases may have affected the weekly amount of efforts dedicated to refactoring in the JDT.CORE project. While we also notice an overall downward trend of refactoring efforts during the shorter releases in the platform projects, the observed decrease in refactoring efforts was not as significant in the platform projects as it was in the JDT.CORE project.

4.3 (RQ3) Do Shorter Releases Impact What Type of Refactoring is Applied to the Code?

Motivation. Past studies have suggested that testing activities are affected by time constraints [47, 48, 113, 202]. Because developers must follow stricter time constraints during the quarterly releases, we investigate if the type of refactorings applied is also affected by the new time constraints. For example, it is possible that only small and simple refactorings such as renaming would be applied; leaving large refactorings that change the structure of the code on the side due to the lack of time. In this section, we look at the type of refactorings being done during the traditional yearly releases and the quarterly releases to see if the shorter releases allow developers to still take care of all types of refactorings after shortening the release cycle.

In RQ3, we conduct two analyses (Section 4.3 and 4.3) to evaluate which refactoring types are the most common. We first start with a fine-grain analysis between yearly releases and quarterly releases based on the refactoring patterns individually. We then conduct a coarse-grain analysis by categorizing the refactoring patterns into distinct categories and analyzing the variation of these categories between the traditional releases and the shorter releases.

Refactoring pattern analysis (Fine-granularity)

Approach. Previous work [196] found that the number of commits is an accurate measure to estimate workload. Therefore, to get a general idea of which patterns were the most often applied in each project, we first examined how many commits for each refactoring pattern there are in all three projects during both time periods. To better visualize the variation in popularity for each refactoring pattern, we ranked every pattern during both time periods

based on the number of commits that implemented them. This allows us to normalize the number of commits between the traditional releases and the shorter releases. Because we are not interested in the amount of refactoring done but rather the type of refactoring being done in RQ3, using the ranks to compare between both time periods allows us to abstract how much refactoring is being done and focus exclusively on which refactoring types are the most common during each release period. Lower ranks represent patterns that are frequently applied to the code and high ranks represent patterns that are not often applied to the code.

To see if the same patterns tend to have a high frequency of application during both time periods or if the new release model affected which refactoring patterns are being applied by the developers, we then selected the top 10 refactoring patterns with the most commits in traditional releases and the top 10 refactoring patterns with the most commits in the shorter releases respectively. To see which patterns were the most affected by the adoption of shorter releases, we also calculated the variation in ranking (ranking Δ) of each refactoring pattern from our top 10 list of patterns with the highest number of commits.

Pattern Ranking: Each refactoring pattern is assigned a rank for the yearly releases period and the quarterly releases period for each project based on how many commits implement them. Common refactoring patterns are assigned low ranking values and uncommon patterns are assigned high ranking values.

Ranking Δ : We calculate the ranking Δ for each refactoring pattern by calculating the variation in ranking between yearly releases and quarterly releases. A positive Δ indicates that a pattern's rank increased and therefore became less common during the shorter releases and a negative delta indicates that a refactoring pattern's rank decreased and therefore became more common during the shorter releases.

Findings 4. We find that most refactoring patterns that were common during traditional releases were also common after shortening the release cycles. To get a general view of which patterns were popular during both time periods, we first extracted how many commits implement each refactoring pattern for each project during the traditional releases and the shorter releases. From Figure 2.3, we notice that the distribution of refactoring patterns that were popular in each project is similar during both time periods.

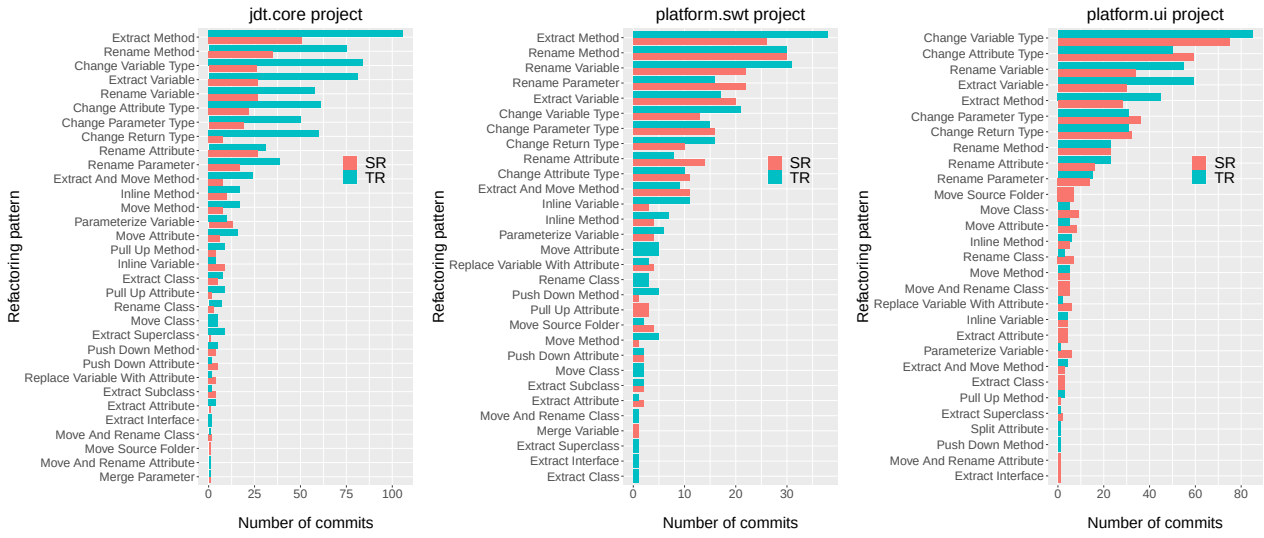


Figure 2.3: Number of refactoring commits per pattern during both release periods

Looking at the refactoring patterns with the most commits for each project, we also find that the three projects share a lot of similarities with patterns such as Extract Method being applied frequently during both the traditional releases and the shortened releases in all three projects.

From the ranking assignment process previously described, we calculate the rank of each refactoring pattern and find that the most common refactoring patterns remain very similar even after shortening the releases. As shown in Table 2.4, 9 of the top 10 refactoring patterns are common in both the traditional releases and the shorter releases in the JDT.CORE project. From Table 2.5, we find that 8 out of 10 patterns are common in both time periods in the PLATFORM.SWT project and from Table 2.6, we find that all 10 patterns with the highest ranks during the traditional releases remain the same during the shorter releases in the PLATFORM.UI project. From these results, we find that most patterns that were popular during traditional releases remained popular after adopting shorter releases.

Looking at the ranking Δ of each pattern in Table 2.4, we notice that the *Change* refactoring patterns have significantly gone down in popularity during the shorter releases in the JDT.CORE project. We also observe that the *Rename* refactoring patterns have been oppositely affected and became much more common during the shorter release cycle. From Table 2.5, we find similar results as in the JDT.CORE project with *Change* refactoring pat-

Table 2.4: Top 10 most common refactoring patterns in the JDT.CORE project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Change Return Type (M)	6	15	+9
Change Variable Type (L)	2	6	+4
Change Attribute Type (M)	5	7	+2
Extract Method (M)	1	1	+0
Change Parameter Type (M)	8	8	+0
Rename Parameter (H)	9	9	+0
Extract Variable (L)	3	3	+0
Rename Method (H)	4	2	-2
Rename Variable (L)	7	4	-3
Rename Attribute (H)	10	5	-5
Parameterize Variable (H)	15	10	-5

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

terns decreasing in popularity and *Rename* refactoring patterns increasing in popularity in the PLATFORM.SWT project. These results align with a previous study’s findings [21] where they observed that Eclipse puts a heavy emphasis on backward compatibility. It is therefore not surprising that developers would be reluctant to perform refactorings that can affect the API such as changing method signatures.

While these results seem to confirm our initial assumption that simpler refactorings such as renaming are increasing in popularity due to the shorter releases, from Table 2.6 we find that the patterns in the PLATFORM.UI project were affected differently than in the JDT.CORE project: the *Extract* refactorings had a small decrease in popularity, the *Rename* refactorings saw no change at all and the *Change* refactorings saw a small increase in popularity. The PLATFORM.UI results therefore suggest that developers have started modifying their refactoring behaviors after adopting shorter releases by taking advantage of the more frequent major releases and applying refactorings that modify the API. Since we cannot draw a clear conclusion from our fine-grain analysis, we proceed with our coarse-grain analysis.

Table 2.5: Top 10 most common refactoring patterns in the PLATFORM.SWT project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Inline Variable (L)	9	17	+8
Change Variable Type (L)	4	8	+4
Change Return Type (M)	7	11	+4
Extract Method (M)	1	2	+1
Rename Variable (L)	2	3	+1
Extract Variable (L)	5	5	+0
Change Attribute Type (M)	10	10	+0
Rename Method (H)	3	1	-2
Rename Parameter (H)	6	4	-2
Change Parameter Type (M)	8	6	-2
Extract And Move Method (M)	11	9	-2
Rename Attribute (H)	12	7	-5

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

Table 2.6: Top 10 most common refactoring patterns in the PLATFORM.UI project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Extract Variable (L)	2	6	+4
Extract Method (M)	5	7	+2
Rename Variable (L)	3	4	+1
Rename Method (H)	8	8	+0
Rename Attribute (H)	9	9	+0
Rename Parameter (H)	10	10	+0
Change Variable Type (L)	1	1	+0
Change Attribute Type (M)	4	2	-2
Change Return Type (M)	7	5	-2
Change Parameter Type (M)	6	3	-3

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

Categorization analysis (Coarse granularity)

Approach. To investigate the impacts of shorter releases on the different types of refactorings being applied rather than the impacts on each refactoring pattern's individual frequency, we proceeded to aggregate the refactoring patterns by categorizing them. We were then able to abstract our analysis and use our refactoring categories to continue our comparison between traditional releases and shorter releases.

We decided to use Murphy et al.'s categorization model [162] because it is less likely to be affected by subjectivity than estimating the complexity of a specific code change.

Following the categorization criteria given in the original paper [162], the patterns detected by the RefactoringMiner were categorized into three different categories: high-level refactorings which only affect the signatures of classes methods and field, medium-level refactorings which change the signature of classes, methods, and field while also significantly changing blocks of code and finally low-level refactorings which only affect blocks of code. In short, medium-level refactorings require to change the code in several places, unlike high-level and low-level refactorings that only target a specific section of the code and therefore are less likely to affect the structure of the codebase. Based on these characteristics, we hypothesize that the frequency of medium-level refactorings should be reduced during the shorter releases due to the combination of the higher complexity involved with changing the structure of the code and shorter release time.

To categorize the refactoring patterns that were not previously categorized in Murphy et al.'s previous study [162], two authors separately classified each pattern detected by the RefactoringMiner. The two authors then combined the result of their individual classification to get the final classification for each pattern. For the cases where there were conflicts, the two authors discussed and agreed on the best categorization based on the criteria defined in Murphy et al.'s previous study [162]. After categorizing every refactoring pattern, we started to investigate if the type of refactorings applied changed between the traditional releases and the shorter releases.

We started by investigating which refactoring category (high-level, medium-level, low-level) was the most impacted by the change in release cycle. We first looked at how many patterns in each refactoring category became more common after shortening the release cycles

(ranking $\Delta < 0$), how many patterns became less common (ranking $\Delta > 0$) and how many patterns were not affected by the change (ranking $\Delta = 0$). We continued our investigation by looking at the overall variation in ranking of each refactoring category. To do so, we aggregated every refactoring pattern according to the refactoring category it belonged to. We then summed up the ranking Δ to see which type of refactoring (high-level, medium-level, low-level) showed the largest increase or decrease in popularity during both time periods.

Refactoring category Δ : To determine if a refactoring category became more or less common, we sum the ranking Δ for each pattern belonging to that category.

Findings 5. We find that medium-level refactoring patterns became less popular in the JDT.CORE project after shortening the release cycles.

From traditional yearly releases to quarterly releases, the JDT.CORE project has one less medium-level refactoring and one more high-level refactoring in its top 10 as shown in Table 2.4. From Table 2.5, we find that the PLATFORM.SWT project has one more low-level refactoring and one less high-level refactoring in its top 10 and the PLATFORM.UI project shows no difference between both periods as shown in Table 2.6. If we look at the top end of the rankings, however, we notice that in both the JDT.CORE project and the PLATFORM.SWT project, medium-level refactorings are less common than high-level and low-level refactorings during the shorter releases. This decrease in medium-level refactorings indicate that developers may not be as inclined to apply more complex refactorings during shorter releases because medium-level refactorings require to change the code in several places [162].

From the ranking Δ shown in Table 2.7, we find that in both the JDT.CORE project and the PLATFORM.SWT project, high-level refactorings got significantly more common after adopting shorter releases. The main difference between the two project, however, is that in the JDT.CORE project, medium-level refactorings became much less common whereas in the PLATFORM.SWT project, low-level refactorings became less common. Surprisingly, the PLATFORM.UI shows a different trend from the JDT.CORE project and the PLATFORM.SWT by having an increased amount of medium-level refactoring during the shorter releases. In fact, the PLATFORM.UI project is also the project with the smallest degree of variation in terms of rankings after adopting shorter releases on top of being the only project that increased medium-level refactorings. From these results, it appears that only the JDT.CORE project

Table 2.7: Ranking and workload variation between TR and SR

Refactoring type		jdt.core	platform.swt	platform.ui
Low-level	Sum of Δ	+1	+13	+5
Medium-level	Sum of Δ	+11	+1	-5
High-level	Sum of Δ	-12	-9	+0

satisfies our initial hypothesis that medium-level refactorings should see a significant decrease during the shorter releases. Our results also suggest that the PLATFORM.UI is the only one out of the three project that was able to keep up with its refactoring workload in all three categories of refactorings even after shortening the release cycle.

We therefore confirm our hypothesis and find that developers do not apply complex refactorings as much after shortening the releases in the JDT.CORE project. Although the PLATFORM.SWT project also shows a decrease in medium-level refactoring, the decrease is not significant enough to confirm our initial hypothesis. The PLATFORM.UI shows opposite results with medium-level refactoring becoming more common during the shorter releases.

Findings 6. We find that patterns that are commonly applied using refactoring tools have greatly increased in popularity during the shorter releases. If we take a look at ‘Renaming’ refactoring patterns which are commonly implemented using refactoring tools [124,188], we also find that these types of pattern became way more common during the shorter releases in two of our three projects. Using the data from Tables 2.4, 2.5, and 2.6, we sum the Δ values of ‘Renaming’ refactoring patterns in each project and find a clear increase in popularity in the JDT.CORE project and the PLATFORM.SWT with a sum of Δ of -10 and -8 respectively. The PLATFORM.UI is the only project where we find a small decrease in popularity with a sum of Δ of +1. These results indicate that shorter releases may increase the use of refactoring tools by developers.

We find that developers do not apply complex refactorings as much during the shorter releases in the JDT.CORE project. We also find that patterns that are commonly implemented using refactoring tools became more common in two of the three analyzed projects after shortening the releases.

4.4 (RQ4) Does Shortening the Releases Affect the Human Resources Dedicated to Refactoring Activities and their Refactoring Workload?

Motivation. In this RQ, we aim to determine how human resources were affected after switching to shorter release cycles. A previous study about the impacts of shorter releases on testing found that switching to shorter releases led to a higher workload for testers due to fewer testers being involved with testing activities [144]. Since releases are shorter and development teams have to keep up with the development of new features, we want to see if the number of developers that refactor is affected when adopting shorter releases.

Approach. For the first part, we investigated if the number of developers that refactored was impacted by adopting shorter releases. We first extract the total number of developers, the number of developers that refactored during the traditional releases and the number of developers that refactored during the shorter releases. We then calculated the percentage of developers that refactored during each time period and compared the results between both periods. To see if shorter releases promote developer involvement with refactoring activities, we then extracted how many developers refactored during both periods and how many developers started refactoring only after shortening the release cycles.

For the second part, we investigated how the refactoring contribution of the developers was affected by adopting shorter releases. Using refactoring instances as our metric, we measured every developer's overall contribution to refactoring activities. We first extracted the median number of refactoring instances per developer. Then, to understand how the refactoring workload is shared among developers that refactor, we looked at the median percentage of refactoring instances contribution per developer during both release periods.

Table 2.8: Developer resources during TR and SR

Project	Release	# Dev	# Refactoring Dev	% Refactoring Dev
jdt.core	TR	43	20	46.51%
	SR	35	14	40.00%
	$TR \wedge SR$	-	8	-
	$\overline{TR} \wedge SR$	-	6	-
platform.swt	TR	55	29	52.73%
	SR	45	18	40.00%
	$TR \wedge SR$	-	13	-
	$\overline{TR} \wedge SR$	-	5	-
platform.ui	TR	100	46	46.00%
	SR	105	55	52.38%
	$TR \wedge SR$	-	16	-
	$\overline{TR} \wedge SR$	-	39	-

Findings 7. We find no clear impacts of adopting shorter releases on the number of developers that refactor.

From our initial data extraction, as shown in Table 2.8, we find that two out of the three projects have a higher number of refactoring developers as well as a higher percentage of developers that refactor during traditional releases. We also notice that for the same two projects, namely the JDT.CORE and PLATFORM.SWT projects, most of the developers that are refactoring during the shorter releases were already refactoring during traditional releases. The PLATFORM.UI project is the only project of the three that differs. In fact, not only is there a higher percentage of developers that refactor during the shorter releases but most of the developers refactoring during that time period did not refactor during traditional releases.

Because the PLATFORM.UI project shows the exact opposite trend when compared with the JDT.CORE project and the PLATFORM.SWT project, we cannot conclude what impacts shortening the release cycles had on the number of developers that refactor. From our observations, a project can be positively affected by making more developers involved with refactoring as shown with the PLATFORM.UI project, or negatively impacted by having less developers refactor the code as shown with the JDT.CORE and PLATFORM.SWT projects.

Findings 8. The refactoring workload is more evenly distributed among develop-

Table 2.9: Refactoring workload contribution of developers during TR and SR

Project	Release	Median # refactoring instances	Median % refactoring instances
jdt.core	TR	28.0	0.82%
	SR	5.0	3.42%
platform.swt	TR	7.0	0.20%
	SR	3.5	3.04%
platform.ui	TR	5.5	0.29%
	SR	2.0	0.89%

ers that refactor during the shorter releases.

We investigate how the contribution of refactoring developers is affected by the change in release cycle. Using the median number of refactoring instances as our metric, we find that all three projects have less than half the number of refactoring instances per developer during the shorter releases as shown in Table 2.9. In the case of the JDT.CORE project, we observe almost six times less refactoring instances per developer. These results indicate that there is a clear downward trend in the number of refactoring instances applied to the code per developer and that most of the refactoring workload may therefore be taken care of by ‘refactoring experts’.

We therefore analyze how the total refactoring workload is split among developers. To measure the refactoring contribution of each developer that refactors, we measure what percentage of the total number of refactoring instances each developer has contributed. We then use the median to determine if the workload is more evenly shared among refactoring contributors or if most of them only take on a minor portion of the total refactoring workload.

We find that in both the JDT.CORE and PLATFORM.SWT projects, each developer take on a higher percentage of the total workload during the shorter releases than during the traditional releases. While the PLATFORM.SWT project also has a higher percentage of refactoring instances contribution per developer during the shorter releases, the increment is much smaller than the other two projects.

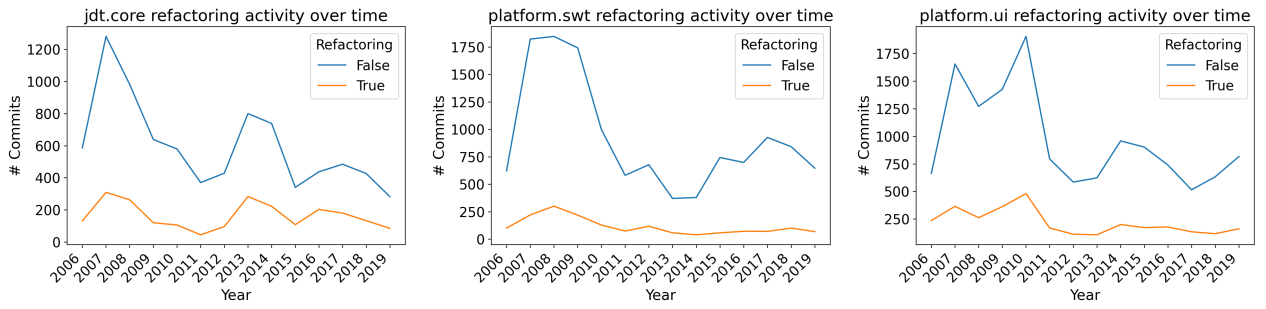


Figure 2.4: Evolution of refactoring activities over time

We find no negative impacts on the amount of human resources dedicated to refactoring after shortening the release cycle. We also find that the refactoring workload is more evenly distributed among the developers during shorter releases.

5 Discussion

5.1 Evolution of Refactoring Activities

Intuitively, it would make sense if refactoring activities diminish over time as a project ages and becomes more mature. If that was the case, then it would be less likely that the impacts observed in RQ1 and subsequent RQs would be consequences of shortening the release cycle. To verify this hypothesis, we plotted the evolution of refactoring activities since the first named version (3.2) in 2006. From Figure 2.4, we notice that the JDT.CORE seems to already have been on a downward trend of refactoring when the release cycle was shortened in 2018. The two platform projects, however, show a very stable amount of refactoring commits regardless of how many non-refactoring commits are being contributed to the projects.

Another concern we face is that the results and the statistical significance we observed in the JDT.CORE project in RQ1 may not be due to the event we are studying: the adoption of shorter release cycles. Since our results might be due to the way we separate our data, we decided to evaluate two separate chunks of data from the same time period (i.e., early TR and late TR, early SR and late SR). By comparing two subsets of data from the same time period, we can determine if the statistically significant difference in the number of commits that we found in RQ1 could be related to the change in release cycles or if this statistical

significance is due to the seemingly decreasing refactoring activity previously observed.

Table 2.10: p -values for the number of refactoring commits during early versus late TR and early versus late SR

Project	early TR and late TR statistical significance	early SR and late SR statistical significance
jdt.core	0.853	0.565
platform.swt	0.236	0.407
platform.ui	0.0215	0.0193

We therefore used a Mann-Whitney test on the number of refactoring commits in the first half of the traditional releases period versus the second half of the traditional releases period and did the same for the shorter releases period for all 3 projects. Our results, as shown in Table 2.10, show that there is no statistically significant difference in the number of refactoring commits during the traditional releases period and shorter releases period for both the JDT.CORE project and the PLATFORM.SWT project. This indicates to us that the statistically significant difference in the number of refactoring commits found between the yearly releases period and the quarterly releases period for the JDT.CORE project may have been caused by the change in release cycles.

5.2 Refactoring and Software Quality

In RQ3, we observed that certain complex refactoring operations became less common after shortening the releases. Using a previous survey based study led by Silva et al. [188], we investigate the reasons why developers apply these patterns that became less popular. More specifically, we are interested to know how they think the quality of the code is improved by applying this type of refactoring patterns. In their survey, Silva et al. investigated the refactoring behavior of developers based on eight refactoring patterns. Two of these patterns; ‘Extract Method’ and ‘Move Method’, are classified as medium-level refactoring patterns per our classification in RQ3. We therefore use these two refactoring patterns as our basis to investigate how developers think they improve the quality of the code when they apply medium-level refactorings.

During the survey led by Silva et al., developers confessed that they usually extract methods to improve the reusability and the readability of the code. In the case of the ‘Move Method’ refactorings, the developers’ answers point to a desire to increase cohesion and reusability. Because readability and reusability are not metrics that can easily be measured, we cannot empirically verify that the developers’ refactorings achieved the intended purpose.

A past study led by Hindle et al. [104] however defined and studied the concept of software “naturalness”. This code “naturalness” studies how “natural” a software’s code feels similar to how a natural language can feel natural or unnatural. A previous study led by Baishakhi and Hellendoorn [178] found that buggy code tends to be more unnatural and becomes more natural as bugs are fixed. We therefore conduct a small literature survey and find two studies led by Arima et al. [9] and Bin et al. [137] that studied the relationship between refactoring patterns and the naturalness of the code. Interestingly, both of these studies have found that the ‘Extract Method’ refactoring improve the naturalness of the code. The ‘Move Method’ refactoring, however, was found to improve the code naturalness in one study and decrease it in the other. We therefore hypothesize that the readability and reusability may indeed increase when these two refactoring patterns are applied.

To conclude, the code’s quality may decrease in the long term if some refactoring patterns are ignored due to their time cost or complexity. Developers should therefore try not to modify their refactoring behaviors even if the release cycles become shorter. On the contrary, they should make sure all types of refactorings are applied to maintain the code’s quality in the long term.

5.3 Summary and Implications

To get an overall view of the impacts of shorter releases, including our new findings on refactoring activities, we aggregated the results of related work studying the impacts of shortening release cycles on different development tasks as shown in Table 2.11. Although other studies use the Firefox project for their datasets, we notice that the impacts observed by previous studies are similar to our findings: although reducing the release cycle length has some impacts, these impacts are manageable. For example, shorter releases had an impact on how many platforms are tested in Firefox but each supported platform is tested more

thoroughly [144]. From a previous study investigating short releases and bugs, one solution to keep up with the workload was to have more developers work on each release [120]. Another study investigating the impacts of short releases on the integration delay of fixed issues in the Firefox project [38] also found that developers adjusted to shorter releases by assigning an issue to a release rather than queuing issues in a backlog.

From this case study's results we were able to see how shortening the releases may affect a project's refactoring activities by looking at the JDT.CORE project's evolution. From RQ1 and RQ2, the decreased weekly number of refactoring instances, the decreased weekly percentage of commits that refactor the code and the lower weekly median number of lines refactored indicate that refactoring activities in the JDT.CORE project may have been negatively affected by the shorter releases. From RQ3, we also found that specific types of refactorings are becoming less common after changing the release cycle in the JDT.CORE project. Past studies have shown that refactoring improves software quality and maintainability [124, 172] which leads to fewer bugs in the long term. In Section 5.2, we also found that the refactoring patterns that were less common during the shorter releases were associated with readability and reusability; two aspects of software quality that can deteriorate in the long term if not addressed. Short releases adopters should therefore take into account the risks of shortening their release cycles on refactoring activities and assign their resources accordingly.

By looking at the PLATFORM.UI project, however, our results also show how a project could benefit from the shorter releases while also keeping up with its refactoring workload. From RQ1 and RQ2, we found that the shorter releases did not impact the amount of refactoring being done in the PLATFORM.UI project. From RQ3, we even find that medium-level refactorings increased after shortening the releases. These results indicate that the developers are adapting their refactoring behaviors with the new release cycles. Since every release is a major release with the new release model, the developers now have the flexibility of refactoring their API during any release and our results show that they are actively taking advantage of that. From RQ4, we also find that a higher percentage of developers refactor during the shorter releases in the PLATFORM.UI project. From this case study, short releases adopters can therefore also see how their project could benefit from shortening their release cycles.

Table 2.11: Impact of shorter releases on software development tasks

	Refactoring Our study	Integration [38]	Bugs [120]	Testing [144]
RQ1: Does adopting shorter releases affect the amount or the frequency of the task	Shorter releases may reduce how much refactoring is applied to the code	Issues are queued up as backlog in traditional releases versus a per-release basis in shorter releases.	Harder bugs are propagated to later releases. Fewer changes per developers but the changes touch more files.	Fewer platforms tested in total but each supported more thoroughly.
RQ2: Does adopting shorter releases affect the effort dedicated to the task	Shorter releases may impact refactoring efforts	Not studied	Not studied	Not studied
RQ3: Does adopting shorter releases affect the characteristics of the task	In some cases, refactorings that affect many parts of the code are not applied as frequently during shorter releases	Issues addressed more quickly but take more time to be integrated. The total time from the issue report date to its integration is not significantly different.	Negligible difference in the number of post-release bugs. Bugs are fixed more quickly but fewer bugs are being fixed.	More tests executions per day but the tests focus on a smaller subset of the corpus. Overall fewer tests but larger builds.
RQ4: Does the change to shorter releases affect the human resources dedicated to the task	The refactoring workload is split more evenly among developers	Not studied	More developers working on each short release.	Fewer testers but higher workload.

While it is not yet clear as to why some projects adapt better to shorter releases than others, we hypothesize that one of the possible reason as to why the JDT.CORE is having a harder time keeping up with its refactoring workload might be due to the nature of the code itself. Since the PLATFORM.UI project is a UI project, the chances of having to refactor the API is most likely lower than in the JDT.CORE project which is made of core application code. As we have seen previously, the Eclipse community also tends to be more reluctant to refactor their API in order to maintain a strong backward compatibility [21]. The same study also finds that the developers are aware of how many downstream users can get affected by their changes and are therefore sensitive about making breaking changes. Since the JDT.CORE project makes up the core of the Eclipse application and can affect all other aspects of the platform, we therefore assume that the developers might be less inclined to refactor their API as much as in other projects.

5.4 Threats to Validity

Internal threats: In RQ1 and RQ2, we aggregate our data on a weekly basis and use the weekly median values for our analysis. Aggregating the data on a weekly basis allows us to have enough data points to follow the evolution of refactoring activities while also keeping the time frame short enough to avoid having outliers significantly affect our results.

In RQ2, we removed all ‘Rename’ refactoring types because more than 50% of developers admit using refactoring tools for renaming in previous studies investigating the usage of refactoring tools [188]. Still, several refactoring types (e.g., ‘Extract Local’) can be performed by tools, which might affect the results of RQ2. As of today, there is still no tool that can allow us to detect if a refactoring was performed manually or automatically. Filtering out patterns that are commonly implemented using refactoring tools based on previous studies’ findings is therefore the best approach we currently have to minimize refactoring tools’ impacts on our results.

In RQ3, we categorize each refactoring pattern based on a previous study authored by Murphy et al. [162]. We know that this categorization is not optimal because it does not allow us to accurately categorize refactoring patterns based on the complexity of the task or the efforts required to apply each refactoring pattern. However, it does allow us to get a hint

of both complexity and efforts by taking into account which sections of the code need to be changed to apply each refactoring pattern.

External threats: Because we only used open-source software projects from the Eclipse foundation, the results shown in this study may not be representative of other software systems. We also found that the Eclipse foundation gradually adopted shorter releases over a long period of time unlike other known projects using short releases such as Firefox. The impacts of shorter releases on the projects investigated in this study may therefore differ from other projects that chose to adopt short releases more abruptly. Because we observed different impacts of shorter release cycles on all three of our projects, it is difficult to generalize our results.

Constructive threats: Code churn has been used by multiple past studies as a measure of effort [10,115,149,153]. However, it has also been suggested that code churn is not a direct measure of effort but rather one of many metrics used to measure development effort [185]. We therefore only get a general idea of the actual efforts put into refactoring by using refactoring churn. Although we used a state of the art refactoring detection tool that was used several times in previous studies [107,163,190,193], we have no guarantee that all refactorings applied to the code were detected.

6 Chapter Summary

In this study, we investigated the impacts of shortening the release cycles on refactoring activities in three Eclipse projects which all developed using both traditional yearly releases and shorter quarterly cycles. Through this case study, we found that shortening the release cycles may impact the refactoring activities of a software project in various ways. We also found that some of the refactoring patterns that were less common after shortening the releases are usually applied to improve the readability and reusability of the code. Not applying them could therefore affect the software's quality in the long term. This case study further contributes to the empirical knowledge related to shortening the release cycles. With the results we present in this study, potential short releases adopters now have a better insight of the potential impacts on refactoring activities that can occur from shortening their release cycle. When aggregated with previous studies that study the impacts of shortening the

release cycles, our results allow short releases adopters to have an overall picture of which area of development can be impacted when shortening the release cycles. It also allows them to dedicate resources accordingly to make the transition as smooth as possible.

Intuitively, it would make sense to observe less refactoring, more bugs, or other issues when shortening a project's development cycle due to the short amount of time available to developers. One interesting factor that however seems common across studies investigating the impacts of shorter release cycles is that developers can adapt to the shorter releases and manage software development tasks even with less time between each release. Although the impacts of shorter seem manageable, it would be interesting to study in the future what problems early adopters such as Google and Firefox have encountered when adopting shorter releases and how these issues were solved. In the case of our study, for example, while we cannot prove with certainty that developers are using more refactoring tools to keep up the with refactoring workload, our results do seem to indicate that more tools may have been used to mitigate the impacts of the shorter releases.

Chapter 3

Exploring the Adoption of Fuzz Testing in Open-Source Software: A Case Study of the Go Community

1 Introduction

As revealed in Anaconda’s 2022 yearly survey on the state of data science [13], the fear of vulnerabilities (such as the Log4j vulnerability¹) in open-source software has caused private companies to start scaling back their use of open-source software. To address the issue of vulnerabilities in open-source software systems, developers are increasingly turning to automated testing strategies, such as fuzzing [1, 184]. Fuzzing is a software testing technique where a large number of inputs are sent to a software system with the goal of triggering unexpected behaviors. While unit tests allow developers to test a software system against expected inputs, fuzzing complements unit tests by testing against unexpected or abnormal inputs. Using fuzzing, developers have been able to detect multiple software vulnerabilities and fix them before an attacker can exploit them (e.g., CVE-2016-6978,² CVE-2017-3732,³ CVE-2019-16411⁴).

¹<https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance>

²<https://nvd.nist.gov/vuln/detail/CVE-2019-16411>

³<https://nvd.nist.gov/vuln/detail/CVE-2017-3732>

⁴<https://nvd.nist.gov/vuln/detail/CVE-2016-6978>

As of 2024, fuzzing is mostly conducted using specialized external tools called “fuzzers” that generate inputs and send them to a target software system. Since the launch of OSS-Fuzz in 2016 [69, 169], a substantial portion of fuzzing activities in the open-source software ecosystem has been conducted via external services. These external services allow open-source projects to benefit from continuous fuzzing without carrying the heavy computational costs of fuzzing.

Due to the recent increase in interest of the open-source community towards fuzzing [1, 184], other initiatives aimed at making fuzzing more accessible have also been launched. For instance, in 2021, the Go community decided to integrate fuzzing (issue #44551) within the Go programming language itself so that developers can write fuzz tests just as easily as unit tests. Additionally, the efforts of the open-source community to make the adoption of fuzzing easier led to Go developers adding features to facilitate the adoption of OSS-Fuzz (issue #50192) and also led OSS-Fuzz developers to add native Go fuzzing support [5].

Since unit tests do not scale well to test against large amounts of abnormal inputs, the increased accessibility of fuzzing now offers developers the option to adopt fuzz testing as part of their standard software testing efforts. With the considerable security benefits that can come from combining deterministic testing, such as unit testing, with scalable approaches, such as fuzzing, the research community must help in making the adoption of fuzzing as easy as possible.

As of 2024 (two years after the introduction of fuzz tests into the Go standard library), **we still do not know if open-source communities have started using fuzzing to make their open-source projects more secure and avoid the spread of vulnerabilities throughout the ecosystem.** In this chapter, we therefore set out to study the co-evolution of fuzz testing alongside standard testing activities in open-source projects hosted on GitHub. To provide an overview of the state of fuzz testing in OSS projects, we conduct a conceptual replication of McIntosh *et al.*’s analysis of build maintenance effort [148] and aim to answer the following preliminary questions:

1. We find that Go fuzz tests are rarely added by developers two years after their introduction into the standard Go library. Our analysis reveals that only 3.58% of the studied projects contain any fuzz tests and those fuzz tests only account for 1.35% (on

median) of all test functions.

2. We observe significantly more commits contributing to non-fuzz test code than fuzz test code. Changes to non-fuzzing test code also tend to be larger than changes to fuzz tests.

Building upon the results of these preliminary results, we further investigate to answer the following research questions:

1. We find that neither non-fuzzing test code changes nor source code changes are tightly coupled with fuzz test changes. Our results suggest that fuzz test development is not being conducted alongside other types of test development, such as unit tests.
2. We find that most projects rely on few contributors to contribute most of the testing code (both fuzzing test code and non-fuzzing test code). Conversely, we find that significantly more developers have contributed to non-fuzzing test code, whereas most developers have not made any contribution towards fuzzing activities.

Our results lead us to conclude that non-fuzz testing is still the backbone of testing activities in OSS projects. From our investigation, we also believe that fuzzing might be best used on parts of a codebase that requires very thorough testing. From the number of developers that have actually conducted fuzzing development, we also conclude that more work need to be done to lower the barrier of entry to fuzz testing.

2 Background and Related Work

2.1 Background

Challenges in adopting fuzzing. Every activity related to software development and maintenance brings its own set of challenges. Kim *et al.* conducted a field study and a series of interviews with refactoring experts at Microsoft to study the challenges of refactoring [125]. They find that refactoring activities are both risky and costly with developers spending on average 10% of their work hours refactoring code. Similarly, Greiler *et al.* previously interviewed 25 senior practitioners to learn more about the challenges of testing plug-in

based systems [99]. Their study reveals that even experienced developers face difficulties writing tests for their software systems.

As fuzzing slowly gets integrated into the software testing process over time [64, 143, 158], software testers and developers will need to manage the challenges incurred by fuzz testing. In 2023, Nourry *et al.* conducted a survey with fuzzing experts that revealed over 22 distinct challenges of conducting fuzzing activities [165]. They find that the usability of fuzzers (i.e., setting up, building, or using a fuzzer) is the current main challenge of conducting fuzz testing. Thus, making fuzzing more accessible could encourage the incorporation of fuzzing into standard testing activities.

Native Go fuzzing. Fuzz tests were integrated into the Go language itself in release 1.18 on March 15th 2022 [4]. To facilitate the adoption of this native fuzzing feature, the Go language developers purposely designed fuzz tests to be similar to unit tests to facilitate the conversion between normal tests and fuzz tests.⁵ We surmise that this native and straightforward feature could help address the usability challenges of conducting fuzzing activities in OSS projects. With this new feature simplifying the development of fuzz tests, we therefore aim to find out if open-source communities are starting to use fuzz testing to make their projects more secure. Thus, we analyze how frequently developers employ fuzz tests and how they evolve alongside non-fuzzing tests.

Listing 1 shows an example of a fuzz test retrieved from the `babylonchain/babylon` project. This function tests the `GenKeyPair` function, which is designed to generate a pair of encryption and decryption keys. As we can see, the structure to write Go fuzz tests closely resemble Go unit tests.

Developers first define the seed values that will be sent to the target function using `f.Add`. Then, developers must write the code that will test the target function and wrap it with the `f.Fuzz` function to make the test into a fuzz test. Unlike with unit tests, the initial values provided to fuzz tests will also be used to generate new inputs for further testing.

⁵<https://github.com/golang/go/issues/44551#issuecomment-785176317>

⁶Retrieved from `babylonchain/babylon`: https://github.com/babylonchain/babylon/blob/682c2d238ca305243d361a512bd0c401eedef1c6/btctxformatter/formatter_test.go#L96

```

// This fuzzer checks if decoder won't panic with whatever bytes we point it at
func FuzzDecodingWontPanic(f *testing.F) {
    f.Add(randNBytes(firstPartLength), uint8(rand.Intn(99)))
    f.Add(randNBytes(secondPartLength), uint8(rand.Intn(99)))

    f.Fuzz(func(t *testing.T, bytes []byte, tagIdx uint8) {
        tag := []byte{0, 1, 2, 3}
        decoded, err := IsBabylonCheckpointData(tag, CurrentVersion, bytes)

        if err == nil {
            if decoded.Index != 0 && decoded.Index != 1 {
                t.Errorf("With correct decoding index should be either 0 or 1")
            }
        }
    })
}

```

Listing 1: Fuzz test example⁶

2.2 Related work

The co-evolution of source code and non-source code. McIntosh *et al.* investigated build maintenance efforts and the logical coupling between source code, test files and build files [148]. They find that build maintenance introduces significant overhead to software development (up to 27% on source code development and 44% on test development). Their results reveal that most developers are impacted by build maintenance. Finally, they find that hiring build experts can greatly reduce the number of developers impacted by build maintenance.

Zaidman *et al.* studied the co-evolution of production and test code in both industrial and open source projects [213]. They find that projects do not increase their testing activity as they get closer to a new release. They also find that an increase in test code is often linked with an increase in coverage. Jianget *al.* studied the co-evolution of infrastructure code and source code in 265 OpenStack projects [112]. They find that infrastructure files are frequently modified and are tightly coupled with test files.

In this study, we examine the co-evolution of fuzzing code and non-fuzzing testing code. Thus, our work aligns with previous research that investigates the co-evolution of source and non-source code. Our work differs from past studies by investigating developer usage of standard test code and fuzzing code when testing open-source projects.

The usage of new language features. The adoption and usage of new language features has been a recurring topic of research in software engineering studies. Dyer *et al.* investigated the use of new Java features in over 31,000 open-source Java projects [51]. They find several areas where developers could use new Java features but do not. Their investigation also shows that the adoption of a new language feature is likely to be driven by compiler support. Moreover, they find that contributors increasingly start using a feature once the feature is used for the first time in the codebase.

The reluctance of developers to adopt new language features seems to be a common pattern in studies investigating the evolution of a programming language. Mazinianian *et al.* [146] studied the use of lambda expressions in Java and surveyed 97 developers that adopted the lambda expression feature. Their investigation shows that the use of a new feature can take some time before it is widely adopted. They also find that developers tend to use built-in features inefficiently by using general features over specialized ones. Similarly, Scarsbrook *et al.* studied the adoption of TypeScript features over time in their 2023 study [180]. Their observations align with the results of Mazinianian *et al.* where they find that the Typescript community tends not to use new language features.

Similar to these studies, our study provides better insights into how developers are adopting fuzz testing as a new feature. Our preliminary analysis (described in Section 4.1) shows that fuzz testing is still not widely adopted with only 3.58% of projects having fuzz tests. Furthermore, fuzz tests make up just 1.35% of testing functions. Our observations align with previous studies' conclusions where they found that new features are not always adopted by developers.

Empirical studies on fuzzing. With the increased availability of open-source empirical data brought by the release of OSS-Fuzz, researchers have been able to conduct more empirical studies on fuzzing in recent years. Ding *et al.* conducted a large scale empirical study of over 23,000 OSS-Fuzz bugs spanning over 316 projects [46]. Their results show that fuzzing is an effective technique to find bugs and that developers are generally quick to fix detected bugs. They, however, find that some flaky bugs detected by fuzzing are problematic to address or reproduce. Moreover, they find that fuzzing campaigns tend to find large amounts of bugs in short timeframes requiring developers to deal with high amounts of bug reports at once.

Böhme *et al.* conducted a four cpu year-long fuzz campaign to study the increasing cost of fuzzing overtime [23]. Their study shows that the computational cost of fuzzing gets exponentially more expensive over time in order to find bugs. Their results prove that improving fuzzers yields more potential improvements to the efficiency of a fuzzing campaign than adding more computational power.

The challenges of adopting fuzzing were also studied in both an industrial and an academic setting. Nourry *et al.* [165] data mined fuzzing-related GitHub issues of open-source software repositories. After manually labeling these issues, they created a taxonomy of 22 challenges related to conducting fuzzing activities and validated their results by surveying fuzzing practitioners. Some of the challenges described in Nourry *et al.*'s study had previously been highlighted in a prior industrial case study by Lian *et al.* where a team of developers at Huawei tried to start fuzzing one of their libraries [136].

Our work makes the following contribution: (1) we conduct a large-scale empirical study on 11,341 projects to study the usage and adoption of native Go fuzzing, (2) we provide new insights into how open-source communities conduct testing activities, (3) we propose a list of candidate functions for developers to fuzz based on our investigation of how fuzz testing is used in practice.

3 Studied Data

Since the goal of our study is to investigate the usage and evolution of fuzz testing in OSS projects, we first needed to find projects that contained fuzz tests. In this section, we describe our approach for selecting projects and locating code and commits to be analyzed within the selected projects. Figure 3.1 provides an overview of the data collection process.

3.1 Selection of Target Projects

We first used the GitHub search tool [39] published by Dabice *et al.* and maintained by the SoftwarE Analytics Research Team (SEART) to find projects. To collect all projects, the only criteria we used to filter our search was that the project had to use the Go language. This resulted in a total of 62,702 projects (as of March 2024). We then filtered out projects that were forks which brought our number of projects down to 60,982 projects.

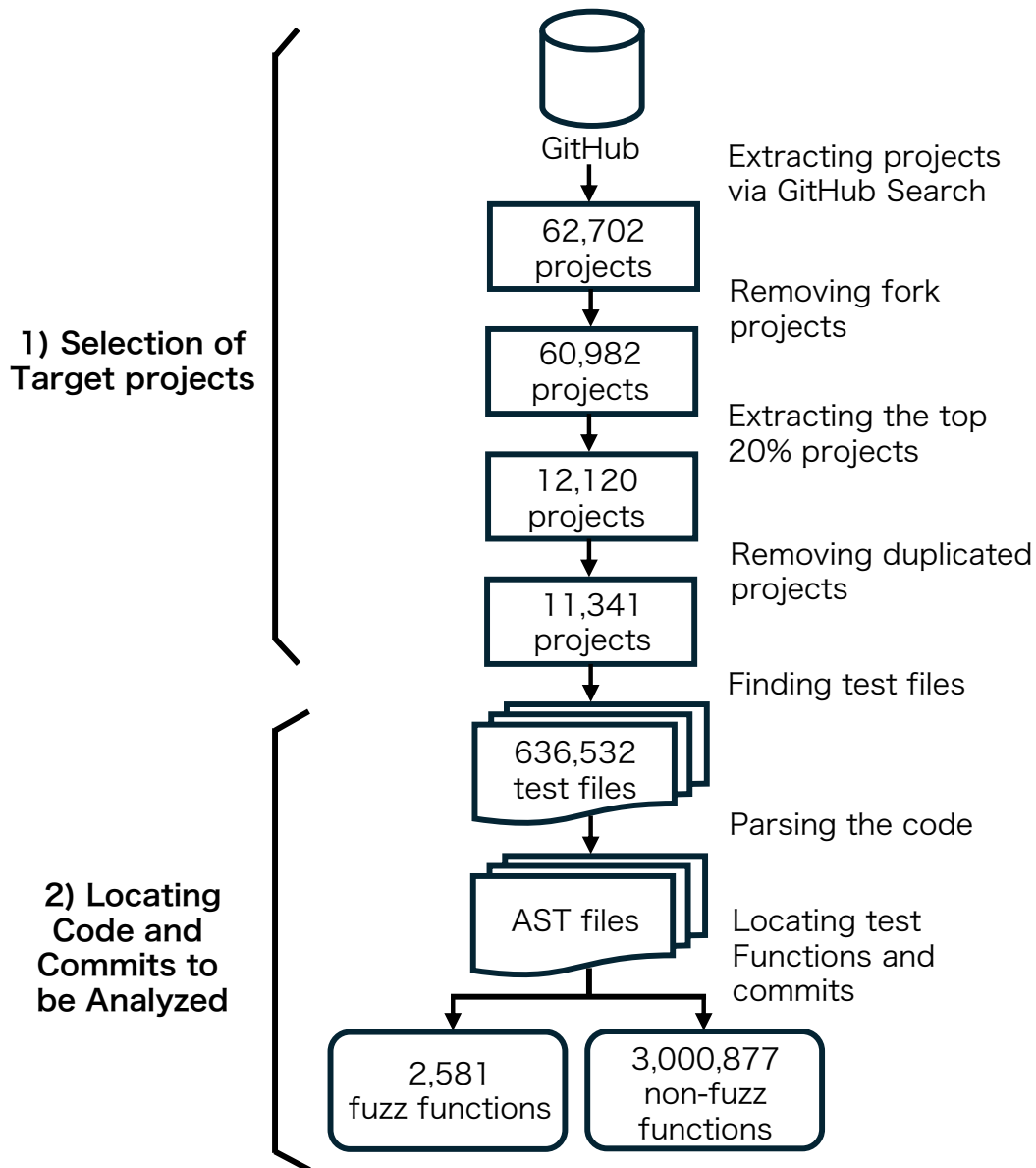


Figure 3.1: Overview of the data collection process

Next, we wanted to minimize the chance of having toy projects in our dataset since we are mostly interested in how open-source projects use fuzzing. We therefore manually filtered the dataset to only keep the top 20% (12,120 projects) with the highest number of commits ($x > 246$ commits). Because larger projects tend to have larger communities, we also assumed that big projects are more likely to have open source contributors conducting fuzz testing within their development team. By keeping only big projects, we therefore hoped to increase our odds of finding fuzz tests in the projects' testing codebase.

While checking our data for inconsistencies, we found that some of the projects in our dataset were duplicated due to having been renamed. For these cases, although two projects had different URLs, clicking the GitHub URL would redirect to the most up to date GitHub repository with the correct URL. To find these cases, we therefore used the python *requests* package to automatically verify the redirecting URL. To avoid data duplication, we therefore removed any project in our dataset whose URL was redirecting to the latest up-to-date repository. This process further brought down our number of projects to 11,341. We then cloned these repositories and proceeded with the code parsing process.

3.2 Locating Code to be Analyzed

Finding test files. When programming using the Go language, developers must follow specific rules in order to write test code. As described in the official Go documentation [2], all test code must be located in a file whose name ends with “*_test.go*”. To find test code in our selected repositories, we therefore looked for all files ending in “*_test.go*”. In total, we were able to locate 636,532 test files in our dataset.

Parsing the code. After locating the test files, we used the Go standard library’s *parser package* ⁷ to parse all Go test files in our dataset. The output of the parser gave us Abstract Syntax Trees (AST) in JSON format showing all test function names along with their return and parameter types. Additionally, the parser also gave us the starting and ending line numbers of each function in the test files. The parsing process was done on all test files for every revision in our studied projects’ history.

Locating test functions. Next, we aimed to identify all test functions (both non-fuzzing test functions and fuzzing test functions) in the test files. To write a fuzz test, developers must follow the naming conventions described in the official Go documentation [3]. Namely, the function name of a fuzz test must start with the word “*Fuzz* (e.g., *FuzzTestOne*)”. Also, the fuzz target must be a function call to the *(*testing.F).Fuzz* function with **testing.T* as the first parameter type.

To locate fuzz tests, we used the ASTs obtained from parsing the test files and looked for functions that satisfied the naming requirements. Since the parser gave us the name of all

⁷<https://pkg.go.dev/go/parser>

functions in the test files, we also looked for functions that satisfied the naming requirements for non-fuzzing tests such as unit tests (the function name starts with the word “*Test*” and the first parameter must have the “**testing.T*” type).

Labeling commits. After locating fuzzing tests/code, we used the AST files to know the starting and end line of each test function for every revision. For each commit, we then used git to extract the name of the files changed along with the line numbers where the changes happened. From there, we matched the line numbers from the AST output and the “diff” output to label all commits as “non-fuzzing”, “fuzzing”, or “source” based on if the changes modified fuzzing code, non-fuzzing test code, or code outside of test files. Note, one commit can have multiple labels.

4 Preliminary Analysis

4.1 Do Developers Use Fuzz Testing? (PQ1)

Motivation. Fuzz tests were only added officially to the Go language in 2022. Because this feature is so recent, we still do not know to what extent native Go fuzzing is used by developers to uncover software vulnerabilities. We therefore aim to know what ratio of test files contain fuzz tests and how many fuzz tests projects tend to have.

Approach. From the process described in Section 3.2, we were able to find the number of test files, non-fuzz test functions and fuzz test functions in our studied projects. In this first preliminary analysis, we therefore aim to report how much fuzz testing was being conducted in open-source projects at the time of data collection (March 2024).

To get an overview of how much fuzz testing is used in practice, we first calculate the ratio of projects that contain at least one fuzz test. From this subset of projects, we then calculate what percentage of test files contain fuzz tests. Using the percentage of test files that contain fuzz tests in each project, we also calculate the median percentage of test files that contain fuzz tests across all projects.

From the file level, we then dig deeper to the function level to calculate the proportion of test functions that are fuzz tests. To do so, we divide the number of fuzzing test functions over the overall number of test functions in a given project. We then repeat the same aggregation

process we did for the file level analysis to calculate the median percentage of fuzz functions over all test functions across all projects.

Results. We find that native Go fuzzing features are still underutilized by developers with only 406 projects (3.58%) out of 11,341 projects containing fuzz tests. When looking at the 406 projects that contain fuzzing code, we find that the median ratio of test files that contain fuzz tests is only 3.15%. Figure 3.2 shows the distribution for the number of non-fuzz test functions and the number of fuzz test functions in projects that contain at least one fuzz test. As shown in the figure, non-fuzzing tests are still far more common than fuzz tests with a median number of 207 non-fuzzing tests per project versus only 2 fuzz tests per project.

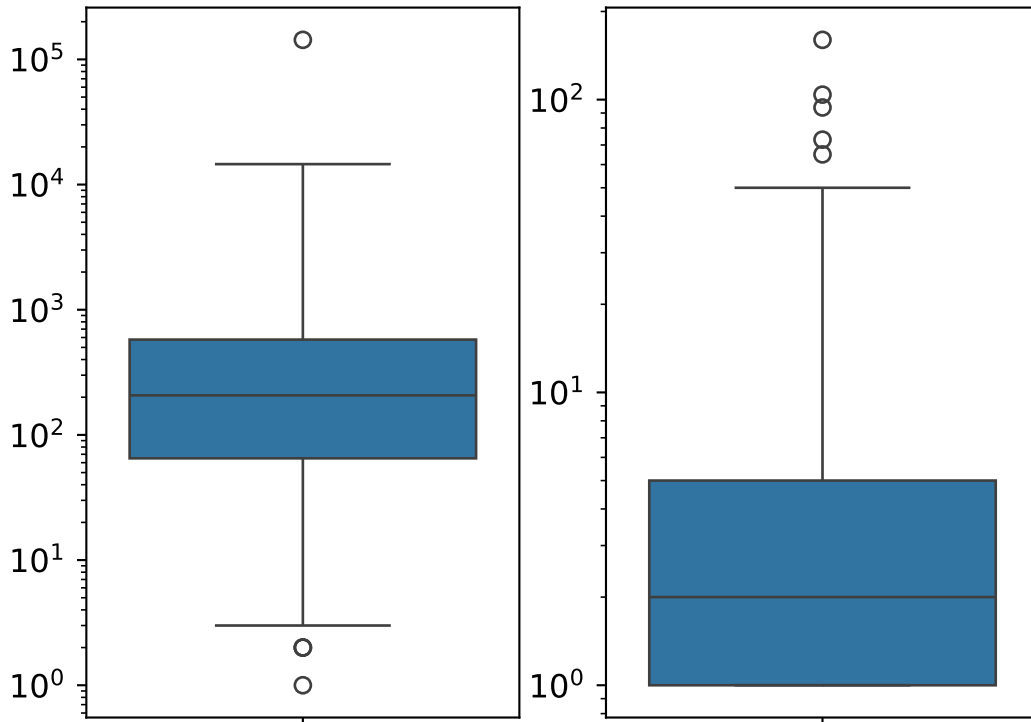


Figure 3.2: Number of non-fuzz tests (left) and fuzz tests (right) per project containing at least one fuzz test (log scaled)

Continuing with our subset projects that contain fuzzing code, we then calculate what percentage of test functions are fuzz functions in each project. Using the ratio of fuzz functions in each project, we then calculate the median ratio of fuzz tests across all projects and find that fuzz tests only account for 1.35% of testing functions. Overall, our results indicate

that two years after being introduced, fuzz tests only account for a small fraction of software testing in our studied projects.

We find that most projects do not conduct fuzz testing to uncover software vulnerabilities. For projects that do fuzz their software system, fuzz tests only account for a small fraction of testing functions. These results indicate that more work needs to be done to spread the adoption of fuzzing among open-source communities.

4.2 Is Fuzzing Test Code More Prone to Change Than Non-Fuzzing Test Code? (PQ2)

Motivation. While the results of PQ1 reveal the extent to which developers use fuzz testing, these results do not provide insights into how frequently developers develop fuzz tests or the amount of effort required to maintain fuzzing activities. To get a basic idea of the amount of effort that developers put into fuzzing their software systems, we investigate the monthly commit activity of fuzz tests and the rate of change of fuzzing test code per commit (churn). We then calculate the same metrics for standard test code (i.e., non-fuzzing test code) to compare the rate of development of standard tests and fuzz tests. We hypothesize that non-fuzzing test code requires more maintenance efforts than fuzzing test code because they require developers to manually write code to test against new inputs. Fuzz tests, however, use the fuzzing engine to automatically generate new inputs and may therefore not require as many interventions from developers.

Approach. Starting from the filtered dataset described in Section 3.2, we further filter out commits dating before the integration of fuzz tests into the official Go standard library.

To determine if a commit changed fuzzing test code or standard test code, we first used the starting and ending line numbers of each test function obtained from the parsing process described in Section 3.2. Next, we used git's *diff* feature to find out what lines were changed in each commit in our dataset. By matching the line numbers of test functions with the output of the *diff* command, we were able to find all changes that happened in either fuzz test functions or non-fuzz test functions. From there, we further filtered our dataset to remove all commits that did not modify a test function.

For the remaining commits, we calculated the testing code churn in each commit by summing up the number of lines added, deleted or modified in test functions. Using the number of lines changed (i.e., added, deleted, or modified) in fuzzing test code and non-fuzzing test code, we were able to calculate the median number of lines changed per commit across all projects. To get an overview of testing activities since the introduction of fuzz tests into the standard library, we then plotted the monthly number of commits involving fuzzing test code and non-fuzzing test code across all projects.

Results. We find that non-fuzzing test functions tend to have larger changes than fuzz test code changes. From the filtering process described in Section 3.2, the removal of commits predating the introduction of fuzz tests into the standard library, and the removal of commits that do not target test functions, we end up with a dataset of 125,927 commits.

Using the number of fuzz test code lines impacted in each commit in our dataset, we first calculate the median number of lines impacted inside fuzz test functions per “fuzzing commit”. We then repeat the same process for non-fuzzing test functions and find a median of 17 lines of code changed per fuzzing commits and 33 lines of code changed per non-fuzzing test functions. Our results indicate that fuzz tests might require less development efforts to maintain than non-fuzzing tests. This could be explained in part by the fact that fuzz tests do not require developers to manually write each input used during testing.

We find that open-source communities commit significantly more non-fuzzing test code than fuzzing test code on a monthly basis. We continue our preliminary analysis by investigating the monthly number of commits involved with test code modifications. Figure 3.3 shows the monthly number of commits modifying non-fuzzing test functions and the monthly number of commits modifying fuzzing test functions between the introduction of fuzz tests into the standard library in March 2022 and the time of data collection in March 2024. During this period, we calculate 102 median number of commits per month that change fuzz test code across our studied projects. This is a stark contrast from non-fuzzing test code where the median monthly number of commits that modify non-fuzzing test functions is 4,535. Overall, our results indicate that developers still see non-fuzzing tests such as unit tests as the main avenue to test a software system. Consequently, it is likely that most

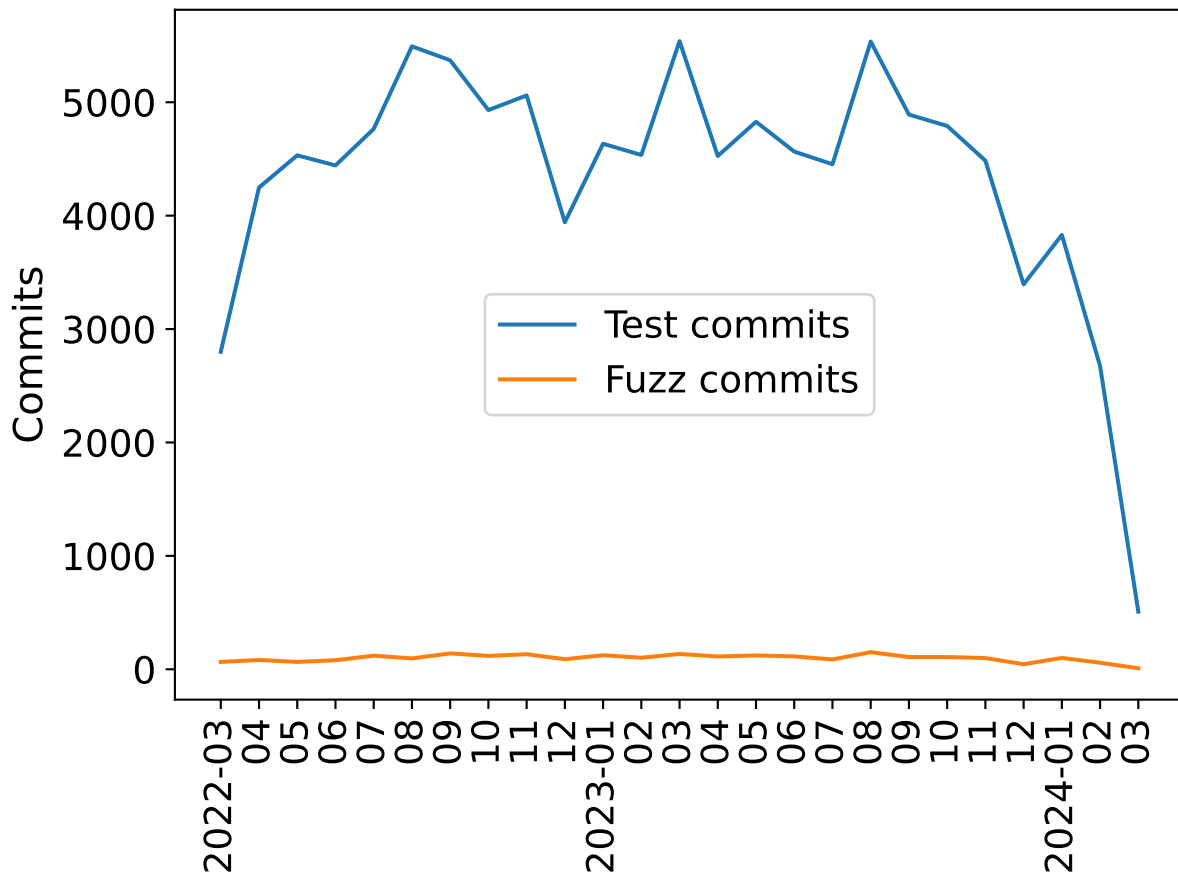


Figure 3.3: Number of commits modifying non-fuzzing test functions and number of commits modifying fuzzing test functions per month since fuzz tests were added to the standard library.

of the testing efforts in open-source projects are dedicated to deterministic testing over fuzz testing.

We find that open-source software projects both commit significantly more often and commit larger changes to non-fuzzing test code over fuzzing test code. Our results suggest that open-source communities tend to spend more development efforts toward non-fuzz testing than fuzz testing.

5 Is Fuzz Testing Development Conducted As Part of the Standard Testing Process? (RQ1)

Motivation. In Section 4.2, we found that non-fuzzing test code is committed at a much higher rate than fuzzing test code. In this section, we aim to get a better understanding of how open-source communities conduct their testing activities. Specifically, we investigate if changes to fuzzing test code and standard test code are typically done at the same time. From this analysis, we aim to find out if fuzzing development is conducted as part of standard test development or not.

Approach. We use association rules [6, 141, 157, 221] to measure the coupling between source code changes, non-fuzzing test code changes, and fuzz testing code changes. In formal terms, association rules are defined as statistical descriptions of the co-occurrence of elements in a dataset [6]. In other words, they are rules used to find associations and relationships between different types of data points. Table 3.1 shows how we calculated the association rules used in this paper.

Table 3.1: Association rule calculations

Rule	Calculation
Support(X)	$\frac{\# \text{ type X revisions}}{\# \text{ total revisions}}$
Conf(X $\rightarrow$ Y)	$\frac{\text{Support}(X,Y)}{\text{Support}(X)}$
Lift(X $\rightarrow$ Y)	$\frac{\text{Conf}(X \rightarrow Y)}{\text{Support}(Y)}$

For our use case, we first use the support rule to know the proportion of commits that contain a specific type of change (source, non-fuzzing test, fuzz test). To measure the coupling relationship between our studied changes (source, non-fuzzing test code, fuzzing test code), we then use the confidence rule which measures the strength of the implication that a change

to X will be accompanied by a change to Y. For example, let's assume our dataset contains 9 commits in total (5 commits that modify the source code, and 4 commits that change non-fuzzing test code). If one of these commits changes both the source code and non-fuzzing test code, then the $Conf(Source \rightarrow Test)$ would be measured as $(1/5) = 0.2$ and the $Conf(Test \rightarrow Source)$ would be measured as $(1/4) = 0.25$. Note that the confidence value is not symmetrical as shown in the example above ($Conf(Source \rightarrow Test) \neq Conf(Test \rightarrow Source)$).

The last rule we use is the Lift association rule. The lift value represents the ratio between the observed rate of co-occurrence and the rate of co-occurrence that would have been expected due to random chance. A lift value of 1 indicates that two variables are independent of each other. A positive lift value greater than 1 means that we are observing more instances of an event than we would expect from a random distribution, and a value lower than 1 indicate that we are observing less instances than we would expect from a random distribution. For our use case, we use the lift value to know if our distributions are likely obtained due to chance or not. Note, the lift value is symmetrical ($Lift(X \rightarrow Y) = Lift(Y \rightarrow X)$).

Results. Table 3.2 shows the median association rule values across all projects. Looking at the support metrics, we find that source code changes occur at a 2:1 rate when compared with test changes (0.667 and 0.309 respectively). The median support value for fuzz tests (0.009) further indicates that most commits do not contain changes to fuzz test code. This is in line with our preliminary results where we found that developers do not use fuzz tests as much as non-fuzzing tests.

Source code changes are much more likely to be accompanied by non-fuzzing test code changes than fuzzing test code changes. The median $Conf(Source \rightarrow Test)$ value reveal that almost 37% of source code changes are accompanied by non-fuzzing test code changes. This result shows that testing activities are well integrated within the software development process in our studied projects. However, the median $Conf(Source \rightarrow Fuzz)$ value shows that fuzz testing is not a consistent part of software development two years after fuzz tests were added to the standard library. From the lift values, we observe a small positive relationship between changes for files involving source code changes. For standard

Table 3.2: Median association rule values calculated across our dataset

	Association	Median
Support	Source	0.667
	Test	0.309
	Fuzz	0.009
	Source/Test	0.249
	Source/Fuzz	0.005
	Fuzz/Test	0.004
Confidence	Source $\rightarrow$ Test	0.369
	Source $\rightarrow$ Fuzz	0.007
	Test $\rightarrow$ Source	0.861
	Test $\rightarrow$ Fuzz	0.015
	Fuzz $\rightarrow$ Source	0.909
	Fuzz $\rightarrow$ Test	0.750
Lift	Source $\rightarrow$ Test	1.239
	Source $\rightarrow$ Fuzz	1.204
	Test $\rightarrow$ Fuzz	2.393

test and fuzz test changes, however, the higher lift value indicates that the coupling values we calculated are unlikely to be due to chance.

Fuzz tests do not seem to be conducted as part of regular testing activities. With the $\text{Conf}(\text{Test} \rightarrow \text{Fuzz})$ value being twice as large as the $\text{Conf}(\text{Source} \rightarrow \text{Fuzz})$ value, we find that fuzz tests are more likely to be added during testing activities than source code development. However, with only 1.5% of testing commits containing fuzzing test code changes, our results indicate that fuzz testing is not a common testing activities in our studied projects. This pattern is further highlighted by the $\text{Conf}(\text{Fuzz} \rightarrow \text{Source})$ and $\text{Conf}(\text{Fuzz} \rightarrow \text{Test})$ values (0.909 and 0.750 respectively) and the distribution of confidence values in Figure 3.4 which show that fuzz test changes seem to be accompanied by both source code changes and standard test code changes indiscriminately.

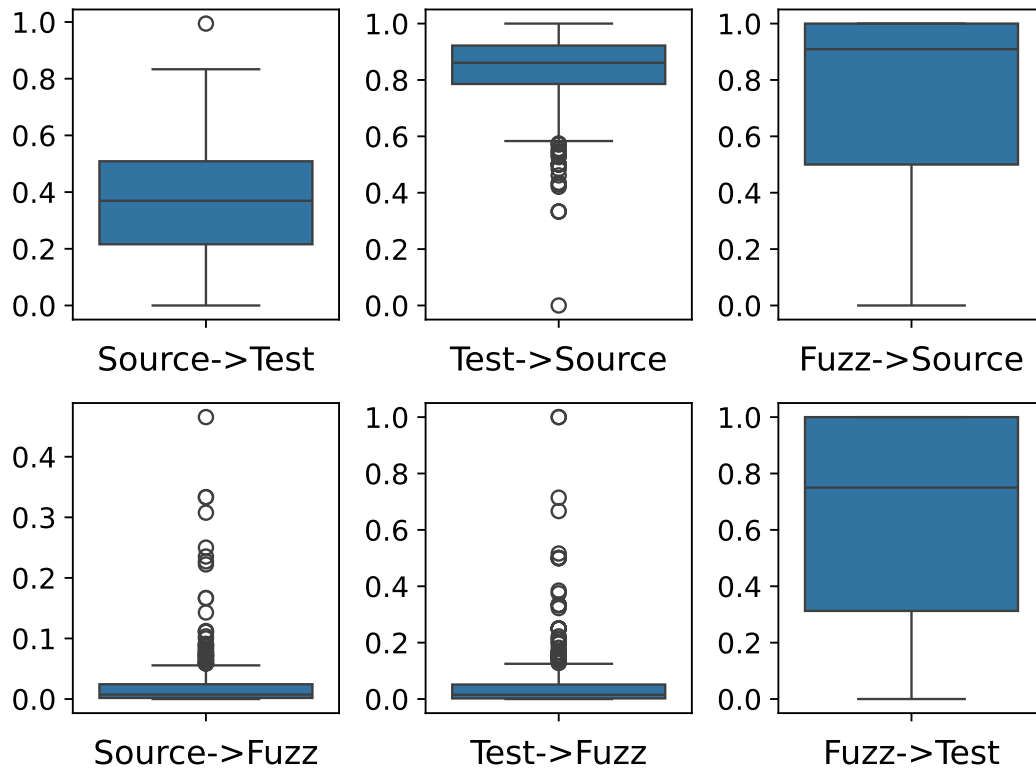


Figure 3.4: Distribution of confidence values across studied projects.

We find that source code changes are much more likely to be accompanied by non-fuzzing test code changes than fuzzing test code changes. We also find that fuzzing test code changes are committed with both source code changes and standard test code changes. Our results indicate that fuzzing development is not being done as part of regular testing activities.

6 Who Conducts Fuzz Testing in Open-Source Projects? (RQ2)

Motivation In PQ1, we found that fuzz testing is still an underutilized feature by open-source contributors which suggests that there is room to improve the security of open-source software systems. While we cannot know the exact reasons behind the low usage of fuzz testing, the testimonies of fuzzing experts in Nourry *et al.*'s 2023 survey revealed that fuzzing has a high barrier of entry [165].

In this section, we aim to find out if fuzzing is mostly conducted by projects that have fuzzing or security experts within their contributors. Moreover, we want to know if the average open-source contributor contributes to fuzz testing activities. Finding out empirically if projects distribute the fuzzing workload among contributors will also help us understand if the barrier of entry to fuzzing is still too high after introducing fuzz tests into the standard library.

Approach From the mining process described in Section 3.2, we also extracted the author of every commit. To find out if an average open-source developer is likely to work on non-fuzzing test code but not fuzzing test code, we first find out the number of developers conducting testing activities in each project. Thus, we sum up the number of unique developers that deleted, modified, or added fuzzing test code during the lifetime of a project. We then perform the same calculation for non-fuzzing test code in order to compare the involvement of open-source contributors with standard testing activities (e.g., writing unit tests) versus fuzzing activities.

To get a better understanding of the workload distribution of software testing activities in open-source projects, we analyzed the distribution of testing commits in our studied projects. Thus, we first grouped together all commits belonging to each project. Then, for each developer, we calculated the number of commits that modified non-fuzzing test functions and the number of commits that modified fuzz test functions.

Once we knew how many non-fuzz test commits and how many fuzz commits each developer contributed to a project, we sorted the developers based on their number of commits in each project. Starting from the contributors with the most commits, we then calculated the cumulative distribution of non-fuzz test commits and fuzz commits in each project. Finally, we calculated how many developers are required to achieve 80% of the fuzz testing workload in each project and then calculated the median number of developers required across all projects. We repeated the same process for non-fuzzing test code in order to compare how developers distribute the testing workload for fuzz testing versus standard testing.

Lastly, we aimed to determine if developers contributing to fuzz testing are “fuzzing experts” that focus most of their open-source contributions toward fuzzing activities. To find out, we sampled developers that had at least one fuzz commit (i.e., a commit with fuzz test

code changes). We then counted how many testing commits each developer contributed in each project. In this context, we define “testing commits” as a commit where the developer adds, delete or modifies a non-fuzzing test function or a fuzz test function. Any change in test a file that does not modify a test function is not considered as a testing commit. For each developer, we therefore calculated what percentage of testing commits are changing fuzz functions to find out if fuzzing developers are focusing specifically on fuzzing activities or not.

Results. We find that only a small number of contributors have contributed to fuzz testing activities whereas a significant portion of contributors have written non-fuzz tests. Table 3.3 shows the median number of developers across projects that participate to standard testing activities, fuzz testing activities and the median number of developers that participate to both. Our results show that almost 24% of open-source contributors (median of 32 developers per project) contributed to standard test code for our selected projects. This indicates that open-source communities view software testing (specifically non-fuzz testing) as a shared effort to achieve better code quality and security. Conversely, we find that only 1.31% (median of 2 developers per project) of developers have contributed fuzz testing code development.

These results show that large open-source projects mostly rely on a few contributors knowledgeable about fuzzing among their community to conduct fuzzing activities. Additionally, these results also indicate that most developers do not feel comfortable writing fuzz test code even though Go developers purposefully designed fuzz tests to be similar with unit tests.⁸

Table 3.3: Median number of developers participating to standard testing activites, fuzzing activities, and both types of testing activities across all projects.

	# of developers	% of developers	# of developers responsible for 80% of changes
Test	32	23.96%	2
Fuzz	2	1.31%	1
Test + Fuzz	1	0.50%	X

⁸<https://github.com/golang/go/issues/44551#issuecomment-785176317>

We find that both standard testing and fuzz testing rely on a few contributors to take on most of the testing workload. Table 3.3 shows the number of developers required to cover 80% of normal test changes and fuzz test changes. Since only a handful of developers contribute to writing fuzz test code, it is not surprising that a few developers can account for a large portion of fuzz test code changes. For standard testing, however, we previously found a 32 median number of developers that have written non-fuzz test code across projects which initially indicated that the testing workload might be shared among the contributors. However, as we can see in Table 3.3, we find that the median number of developers required to take on at least 80% of the non-fuzz testing workload is only two. From our results, we therefore find that open-source projects mostly rely on a few dedicated contributors to conduct software testing activities (both fuzz testing and standard testing).

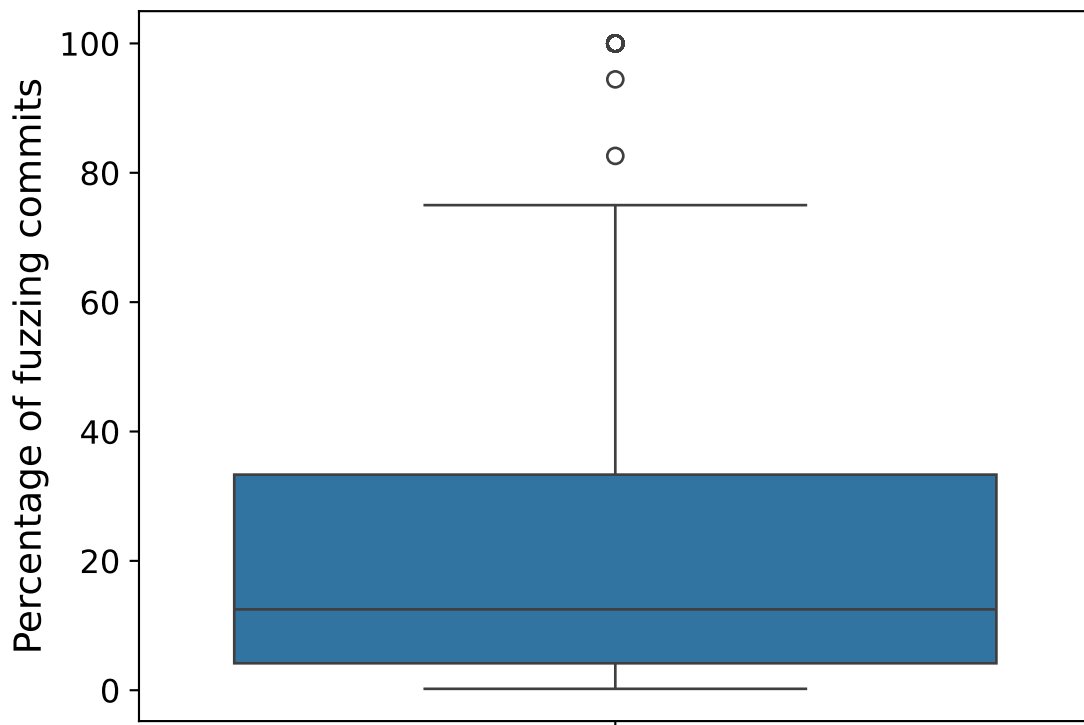


Figure 3.5: Percentage of testing commits that modify fuzzing code for developers conducting fuzzing activities.

We find that fuzzing developers primarily contribute to standard testing activities over fuzzing activities. To know if contributors contributing to fuzz testing in open-source projects are mostly dedicated to fuzzing activities, we calculated what percent-

Table 3.4: Functions and functionalities that are being fuzzed in open-source GO projects.

	Label	Examples
Data	Data manipulation	Data encoding/decoding, hashing, zipping/unzipping
	Queries	Request queries, message queue queries, db queries
	Database	Database endpoints, reads, storage
Features	Project specific features	Authentication functions, play/pause/stop functions
	Networking features	Packet fuzzing, request headers fuzzing
Internal Codebase	Generative functions	Public key generations, token creation, uuid generations, etc.
	Parsers	Request parsers, query parser, packet parsers, data parsers
	Storage	Storing to a server, into memory or auth token storage
	Setup functions	Loading configs, database setup scripts, env setup
	Utility code	Compression, Remote Procedure Calls (RPC), custom address iterators, file deletion functions, timestamp/dates.

age of testing commits (i.e., commits that change either a non-fuzz test or a fuzz test.) contain changes to fuzzing test code. Figure 3.5 shows what percentage of testing commits modify a fuzz test function. With a median percentage of 12.50% fuzz commits, we find that fuzzing developers dedicate most of their contributions towards non-fuzz testing rather than fuzz testing. These results indicate that open-source contributors conducting fuzz testing are mostly “testing experts” that use fuzzing alongside non-fuzz testing rather than “fuzzing experts” that focus solely on fuzzing.

We find that less contributors have contributed to fuzz testing activities than non-fuzz testing activities. We also find that most test code is developed by a few contributors that focus on testing. Our results suggest that fuzzing development is not conducted by fuzzing specialists but rather contributors dedicated to testing activities in general.

7 Discussion

7.1 What Projects Are Making The Most Use of Fuzz Testing?

Our results in this study have shown that fuzz testing only represents a small fraction of ongoing testing activities in OSS projects. While it is surprising how little fuzz tests are

used in practice, there is a clear complexity and resource cost gap between fuzz testing and standard testing that cannot be ignored. The case study led by Lian *et al.* [136] on the adoption of fuzzing at Huawei empirically showed that even experienced developers can experience difficulties learning fuzz testing. Computationally, Böhme *et al.*'s study on the cost of fuzzing [23] has also shown that long fuzz sessions are extremely inefficient to run. Due to these limitations, most projects might not want to have most of their testing infrastructure revolve around fuzzing.

While fuzz testing is more complex and more computationally expensive than standard testing, they still have their place within the realm of software testing. As of 2024, fuzz testing is one of the best automated approach available to test at scale and find vulnerabilities. However, due to their high cost, it is relevant to find out in what context fuzz tests are suitable and which functions warrant the high computational cost. For example, unit tests might be better suited to test simple functions while fuzz tests might be more suitable for critical features related to security or data management.

Projects revolving around high-risk fields such as avionics, finance, and networking might therefore be more comfortable with the high cost of fuzzing in exchange for better security. We therefore raise the question: **What kind of projects use fuzz testing the most as of 2024?**

Table 3.5 shows the ten projects in our dataset that had the highest number of fuzz test functions. From these top 10 projects, we find that seven of them are directly related to different types of infrastructure systems such as blockchain networks, cloud computing technologies, and networking infrastructure. In fact, we find that half of the top 10 projects using fuzz testing the most are projects developing blockchain networks where financial transactions must happen securely over the network. The other three projects not related to networking are comprised of custom libraries for software developments and a custom database driver.

For most of the projects on the top 10 list, we observe a clear pattern where projects using fuzz testing a lot are dealing with some critical feature involving data integrity or network security. A vulnerability in these kinds of systems can result in catastrophic consequences costing billions of dollars, widespread network security breaches, or large data breaches.

Interestingly, we also observe that the list of top 10 projects is quite representative of

Table 3.5: Top ten projects with the highest number of fuzz functions in our dataset.

Project URL	Fuzz functions	Project description
babylonchain/babylon	160	Crypto/Blockchain network
lightningnetwork/lnd	104	Crypto/Blockchain network
spacemeshos/go-spacemesh	94	Crypto/Blockchain network
cncf/cncf-fuzzing	73	Repository used to fuzz CNCf projects
ava-labs/avalanchego	65	Crypto/Blockchain network
luxdefi/node	59	Quantum crypto/blockchain network
thepudds/fzgen	59	Fuzz wrapper generator in Go
mt00hey31/iter	50	Go package for iterator functions
gravitational/teleport	50	Network infrastructure platform
scylladb/scylla-go-driver	47	Experimental, high performance ScyllaDB driver

the rest of our dataset of projects that use fuzzing. After performing a quick manual check on the rest of our dataset, we find that projects that fuzz are overwhelmingly comprised of blockchain related projects and network/cloud infrastructure projects. The remaining projects consist mostly of custom made libraries and database related projects.

7.2 What Are the Main Uses of Fuzzing?

From our manual investigation, we were able to notice a pattern in what types of projects use fuzzing the most. Since most of these projects deal with high-risk or critical features, we are interested to know what features these communities deem worth fuzzing. To find out, two authors manually investigated the GitHub repositories of the projects listed in Table 3.5 and inspected their fuzz tests.

Since large open-source communities usually have contribution guidelines, test function names are typically descriptive of the purpose of the test (e.g., *FuzzParamsQuery*, *FuzzEncodingDecoding*, etc.).

By inspecting the name of the fuzz test functions and the fuzzing test code, the authors were therefore able to determine the purpose of each fuzz test and to make a list of reasons why open-source communities fuzz their software systems.

Table 3.4 shows the types of functions fuzzed by open-source communities and gives examples found during the manual review process. During the manual investigation, we found that most projects tend to use fuzz testing to fuzz critical functions. That is, functions that handle core features of a system and where any bug or unusual behavior can bring down the entire system or cause serious consequences. For example, a function that rewards users with cryptocurrency coins while mining cryptocurrency would be considered a critical feature.

From Table 3.4, we can see that several of these functionalities such as data encoding/decoding or database queries are used in projects that would not be considered as “high-risk”. While it is logical to fuzz critical functions for projects in sensitive fields (i.e., financial, networking, etc.), we are also interested to know if projects not involved in such fields also fuzz their critical features.

We therefore randomly sampled 5 projects from our dataset which were not involved with blockchain technologies or did not develop critical open-source infrastructure. We performed the same process as before and investigated their usage of fuzz testing. Interestingly, we find that these projects do not fuzz their most important features as much as projects developing critical software systems. For example, the *sourcegraph/zoekt*⁹ project is a large open-source project providing source code search functionalities. In this project, we find many functions used to parse, query, and manipulate data which are key features of the system but find very few fuzz tests within their testing codebase.

7.3 Summary and Takeaway

We find that fuzz testing is mostly used by software systems that require a higher level of security than most projects. Our manual investigation reveals that projects use fuzz tests to test functions that are critical for the system to run properly.

Our qualitative analysis’s results are consistent with our quantitative results where we

⁹<https://github.com/sourcegraph/zoekt>

found that fuzz testing is not used by most projects and only represents a small fraction of test code for projects that do use them. From our qualitative analysis, it seems more likely that fuzz tests are not ignored by developers but rather only used for specific use cases where fuzz testing excels over other types of testing.

For developers, we highly encourage every project to use fuzz tests to detect vulnerabilities since the risk of spreading vulnerabilities is always present due to the interdependent nature of the open-source ecosystem [138, 198]. For those not familiar with fuzz testing, we recommend fuzzing functions such as the ones listed in Table 3.4.

For researchers, our manual investigation can serve as a starting point to explore the possibility of automatically detecting important functions that should be fuzzed given a codebase. If such a detection approach could be designed and paired with automatic test generation tools, we could see a significant increase in number of developers that conduct fuzz testing.

8 Threats To Validity

Internal threats to validity concern factors internal to our study that could affect or bias our results. In this study, we used the top 20% largest open-source projects on GitHub to conduct our analysis. This could bias our results by not being representative of smaller projects. Using this process, however, we reduce the risk of analyzing toy projects and increase the chance that our studied projects are conducting real open-source software development.

External threats to validity concern the ability to generalize the result of this study. In this study, we only analyze the evolution of fuzz testing in OSS projects. Our results may not be generalizable to other languages that do not offer native support for fuzzing testing. Because Go is one of the first language to support native fuzz tests, we, however, believe that our study is representative of the current use of fuzz testing by the average developer (i.e., developers that are not fuzzing experts like the ones working on OSS-Fuzz).

Construct threats to validity concerns the ability concern the relationship between theory and observation. To find and mine Go projects, we used the GitHub search tool published by Dabicet *al.* and specified “Go” as the main language in the search field. While the projects returned by the tool all have Go as their main language, the proportion of Go language over

the entire codebase may differ (e.g., some projects might be 100% in Go while others only have 45% of their code in Go). It is possible that the use of fuzz testing may differ between projects developed purely in Go versus projects using Go for specific parts of their codebase.

9 Chapter Summary

In this study, we investigated the usage of native Go fuzzing in open-source projects. Our results show that very few contributors develop fuzz tests in open-source projects. We also find that developers conducting fuzzing tend to target specific functions rather than use fuzzing as a general testing strategy. Our results also revealed that less than 4% of our studied projects conduct fuzzing activities. Due to the risk of spreading vulnerabilities throughout the open-source ecosystem, we highly encourage projects to start adding some fuzz tests to their testing codebase.

For developers, we recommend using fuzz tests in combination with unit tests to get the best balance between computation cost and code robustness. For developers looking to start fuzzing their software, we propose a list of functionalities that are strong candidates for fuzz testing.

For researchers, we provide new empirical evidence of how open-source communities conduct software testing activities. Our results provides researchers and software engineers a better understanding of the factors limiting the adoption of fuzz testing. From our results, we hope that better tools and methodologies that help developers adopt fuzz testing can be developed by the software engineering community.

Chapter 4

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

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. However, recently disclosed issues such as the heartbleed vulnerability [29] and the log4j vulnerability [8] 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 [7]: 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 [106]. In particular, one type of automated testing techniques called fuzz testing (or fuzzing) [134] 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 [27, 30, 173, 179, 219], 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 [169], 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.* [22], 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 ap-

proach [191] 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.

We summarize the main contributions of this chapter 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 prevalence 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-Fuzz¹ 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 500⁴ 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 fuzzers⁶ [214]. This process led to 2,562 issues.

A manual inspection of dozens of issues revealed two main problems within our dataset: First, many issues were merely reporting bugs found by fuzzers; second, the issues in the

¹<https://github.com/google/oss-fuzz>

²<https://llvm.org/docs/LibFuzzer.html>

³<https://aflplus.plus/>

⁴The number is dynamic and keeps increasing.

⁵<https://google.github.io/oss-fuzz/>

⁶While the fullname of “afl” is “American Fuzzy Lop”, in practice, developers tend to use its abbreviated version.

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 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.

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 [41] which guarantees a confidence level of 95% with a significance interval (margin of error) of $\pm 3\%$. 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.

⁷<https://github.com/google/gopacket/issues/983>

We followed the card sorting procedure [191] 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 4.1).

OSS-Fuzz Label Home Check Issues My Labels Charts Log Out

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

Figure 4.1: Web App used for inspecting issues

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 essence 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.

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).

Creating the First Taxonomy

The three authors who participated in the annotating process worked together to conduct the card sorting [191] 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 [110]. 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.

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”) [31]. 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.

Survey Participants

Given the very specific topic investigated in the study, 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% [130]. 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 4.1: 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 4.1 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:

- ⓘ: 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

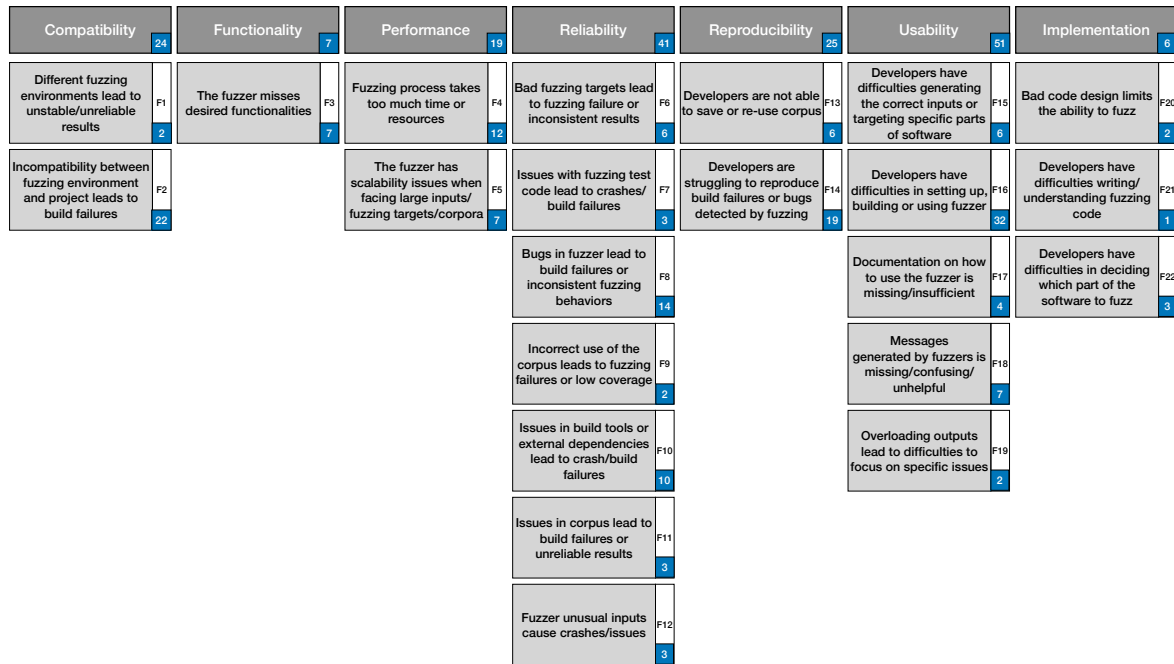


Figure 4.2: Taxonomy of challenges from the fuzzing process itself.

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 4.2 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 [110]. 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.

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.

❶ For example, in issue #3347 of the scapy project [182], 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.

❶ As shown in issue #19557 from the bitcoin GitHub repository [17], 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.

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.

📌 In issue #1675 of the official OSS-Fuzz GitHub repository [81], 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.

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 [28] 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 [18], 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.

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.

❗ 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 [77].

👤 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.

❗ 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 [91].

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.

❶ 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 [76].

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.

❶ 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 [88].

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.

❶ 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 [93].

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.

❶ For example, a developer describes issues he has had with inputs crashing his fuzz target in issue #1622 of the syzkaller repository [97]. 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.

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.

❗ For example, in issue #52292 of the Go project [66], 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 [84] and #6147 [90] 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.

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 [167] 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 [36], 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 [53], 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 [65], 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 [96], 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.

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 [19], 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.

❶ In issue #1398 of the scapy project [181], 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.

❶ In issue #6201 of the Solidity project [52], 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 4.3), 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	Documentation	Functionality	Performance	Permission	Reliability	Usability
28	8	5	4	14	24	15
Updates on the side of OSS-Fuzz lead to project crash/build failures	Documentation about how to use/integrate OSS-Fuzz is missing/unclear	OSS-Fuzz misses desired functionalities/features	OSS-Fuzz does not allocate enough resources to (a) project(s) for fuzzing	Developers do not have access to project status/fuzzer logs/bug reports/test case on OSS-Fuzz	Connection issues on OSS-Fuzz lead to project crash/build failures	Disorganized/missing information on issue trackers makes it difficult to locate the problem
O1	O3	O5	O6	O8	O10	O15
7	6	5	2	13	2	2
Compatibility of environment or dependencies between project and OSS-Fuzz lead to project crash/build failures	OSS-Fuzz does not provide clear guidance on how to reproduce the bug		OSS-fuzz cannot handle small fuzz targets	Oss-fuzz configurations lead to undesired bug public release	Key resources unavailable on OSS-Fuzz lead to project crash/build failures	Information in fuzzing reports can be missing, unclear, misleading or even wrong
O2	O4		O7	O9	O11	O16
21	2		2	1	2	10
					OSS-Fuzz bugs lead to build/fuzzing process failures	Oss-fuzz configurations lead to confusing/unreported/hard-to-reproduce issues
					O12	O17
					1	3
					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	

Figure 4.3: 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.

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.

❶ In issue #153 of the OSS-Fuzz [72] 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 be the most frequent provider challenge during our manual analysis, accounting for close to 21% of all the relevant issues related to external fuzzing providers.

❷ In issue #823 of the OSS-Fuzz repository [78], a project was using the latest features offered by a new compiler version which were not yet supported by the OSS-Fuzz environment.

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 [68]. 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.

❸ In issue #2094 of the OSS-Fuzz repository [82], 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.

- ❶ In issue #1344 [80], a developer asked for guidance from OSS-Fuzz developers due to not being able to reproduce the bug using only OSS-Fuzz’s documentation.

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.

- ❷ *For example, in issue #2558 [83], 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.*

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.

- ❸ In issue #3014 [85], 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.

❶ In issue #1333 of the OSS-Fuzz repository [79], 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.

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.

❷ In issue #243 [73] 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 [67]. 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.

❸ In issue #244 [74], 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.

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.

❶ In issue #7581 [95], 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.

❶ For example, in issue #7198 [94], 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.

❶ In issue #3082 [86], 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.

❶ In issue 5526 [71], a developer reported that multiple issues that were previously reproducible and subsequently fixed were 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.

- ❗ While discussing how OSS-Fuzz should handle timeouts and OOM errors in issue #3432 [89], a developer mentioned that he had instances where the CIFuzz jobs timed out due to infinite loops without reporting any problem.

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 [32] 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.

- ❗ In issue #273 [75], 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.

- ❗ For example, in issue #6885 [92], 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 [87].

i In issue #6587 [54], a developer discovered that some bugs found by the fuzzer were ignored and not treated as errors.

3.3 Survey Result Analysis

Frequency of Fuzzing Challenge Occurrences in the Survey

Figure 4.4 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 4.1), 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.

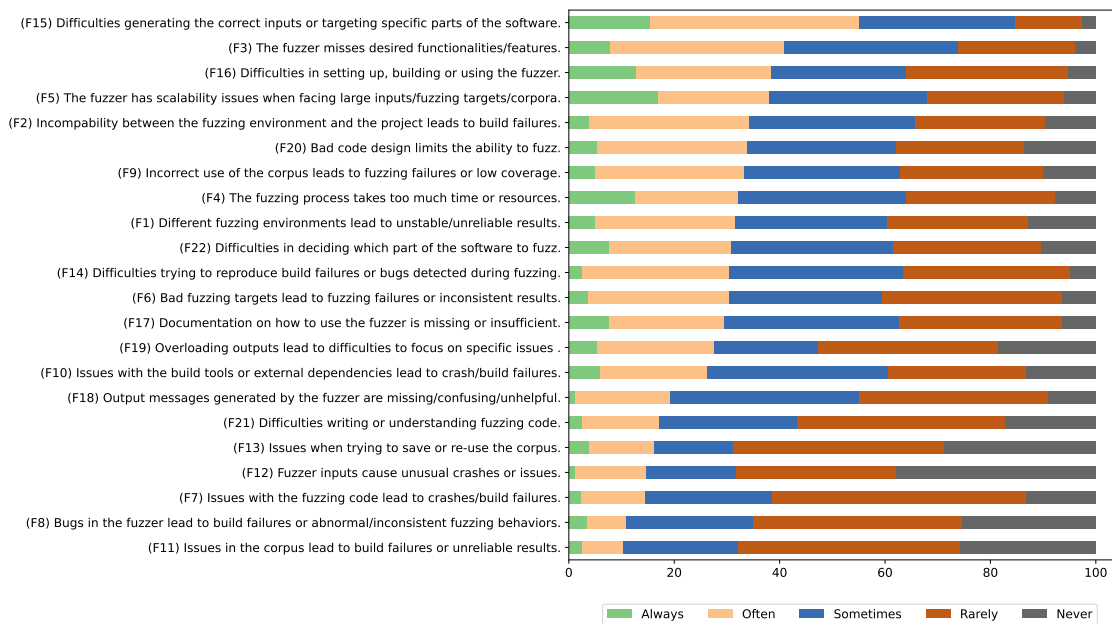


Figure 4.4: Frequency at which the respondents encounter each challenge in percentage.

Frequency Comparison Between Survey and Manual Analysis

From our manual analysis (Figure 4.2), difficulties in setting up, building or using fuzzer (F16), incompatibility between fuzzing 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 4.4). 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.

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 [166]. 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.

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 “fuzz” 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 [169], 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 [98] or microcontrollers [133] 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.* [150] 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.* [164] 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.* [171] 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 [26]. 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 [49, 151]. Zhou *et al.* [217] 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.* [186] 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.* [11] 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.* [204] developed **FreeFuzz**, which aims at fuzzing deep learning libraries via mining open source software repositories. Gu *et al.* [100] 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.* [216].

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.* [212] 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.* [46] 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 [24]. 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.* [126] 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.* [117] 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 usability issues (F15 in our taxonomy) to a certain extent. Liang *et al.* [136] 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 [22], 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.* [211] 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 [46] 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 [24] 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 Chapter Summary

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.* [136].

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.

Chapter 5

Automating Software Quality

Maintenance for Better Efficiency and Reduced Complexity

1 Introduction

With the increasing size and complexity of modern software systems, it is becoming increasingly harder to avoid bugs and vulnerabilities from being introduced into a codebase. In the context of open source projects, a single security flaw can have worldwide impacts as we observed with recent vulnerabilities such as the Heartbleed vulnerability [109] and the Log4Shell vulnerability [42]. To minimize the risks of such events happening again, development communities increasingly turn to fuzzing to actively look for vulnerabilities in their codebase. Fuzzing works by sending large amounts of unexpected inputs to a software system to trigger bad behaviors and find security flaws such that they can be fixed before a malicious actor makes use of them.

The open-source software ecosystem, specifically, has seen a significant increase in fuzzing activities in recent years. For example, as a result of the Heartbleed vulnerability [109], Google launched the OSS-Fuzz [169] service with the aim of making the entire open-source ecosystem more secure. The OSS-Fuzz service provides continuous fuzzing to open-source software systems and notifies a project's developers when vulnerabilities are found by OSS-

Fuzz’ fuzzers. On top of external fuzzing services, other open source communities have made efforts to spread the adoption of fuzzing. For example, the Go development team decided to include fuzzing as a feature within the Go programming language itself to let developers easily write fuzzing tests and make their software more secure [58].

While fuzzing helps improve the overall security of the ecosystem, it also requires open-source communities to take on new challenges related to the process of fuzzing. These challenges cover the entire lifecycle of fuzzing starting from choosing what to fuzz and how to write fuzz targets, to setting up, using and building fuzzers, and finally maintaining fuzzing infrastructure over time. Several studies have been conducted by researchers to know more about the current state of fuzzing and to find the current limitations of existing fuzzing tools and processes [16,135,145,220]. In 2020, a Shonan meeting [20] was held to discuss the overall state of fuzzing, to reflect on current fuzzing methodologies and processes, as well as to discuss current challenges and possible future work in the field. While the field of fuzzing has many publications on fuzzing tools and techniques to improve fuzzing coverage [129,206,207], the lack of open source empirical data has limited the ability of researchers to investigate fuzzing practices and methodologies so far.

The recent advent of OSS-Fuzz, however, enables researchers to have access to high quality empirical data to conduct empirical studies [46,116] and empirically find out more about current fuzzing practices and challenges. In 2023, Nourry *et al.* [165] mined and used OSS-Fuzz data to empirically find out what challenges fuzzing developers are facing and to get fuzzing developers’ opinions on the current limitations of fuzzing. Based on manual analysis and testimonies of developers involved with fuzzing activities in open-source software systems, Nourry *et al.* found that the most common type of issues in the context of fuzzing are build-related issues. While this finding sheds more light on why fuzzing activities are challenging, it is still unclear in what ways build management is causing issues for fuzzing practitioners. Specifically, the survey based study conducted by Nourry *et al.* does not reveal how prevalent build related issues are when conducting fuzzing activities nor does it reveal what are the root causes underlying these build issues.

Using publicly available OSS-Fuzz data, it is now possible to empirically investigate some of the fuzzing experts’ concerns highlighted in Nourry *et al.*’s survey starting with fuzzing

build failures. OSS-Fuzz currently supports most state of the art fuzzers (AFL++, libfuzzer, HongFuzz, and Centipede) which allows any important open-source project using C/C++, Rust, Go, Python or Java to use the OSS-Fuzz service. Due to the wide range of fuzzers and programming language supported, fuzzing activities conducted through the OSS-Fuzz service are representative of most use cases of fuzz testing. Consequently, the build issues found during the OSS-Fuzz build process and the resulting failing build logs are also representative of build failures encountered by developers conducting their own fuzzing activities. Similar to how other studies have investigated build failures in other contexts (CI, Docker, Software development, etc.) [50, 59, 177, 183, 208], we set out to learn more about build failures in the context of fuzzing using openly available fuzzer build log data from OSS-Fuzz. More specifically, we first conduct a quantitative analysis to quantify the prevalence of build failures in the context of fuzzing. Through this quantitative analysis we aim to find out the following:

(RQ1) How often do fuzzing builds fail?

We analyze the distribution of build failures across projects participating in OSS-Fuzz to understand how common are build issues in the context of fuzzing. We calculate a median percentage of build failure of 5% across all projects indicating that OSS-Fuzz projects carefully manage their fuzzing builds. We also find that a few projects do not actively maintain their OSS-Fuzz fuzzing build.

(RQ2) How long does it take to fix failing fuzzing builds? *We investigate how long it takes for a failing fuzzing build to be fixed in order to better understand the time cost of fixing fuzzing build failures and also to get some insights as to whether or not open source communities quickly address fuzzing-related issues or not. We find that 80% of fuzzing build failures are fixed within a day of the first failure. We also find that 73.85% of failing builds have no subsequent failing builds.*

Following the quantitative analysis, we then conduct a qualitative analysis to get a better understanding of the factors causing build failures in the context of fuzzing. Through this qualitative analysis we aim to answer the following:

(RQ3) What are the root causes of fuzzing build failures?

Using a manual analysis approach, we empirically find the root causes of build failure in

677 failing fuzzing builds logs. We then define a taxonomy of build failure root causes pertaining to fuzzing builds. Using this taxonomy, we aim to make diagnosing fuzzing build failure easier and quicker for developers maintaining fuzzing activities in open source projects. Our manual investigation reveals 25 distinct root causes of fuzzing build failures. We find that multiple build failures are not specific to fuzzing but rather related to using build systems in general and that multiple fuzzing builds fail due to circumstances outside of the developers' control.

We summarize the main contribution of this paper as follows:

1. We conduct a quantitative analysis to find out the prevalence of build failures in the context of fuzzing.
2. We conduct a qualitative analysis to find out why fuzzing builds fail and propose a clear taxonomy of build failure root causes for fuzzing builds.
3. We conduct an experiment to automatically classify fuzzing build failures using a machine learning model.
4. We provide a manually labeled dataset of 677 failed fuzzing build logs along with the identified root cause of failure.

2 Background

2.1 Fuzzing Process

Fuzzing is an automated software technique that consists of sending large amounts of inputs to a software system in order to trigger unexpected or bad behaviors. While fuzzing can be used for a variety of use cases [105, 152, 156, 175], the current main applications so far have been to find vulnerabilities via penetration testing or software testing. To implement fuzzing, a developer must first define the fuzz target(s) that will be receiving the fuzzer inputs. In the context of penetration testing, a fuzz target could be any software that expects an input such as an API expecting a request [12, 218]. For software testing, a fuzz target can be a piece of source code that expects a specific input to be executed [58]. For software systems whose

codebase contains a lot of interdependency in the source code or whose software architecture follows a monolithic design, the process of creating a fuzz target can be very challenging for developers. The main reason is that for each fuzzing session, each fuzz target must be able to be compiled and executed on its own. Developers must therefore find ways to extract parts of a codebase such that the source code can be compiled and executed on its own without needing to compile the rest of the software system.

After defining and instrumenting (if necessary) the fuzz target(s), a developer must set up the fuzzing environment by defining the necessary environment variables and installing the required dependencies. Then, the developer must configure a fuzzer based on the type of fuzzing that will be conducted (network fuzzing, UI fuzzing, binary fuzzing, etc). Because fuzzers are often configured for a specific type of software and can only be executed after the environment is perfectly configured for the fuzzing use case, it is not uncommon for developers to reconfigure their fuzzers as a result of external changes that affected the environment (i.e., a dependency or the compiler updated its version number). The complexity of fuzzing tools and the domain knowledge required to properly configure a fuzzer and its environment can therefore be a significant challenge for developers that are not experts in fuzzing.

Once the environment and the fuzzer are both configured, a developer needs to download and fetch the remaining resources required for his/her fuzzing use case such as a corpus. In the context of fuzzing, a corpus is a set of test inputs used as a baseline to generate new inputs for further testing. When a new input is found to crash a target system or increase the coverage, developers will often add this new input to their corpus so that it can be reused in future runs. After aggregating all necessary resources and setting up the environment, the developer can finally trigger the build process which will download all dependencies, compile the project and its fuzz target(s), and compile the fuzzing tool. If any of these previous task fails, the build process fails and the developer must then find out what caused the build failure, fix the issue, and run the build process again.

If a build process successfully completes, the fuzzer is then executed and starts continuously sending inputs to the fuzz target(s). If the developer provided a corpus to the fuzzer, the inputs contained in the corpus will be used to try to crash the target system and also to generate new inputs via mutations. These mutations can be done using a variety of strate-

gies. For example, the popular AFL++ fuzzer [56] supports a wide range of input mutation strategies such as changing the length of an input, flipping random bits, substituting parts of an input and even merging multiple inputs into one just to name a few.

While fuzzing, the responses of the target software system are continuously monitored so that any unexpected behavior such as crashing or wrongly providing elevated permissions is recorded and subsequently made into a bug report. To improve fuzzing over time, all inputs that trigger vulnerabilities and cause crashes are added to the corpus so that future runs can use them to generate new inputs. Additionally, developers can also generate coverage reports and use coverage as a benchmark to increase the amount of source code triggered during fuzzing over time.

2.2 OSS-Fuzz Infrastructure

OSS-Fuzz is a free continuous fuzzing service provided by Google for open-source projects that are considered critical for the open source software ecosystem. Using this service, open-source projects and the open-source ecosystem as a whole can benefit from continuous fuzzing to check for vulnerabilities without having to bear the financial cost themselves. Although there is no official documentation to know how often or how long each project is fuzzed every day, OSS-Fuzz supports continuous integration to fuzz every pull request of a project.¹ If developers opt to use continuous integration, they are then able to set how long their project should be fuzzed using the “fuzz-seconds” argument in their configuration (default of 600 seconds of fuzzing up to a maximum of almost 6 hours). Since OSS-Fuzz developers are able to assign more or less CPU time to each project individually via internal weights that are not visible to the public,² to the best of our knowledge there is currently no way to know how much fuzzing is conducted on each project.

While OSS-Fuzz is useful to automate the fuzzing process (i.e., fuzz continuously), using an external service to fuzz a software system also adds its own set of challenges. For instance, using an external provider might make it more difficult to locate the root cause of a build failure than if a developer is always using his/her own local environment to fuzz their project. Additionally, integrating a project to OSS-Fuzz, managing the fuzzing build over time on an

¹<https://google.github.io/oss-fuzz/getting-started/continuous-integration/>

²<https://github.com/google/oss-fuzz/issues/3014>

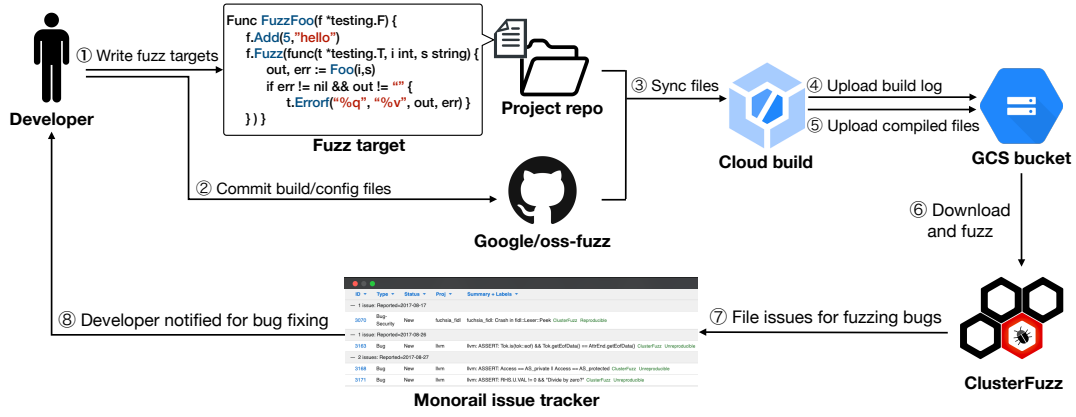


Figure 5.1: OSS-Fuzz architecture

external service, and using OSS-Fuzz’ bug issue tracker are all additional challenges incurred by using an external provider rather than fuzzing locally. Figure 5.1 shows an overview of the OSS-Fuzz architecture and describes the steps to use OSS-Fuzz.

Integrating a project to OSS-Fuzz. As described on OSS-Fuzz’ website [108], when an open-source project is accepted into the OSS-Fuzz service by the OSS-Fuzz team, one of the project’s developers or a maintainer of the project must first create fuzz targets for the target project then integrate the fuzz targets into the project’s build and testing infrastructure (①).

Next, the developer must commit the build files and configuration files (i.e., `project.yaml`, `Dockerfile`, `build.sh`) required from the OSS-Fuzz service into the official OSS-Fuzz GitHub repository (②).³ The first file required is the *project.yaml* file. It includes general information such as a link to the project’s repository and the contact information of the maintainer but also which fuzzing engine (i.e., libfuzzer, AFL++, etc.) and which sanitizers (i.e., MSan, ASan, etc.) to use.

The second file required is a *Dockerfile* which allows OSS-Fuzz to reproduce the docker environment in which a project’s fuzzers will be built and the fuzz target(s) will be fuzzed. The third and last required file is the *build.sh* file which contains configuration commands to download or set up the required dependencies such as the environment variables, the symlinks, the corpus/corpora, and the pip packages to install.

OSS-Fuzz’ process flow. Once all the build files and configuration files are uploaded to

³<https://github.com/google/oss-fuzz>

the OSS-Fuzz repository, the OSS-Fuzz service will use a cloud builder⁴ to build the project using the provided build/config files (③). At the end of the build process, the resulting build log will be uploaded to a Google Cloud Storage (GCS) bucket (④) dedicated to OSS-Fuzz. Additionally, the metadata files stored in the OSS-Fuzz GCS bucket will be updated with the newly created log's information. After uploading the build log and updating the metadata files, the cloud builder will then upload the compiled project's files and fuzz target(s) to the GCS bucket (⑤). Finally, OSS-Fuzz' ClusterFuzz infrastructure⁵ will download the fuzz targets, start fuzzing the project and continuously monitor the system for crashes or unusual behaviors (⑥). If a vulnerability is found during fuzzing (⑦), an issue is automatically created on the official OSS-Fuzz issue tracker [70] and the project's developers/maintainers are notified of the vulnerability (⑧).

In the case of failed builds, we have found through a manual process several different ways that project developers are notified of build failures. One common way that OSS-Fuzz developers notify the project developers is by tagging them in a GitHub issue on the official OSS-Fuzz repository (e.g., issue#5558). We have also found cases where a project developer will notice the failure on his/her own and make an issue in either the OSS-Fuzz GitHub repository or the project's GitHub repository (e.g., issue#40714). In other cases, the official bug tracker for OSS-Fuzz will also notify project developers automatically of build failures happening for their project (e.g., issue#23673). It is also highly likely that a project's maintainer gets automatically notified of a OSS-Fuzz build failure via the ClusterFuzz panel (the management panel for all OSS-Fuzz processes related to a project). Since each project's panel can only be accessed by the official maintainer, we however cannot confirm if the management panel does indeed notify developers.

OSS-Fuzz projects build logs. The logs stored in the GCS bucket shown in step (④) of Figure 5.1 are the logs we used in this study to investigate what causes fuzzing build failures. These are the build logs generated from the cloud builder as a result of compiling the target project, the fuzz targets and the fuzzer. If the cloud builder build process is unsuccessful, the project's maintainer and members of the OSS-Fuzz team must look at the

⁴<https://cloud.google.com/build/docs/cloud-builders>

⁵<https://google.github.io/clusterfuzz/>

resulting build log to figure out what caused the failure and who needs to fix the issue. A build failure will normally be fixed either by one of the project's developers/maintainers or a member of the OSS-Fuzz team based on whether the error happened on the project's side or from an OSS-Fuzz related issue.

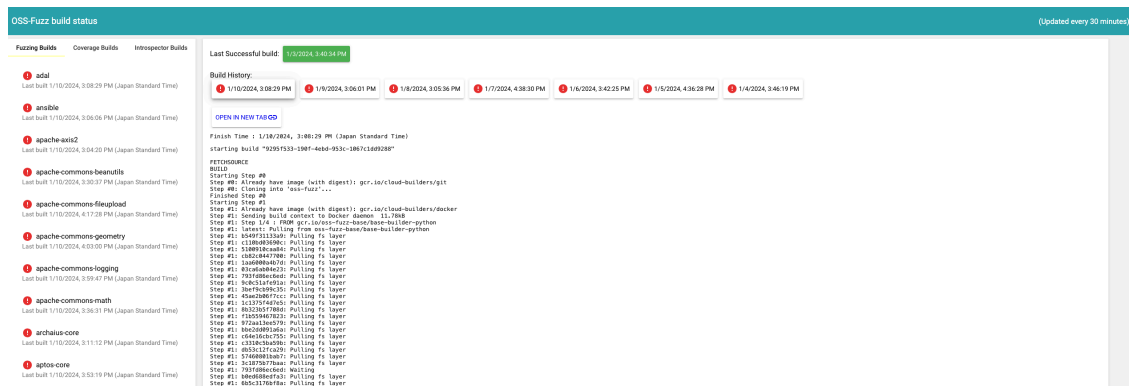


Figure 5.2: OSS-Fuzz public panel to OSS-Fuzz projects' fuzzing builds status

As shown in Figure 5.2, a dashboard [170] showing projects being fuzzed by OSS-Fuzz is publicly available online. Using this dashboard, anyone can view the last 7 fuzzing build logs of each project and the date of the last passing build. As described in the official documentation ⁶, each project is built once a day by default but developers can increase the frequency up to four times a day using the *builds_per_day* argument in the configuration file. To find fuzzing builds older than 7 days, we had to find the metadata files contained in the OSS-Fuzz build logs GCS bucket (step (④) in Figure 5.1) which contains generic information about all build logs generated from the cloud builder build process. To access the metadata files, the Google storage API ⁷ must be used to mine the OSS-Fuzz bucket. By parsing these metadata files, we were able to extract the links to older fuzzing builds and the date at which they were executed. Once we had a list of all OSS-Fuzz projects’ fuzzing builds, we were finally able to mine the entire history of fuzzing builds logs from OSS-Fuzz’ GCS bucket.

⁶https://google.github.io/oss-fuzz/getting-started/new-project-guide/#build_frequency

⁷https://cloud.google.com/storage/docs/json_api

3 Quantitative Analysis

In this section, we first conduct a quantitative analysis to reveal how prevalent are build failures in the context of fuzzing and how much time developers spend on fixing their fuzzing builds.

3.1 Quantitative dataset

Using the metadata files stored in OSS-Fuzz’ GCS bucket described in Section 2.2, we were able to extract and mine the links (URLs) to past build logs. Using these URLs we then started mining every build log contained in the metadata files. From the metadata files, we also extracted the unique identifier (hash) and the creation time of each build log so that we could establish the full historical timeline of fuzzing builds for each project. At the end of the log mining process, the total number of fuzzing build logs amounted to 1,223,075 logs dating from March 2017 to September 2022 and spanned over 748 projects. We then parsed each build log to extract the data necessary for our study namely the name of the project being built, the project’s revision hash at the time of the build, and the URL to the project’s repository.

As shown in Table 5.1, we then filtered out logs where the build crashed so early that the target project had not even been cloned yet. For these cases, because the crash happened so early, the name and URL of the project being built were not included in the crashing log, which made it impossible for us to know which project a log belonged to. Because we need this information to conduct the quantitative analysis, we filtered out these instant crash cases which brought down our number of logs to 974,431 build logs.

After manually checking the dataset, we found that some of the projects’ URLs extracted from the build logs were pointing to repositories where fuzzers were being developed or repositories storing corpora used by a fuzzer to fuzz a target. In the case of corpus repositories, we found that some of them were used purely for the purpose of storing a corpus and did not contain any source code.⁸ To ensure that we analyzed build logs that reflect “standard” fuzzing use cases (i.e., finding software vulnerabilities), we decided to remove from our dataset all build logs from which we extracted a URL pointing to a repository where a fuzzing corpus

⁸<https://github.com/guidovranken/cryptofuzz-corpora>

was stored. Additionally, since we are interested in real fuzzing use cases where projects are fuzzed to find vulnerabilities and make the open source ecosystem safer, we also decided to remove build logs from which we extracted a URL pointing to a repository where a fuzzer was being developed.

Table 5.1: Number of logs remaining after each step of the filtering process

Filter step	Logs removed	Number of logs remaining
Total Number of logs	0	1,222,075
Name and URL shown in the logs	247,644	974,431
Is a corpus or fuzzer repository	5,509	968,922

To find corpus repositories and fuzzer repositories, we first isolated all build logs where the extracted project URL contained the substrings “fuzz” or “corpus”. To ensure that we were not discarding valid repositories, one of the authors manually opened each URL and examined the repositories on their respective version control systems. As a result of this manual examination, we found and filtered out 43 URLs belonging to repositories used for storing corpora or developing a fuzzer. This next layer of filtering brought down our number of build logs from 974,431 logs to 968,922 logs.

3.2 (RQ1) How often do fuzzing builds fail?

Motivation. We first do a preliminary analysis to find out how often builds fail in the context of fuzzing. As shown in previous studies investigating build failures [183, 208], knowing the prevalence of build failure can help us quantify how serious of a problem is build management when conducting fuzzing activities. Additionally, finding out that fuzzing builds often fail could reveal underlying issues such as fuzzers needing better compatibility with build systems.

Approach. To get an overview of how common build failures are, we first calculate the percentage of total build failures between 2017 and 2022. Since each project might have different levels of dedication to fuzzing activities, some projects might fail more often than

others and introduce bias in the total percentage of build failure. We therefore aggregate all build logs for each project, sum up the number of build failures and calculate the fail percentage for each project individually. We then aggregate the number of failing build logs for all projects and calculate the overall mean and median number of fuzzing build failure across all projects. To calculate the average and median percentage of build failure per project, we repeat the same process and calculate the percentage of build failure for each project individually then calculate the mean and median percentage of build failures across all projects. Finally, we investigate projects that have a high ratio of build failures to understand why some projects fail to maintain their fuzzing build while other projects do not.

Results. From the filtered dataset described in Section 3.1, we found 877,682 builds out of 968,922 to be passing builds and 91,240 to be failing builds. From March 2017 to September 2022, we therefore find that only 9.41% of OSS-Fuzz fuzzing builds failed. We then calculate the number of build failures and the percentage of build failure in each project to calculate the median and average number of failure across all projects. As shown in Table 5.2, we find that the median number of failed builds per project is 197.5 and the mean number of failed builds is 680.4 per project. To calculate the mean and median percentage of build failure across all projects, we calculate the build failure percentage in each project individually then calculate the mean and median using every projects' fail percentages. We find that the mean build failure percentage across all projects to be 12.36% and the median build failure percentage to be 4.76%. This indicates that some projects' fuzzing builds are failing much more often than other projects' fuzzing builds.

Table 5.2: Statistical data of how many builds fail in each project (in absolute number of builds) and across all projects (in percentage).

	Mean	Median
Per project (in builds)	680.4 builds	197.5 builds
Across all projects (percentage)	12.36%	4.76%

Investigating the few projects (12 projects) that have a very high percentage of build failures (>70%), we find a mix of currently active projects, two inactive projects, and also

google internal projects that interns added to OSS-Fuzz. In this context, we considered two projects as inactive. First, the libra project⁹ because Facebook officially shutdown the project and sold it to a private organization which did not make further contributions to the open-source repository. At the time of data collection, a lapse of two months had passed without any contribution to the repository with no further contribution made since then. A second project we considered inactive was an intern project where interns were tasked to set up fuzzing for one of Google's project. At the time of data collection, that project had no commits in over three years.

For the inactive project and the google internal projects, we find that these projects are not actively being fuzzed by OSS-Fuzz as of December 2023. For active and inactive projects, we found that two projects had stopped maintaining their OSS-Fuzz build at the time of data collection. The libra project had stopped maintaining their OSS-Fuzz build since April of 2020 and the ClickHouse project¹⁰ had stopped maintaining their OSS-Fuzz build since December of 2021. This indicates to us that in the case of active projects such as ClickHouse, although development was still currently taking place, the developers chose not to dedicate resources to fuzzing at that time. From the OSS-Fuzz dashboard¹¹, we find that ClickHouse eventually fixed their fuzzing build (latest passing build dates back to May 13th 2023 as of December 2023) but stopped maintaining their fuzzing build once again. As for the libra project, the project remains abandoned as of December 2023.

We find a median percentage of build failure of 4.76% across all projects indicating that most projects participating to OSS-Fuzz carefully manage their fuzzing builds. We also find two instances of projects that do not actively maintain their fuzzing infrastructure. OSS projects deciding not to maintain their fuzzing build indicates that fuzzing build management might be challenging and require significant efforts from open-source contributors.

⁹<https://github.com/libra/libra>

¹⁰<https://github.com/ClickHouse/ClickHouse>

¹¹<https://oss-fuzz-build-logs.storage.googleapis.com/index.html>

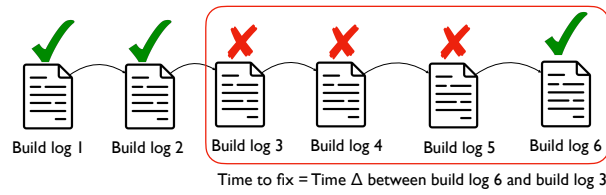


Figure 5.3: Approach to calculate the time to fix a build failure in a failing build sequence.

3.3 (RQ2) How long does it take to fix failing fuzzing builds?

Motivation. In their study investigating the cost of build failures, McIntosh *et al.* [148] found that build maintenance can add an overhead of 27% on source code development and 44% for test development. Additionally, the testimonies of fuzzing experts surveyed by Nourry *et al.* [165], reveal that fuzzing developers experience difficulties fixing their fuzzing build. To get a better understanding of the cost of build maintenance in the context of fuzzing, we decided to investigate how long it takes for developers to fix a failing fuzzing build. Knowing how much time is required to fix a build gives us some insight about whether or not open source communities are willing to dedicate human resources to fuzzing activities over source code development. Additionally, the time required to fix fuzzing builds can give us some indications as to how complex and time consuming fuzzing maintenance is for developers conducting fuzzing activities in open source projects.

Approach. To measure the time to fix failing fuzzing builds, we first looked at isolated failing builds (i.e., a single failing build with no subsequent build failures) and calculated the time difference between the build failure and the following passing build. Next, we investigated sequences of build failures (at least 1 subsequent build failure after the initial failing build). For these cases, we calculated the time difference between the first failing build (build log 3 in Figure 5.3) and the first passing build (build log 6 in Figure 5.3) following the initial build failure as shown in Figure 5.3. If no passing builds could be found after the start of a build failure sequence, then that indicated to us that the project’s fuzzing build was still failing at the time of the data collection. For such cases, the entire sequence of failing builds was discarded since we could not calculate the build fix time without a passing build.

Since projects participating to OSS-Fuzz have different level of activity from their contributors and OSS-Fuzz allows up to 4 builds per day, very active projects with contributors

dedicated to fuzzing activities might have the luxury of running multiple builds on OSS-Fuzz within a single day and therefore have very short build fix times. On the other hand, smaller projects which do not have contributors dedicated to fuzzing activities might only be able to fix their fuzzing build once a day or every few days. To normalize build fixing time across projects regardless of how often a project builds its fuzzers on OSS-Fuzz, we calculated the number of subsequent builds required to fix a project's failing fuzzing build. For this analysis, we also discarded sequences of builds which do not have a subsequent fixing build.

Results. We found that the overwhelming majority of projects keep close attention to their fuzzing build. Table 5.3 shows our results when assessing how long it takes for builds to get fixed. When measuring the fixing time in days, we find that close to 80% of fuzzing builds are fixed within a day of the first failure and that 95% of fuzzing builds are fixed within the first 2 days of the first failure. To get a more in depth view of the time required to fix failing builds, we also measured the time to fix in hours rather than days. Figure 5.4 shows the time required to fix failing fuzzing builds in hours. As shown in Table 5.3, we find that 40% of fuzzing builds are fixed within 12 hours of failing and 75% of fuzzing builds are fixed within the first 24 hours. From our results, we also find that 95% of initial failing builds tend to be fixed by the second day.

Next, we counted the number of builds between the initial failing build and the next passing build as a proxy metric to calculate the time required to fix a failing build. Figure 5.5 shows the distribution for the number of subsequent failing builds before the next passing build. In other words, a value of 4 in the x-axis means that the fuzzing build failed 4 additional times after the initial build failure then passed on the 5th subsequent build. The y-axis value shows the proportion of all builds which share the same number of subsequent failing builds.

Our results show that 73.85% of failing builds have no subsequent failing builds and 93% of builds have either a single or no subsequent failing builds. When looking at each project individually, we find that once a build fails, the median number of subsequent failing builds is 0 and the mean number of subsequent failing builds is 4.70. This indicates that most projects are keeping up with their fuzzing activities and that only a few projects are delaying or struggling with their fuzzing build repair. When calculating the proportion

Table 5.3: (Left) Cumulative percentage of initial build failures fixed within the specified time interval.

(Right) Percentage of initial build failures with their corresponding subsequent number of build failures.

Time interval (in hours)			Subsequent failing build(s)		
$\leq 12\text{h}$	$\leq 24\text{h}$	$\leq 48\text{h}$	0	≤ 1	>10
40%	75%	95%	73.85%	93%	2.12%

of failing builds with more than 10 subsequent failing builds, we find that in only 2.12% cases builds fail more than 10 times before it is fixed. Figure 5.5 also reveals that some projects participating to OSS-Fuzz have abandoned their fuzzing activities for a significant amount of time (up to several years) before starting to manage their fuzzing build again. For these extreme cases where the build seems to not be maintained for long periods of time, we find that 0.32% of initial build failures have over 100 subsequent failures before the build is fixed. Since projects accepted into OSS-Fuzz are important projects for the open source software ecosystem, this extended time without fuzzing represents missed opportunities to find vulnerabilities that can have a significant impact on the ecosystem.

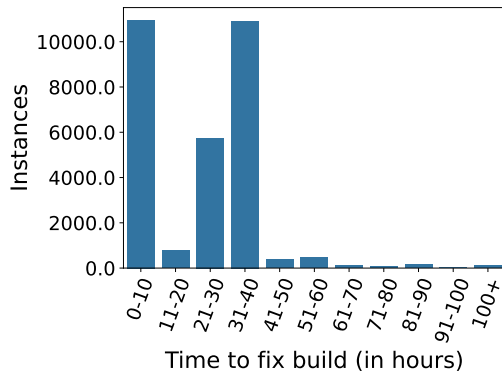


Figure 5.4: Time to fix a fuzzing build failure in hours

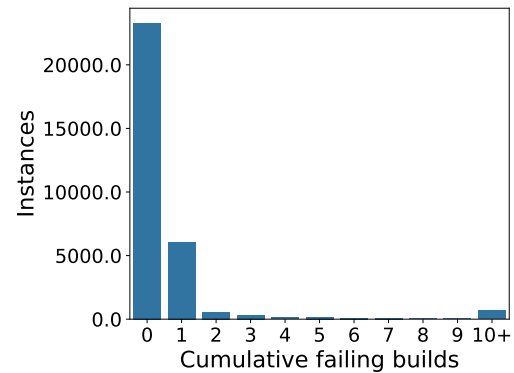


Figure 5.5: Number of subsequent failing builds

We find that 93% of projects keep up with their fuzzing workload and fix their failing fuzzing builds within one or two build cycles. In fact, we find that only 2.12% of initial build failures require 10 or more subsequent builds before the failure is fixed and 0.32% of initial build failures require 100 or more builds to be fixed. For these more extreme cases, these periods without fuzzing leave important open source projects potentially vulnerable to security issues that could have been uncovered by OSS-Fuzz.

4 Qualitative Analysis

In Section 3, we found that some projects have a significant amount of build failures and that a small number of projects stop fuzzing altogether. While the quantitative analysis showed that some projects are having difficulties maintaining their fuzzing build, we still do not understand why that is the case. To understand why some projects are struggling with fuzzing build management, we decided to conduct a qualitative analysis to find out the root causes of fuzzing build failures.

A qualitative analysis brings several contributions to the field of fuzzing which currently has few empirical studies. Finding out the root causes of build failures establishes some of the ground work required to start improving fuzzing methodologies and fuzzing development in general. Defining a clear taxonomy of root causes for fuzzing build failures provides a clear list of areas that can be improved by researchers and developers working on developing automation tools for fuzzing.

Throughout the following section, we use the following symbol  to provide the URL to a sample build log that failed due to the described root cause.

4.1 Qualitative dataset

Using the filtered dataset used for the quantitative analysis, we further extracted a smaller subset to conduct a manual analysis of failing build logs. To do so, we first used the log creation date extracted from the metadata files to establish the full fuzzing build log history for each project in our dataset. We then isolated all pairs of pass/fail log pairs throughout the studied projects' build histories. In other words, we paired together all instances where a build was passing and the following build was failing. The reason we took this approach

is that we hypothesized that the first failing build (in a sequence of failing builds) contains the root cause of the failure. Additionally, by sampling logs corresponding to initial build failures (no previous failing build), we reduced the odds of analyzing logs with multiple unrelated errors compounded over time (i.e., a new unrelated error affects an already failing build causing). Since build logs are complex to parse and understand, avoiding build logs with compounded errors was important to reduce the complexity of the log data and help annotators figure out the true root cause of a failing log. From the 968,922 logs dataset described in Section 3.1, we further removed all logs which were not part of a pass/fail log pair. This brought the number of valid logs for the qualitative analysis down to 66,254 build logs spanning over 677 projects.

Because some projects have a much longer fuzzing history than others on OSS-Fuzz, we decided not to sample random build failures from our entire dataset to avoid biases towards projects that have been fuzzing with OSS-Fuzz for a longer period. Additionally, to avoid biases at the project level where a single unfixed issue makes up a large percentage of fuzzing build failures within a project, we decided to randomly sample only one build failure from each of the studied projects. By randomly sampling the build failures, we were also able to get a fair representation of what causes fuzzing build failures regardless of the kind of project or the project’s fuzzing experience.

The dataset for the manual analysis therefore consisted of one randomly sampled pair of pass/fail build logs for each of the 677 remaining projects (1,354 total logs). Additionally, the manual annotators were also provided with a diff file showing the difference between the content of the passing build and the content of the failing build to easily find the differences between a passing build log and its subsequent failing build log.

4.2 Qualitative analysis methodology

To find the root cause of build failures, three annotators took part in the manual investigation of the randomly sampled failing build logs. Based on previous qualitative studies’ approaches [131, 147], we designed the manual labeling methodology as follows.

Establish the base taxonomy. The three annotators first conducted a trial run by indi-

vidually summarizing the cause of build failures in 100 failing build logs. The three authors then got together to 1) fix disagreements for cases where the cause of failure differed between the authors and 2) derive classification labels from the root causes of build failures observed during the trial run. Step 1) and Step 2) were repeated until a base taxonomy was established from the 100 logs used for the trial run.

Validate the initial labels and derive a shared understanding. After agreeing on an initial set of labels, 200 failing logs (100 logs from the trial run and 100 new ones) were used to verify the validity of the labels. After relabeling the 100 logs used during the trial run and labeling 100 new logs, the three authors once again got together to compare their classifications and update the taxonomy. During this part, the authors focused on deriving a shared understanding/definition of each label.

Additionally, the three authors also agreed on the criteria to create new labels during that period. A new label was to be created only if no existing label could describe the cause of failure found in a build log. Additionally, the new label needed to be as descriptive and specific of the root cause of failure as possible (i.e., small granularity). Each author had to ensure that newly created labels described the cause of failure rather than the symptoms of the failure (i.e., describe the root cause of failure rather than the error messages shown in the build logs).

Once all authors were in agreement with the definition of each label and the criteria to create new labels, the remaining sampled logs were split into three sets. Each set was manually labeled by two authors and conflicts were reviewed by all three authors together. The full dataset of logs was split into three sets so that the manual labeling could be done in three separate iterations. By proceeding iteratively, the annotators were able to share their new labels at the end of each iteration and agree on the updated taxonomy together.

Label all build logs. For each set of logs, the authors first separately assigned a root cause to each failing build log within a set. For cases where the error messages of a build log were not clear enough as to what caused the build failures or the failure patterns were too complex to easily assign a label, the three authors discussed in details at the end of an iteration which labels should be used based on their understanding of the failure. Additionally, the annotators manually searched through the GitHub repositories of participating projects and

the OSS-Fuzz GitHub issue tracker to find issues, threads, or pull requests discussing build failures for cases where the cause of a build failure could not be understood from the build log alone. When not enough information was provided from a build log's error messages and no additional information could be found online, the authors labeled the build log as "Not enough information".

In the case where multiple patterns of failures were found, the authors assigned the label of whichever failure pattern first appeared in the build log. For example, while labeling the authors encountered a situation where a failing build log contained an error message saying that the command used to extract or unzip a corpus could not be recognized or failed. Then, later in the log another error message would state that the corpus could not be found or was broken. In such situations, the authors would consider the first failure pattern related to the command failing as the root cause of the issue which caused the following error message stating that the corpus could not be found.

After each iteration, the authors compared their labeling and reached an agreement for cases where the classification differed. Between each iteration the three authors also updated their taxonomy whenever a new root cause of failure was found. At the end of the labeling process, the three authors got together and merged similar labels in order to keep the taxonomy consistent in terms of granularity (specific vs generic labels) and to make it easy and natural to understand. At the end of the labeling process, we calculated Cohen's Kappa [37] and obtain a 0.78 inter agreement coefficient between the labelers. In this context, a value of 0.78 indicates that the reviewers labeling was in agreement for most cases.

4.3 (RQ3) What are the root causes of fuzzing build failures?

Table 5.4 shows the resulting taxonomy from the labeling and merging process described in Section 4.2. In total, 11 generic categories were derived from grouping together similar root causes. From these 11 categories, we further breakdown our taxonomy into 25 unique root causes (RC1-RC25) of fuzzing build failures. Our manual labeling process reveals that corpus related issues cause the most fuzzing build failures, followed by failure to download external resources and compiler issues.

Table 5.4: Root causes of fuzzing build failures observed during the manual analysis of failing fuzzing build logs. The number in parentheses () shows how many instances the annotators found during the manual labeling process.

Category	Root Causes	Fine-Grained Prev. Study [140]	Coarse-Grained Prev. Study [177]
Environment issue (114)	<i>RC1</i> : Compiler issues (65)	Others	crash
	<i>RC2</i> : Coverage file and directory issues (26)		
	<i>RC3</i> : Project environment issues (10)		crash
	<i>RC4</i> : Network issues (7)	Server connection error	crash
	<i>RC5</i> : Hardware issues (5)	Memory issue	crash
	<i>RC6</i> : Permission issues (1)	Execution permission error	
Corpus related issues (99)	<i>RC7</i> : Corpus related issues (99)		
Issues downloading external resources (97)	<i>RC8</i> : Issues downloading external resources (97)		
Project dependency issues (58)	<i>RC9</i> : Project dependency issues (58)	Dependency resolution	dependency
Build and configuration issues (58)	<i>RC10</i> : Project configuration and build file issues (28)	Parse	buildconfig
	<i>RC11</i> : Coverage build configuration and file issues (25)	Parse	buildconfig
	<i>RC12</i> : Fuzzer build script issues (5)		
Project source code related issues (56)	<i>RC13</i> : Source code related project compilation errors (39)	Compiler error	compile
	<i>RC14</i> : Missing source code files (17)	Missing file	
Command and argument related issues (48)	<i>RC15</i> : Command and argument related issues (48)		
Runtime issues while fuzzing (43)	<i>RC16</i> : Runtime issues while fuzzing (43)		
Not enough information (43)	<i>RC17</i> : Not enough information (43)		unknown
Fuzz target issues (47)	<i>RC18</i> : Sanitizer errors (24)	Other run-time error	
	<i>RC19</i> : Broken fuzz target (21)		
	<i>RC20</i> : Missing fuzz target (2)		
Miscellaneous (14)	<i>RC21</i> : Input causes unusual fuzzer crashes or behaviors (7)		
	<i>RC22</i> : Failing test cases (3)	Test run-time error	testfailure
	<i>RC23</i> : Missing OSS-Fuzz scripts (2)		
	<i>RC24</i> : Unusual crash from the target binary (1)	External executable error	
	<i>RC25</i> : Regression in the fuzzer causes build crash (1)		

Environment issues. In this category, we find build failures where the root cause of failure was related to the environment in which the build was executed. This includes the software environment, the hardware used by the physical device, and the network environment. The most common root cause of failure related to environment issues we found was related to compiler issues (9.60% of all build failures in our qualitative dataset). These failures usually happened as a result of a compiler related change (e.g., version update) causing many fuzzing builds to break at once. In the case of compiler breaks, we found different scenarios for the breakage would stem from different sources. For example, OSS-Fuzz developers updating the compiler version in their environment (e.g., issue#6978) could be the cause of the breakage. Other times, the root cause came from an external source such as compiler developers updating their codebase and breaking OSS-Fuzz builds as a result of the changes (issue#6957). Due to the complexity of the build logs that failed as a result of a compiler issue, we were only able to figure the true root cause as a result of manually investigating the OSS-Fuzz GitHub repository and finding issues discussing these crashes (issue#6957, issue#6978).

The environment issue category also includes “coverage file and directory issues” which refer to cases where a coverage build is executed (i.e., code coverage is generated during the build process) but files due to issues unique to being a coverage build. For example, coverage build have several requirements to execute properly such as having the right directory structure. In many cases, we found that the required coverage directory was either not in the right location or missing altogether.

Compiler related failure sample build log:  log-b8e66607-6bae-4ae0-b23d-94f884eacdfa.txt

Corpus related issues. This root cause was the most common root cause of build failure found by the annotators while doing the manual analysis (14.62% of our dataset). We find in this category build failures that were caused as a result of not providing a valid corpus or where the cloud builder could not locate the corpus (either because none was provided or the provided path was incorrect).

Corpus related failure sample:  log-349a5b80-51e3-4852-ae84-cee5718acc40

Issue downloading external resources. This root cause of failure was the second most

common root cause we observed while manually labeling build failures (14.33% of all failures). Build failures in this category were most often the result of the build process failing to download external resources required for the fuzzing build. These cases usually happened as a result of a faulty URL (i.e., expired URL, typo in the URL, no resources found at the URL provided, invalid URL) which caused the build to crash and stop.

Faulty URL sample build log: : log-7029ee20-e728-41d2-a22e-e8e6ae38da9c

Project dependency issues. This root cause includes cases where dependency issues were found while compiling the target project and crashed the build as a result. This includes cases such as using the wrong dependency version or missing required dependencies entirely. Dependency related error sample log : log-bd799639-70c2-41ec-a62e-d7ef98434929

Build and configuration issues. In this category we find root causes of build failures related to the various build scripts and configuration files that the cloud builder use to execute the build process. In most cases, one or multiple of the build/configuration files would contain an error such as a typo or an undefined variable. As for “RC11: Coverage build configuration and file issues”, this root cause includes cases where the configuration failure was specific to coverage builds. We found that configuration errors were much more common in project build scripts (4.14% of manually labeled failures) than fuzzer build scripts (0.74% of manually labeled failures).

Example log where a variable in the *build.sh* file was not defined: : log-06f68129-5be9-43d8-9e47-619b3ecbc598

Project source code related issues. This category of root causes includes build failures that happened as a result of errors related to the source code of the target project. This includes errors in the source code itself (i.e., compilation errors) and source code files missing (i.e., missing file, typo in path, etc.) during the build process. The most common root cause in this category “Source code related project compilation errors” represented 5.76% of build failures in the qualitative dataset.

Example log of a project having errors in its source code : log-e4abab45-b863-4827-b829-

6ba5f2360fb3

Command and argument related issues. This root cause includes cases where a wrong command or wrong command arguments were provided during the build and caused the entire build to crash as a result. The errors included in this category vary from typos in arguments to missing arguments when executing build scripts or commands. For example, in the sample below, the build log states that the *test_all* command was not found. Thanks to a manual investigation of the OSS-Fuzz GitHub, the authors found out that the OSS-Fuzz developers had made a mistake in the provided arguments and forgot to write a file extension (issue #4781¹², fixing PR #4783¹³). Command and argument related issues accounted for 7.09% of the manually labeled build failures.

Command and argument failure sample log 🐞: log-33248e12-1c84-4c02-9852-005847481744

Runtime issue while fuzzing. In some cases (typically during coverage builds), a small fuzz session is conducted during the build process. This root cause covers cases where a runtime issue will cause the small fuzz session to fail unexpectedly and consequently crash the whole build process. This root cause accounted for 6.20% of the fuzzing build failures we manually labeled.

Sample log for runtime issue while fuzzing: 🐞: log-0ee126ba-48aa-4f12-8452-569f37d6e4c6

Not enough information. This label includes cases where the annotators could not determine the root cause with certainty. These cases were often a result of a build log not providing sufficient information to diagnose the root cause of failure from the build log alone. In other cases, the build logs had gigantic error sections out of which the authors could not extract the true cause of failure.

Sample log labeled “Not enough information” 🐞: log-0ea5baf2-321e-477c-9c7a-0b2628a47070

Fuzz target issues. In this category, we find root causes of build failures related to the fuzz

¹²<https://github.com/google/oss-fuzz/issues/4781>

¹³<https://github.com/google/oss-fuzz/pull/4783>

targets. In most cases these failures happened as a result of sanitizer related errors. When configuring their fuzzing builds, some projects choose to add sanitizers which are software tools that look for bugs at runtime (i.e., AddressSanitizers, MemorySanitizers, Undefined-BehaviorSanitizer, etc.). To check for bugs in the fuzz targets, some projects conduct small fuzz sessions during the build process. Based on the severity of the bug found, a sanitizer can trigger either an error or a warning. If a sanitizer triggers an error, the build will automatically fail and stop. Cases labeled as “Sanitizer errors” only includes cases where a sanitizer found an error that caused the build to stop. For cases where a sanitizer triggered a WARNING but no ERROR in a failing build, we did not label these logs as sanitizer errors. These sanitizer errors accounted for 3.55% of the failures we manually labeled.

For cases where issues with the fuzz targets crashed the build process for reasons unrelated to sanitizers, we labeled them as “Broken fuzz target” (3.11% of our qualitative dataset). A fuzz target can be considered broken for a variety of reasons including having errors in its source code or not being instrumentated properly. We also found two cases where the build process crashed as a result of not finding the fuzz targets.

Sample log where the AddressSanitizer finds an error 🐛: log-f6d8e847-1494-4b1a-9b15-bf0299333385

Miscellaneous. In this category, we find a variety of root causes that did not fit into any of the other categories. Several of the root causes in this category happened as a result of unexpected crashes and behaviors and are therefore either flaky in nature or very domain specific to the project.

For example, the most common root cause in the “Miscellaneous” category is the “input causes unusual fuzzer crashes or behaviors” which happens when an input is sent to the fuzzer during a fuzz session for a coverage build. While all failing logs in that root cause suggest that the crash is due to an issue with the input, we observed different symptoms in each case. Some logs stated that the first input failed (AFL requires the first input to be valid) while other logs stated that the input(s) caused the tests to stall. Since we do not have access to the exact input that caused the crashes, we classified all cases under the same “input causes unusual fuzzer crash or behaviors” root cause.

In the “Miscellaneous” category we also find cases where unit tests would fail during the build process or OSS-Fuzz scripts required during the build process were missing. Finally, we also found another unique case where the target binary crashed before any input could be sent to it (RC24). This case differed from RC16 because the small fuzzing session executed during the build process crashed before even starting to send inputs unlike cases labeled as “Runtime issues while fuzzing” which crashed during the fuzzing session.

Sample log where an unexpected input crashes the fuzzer : log-7d43d249-3182-4402-94ee-d5cd148d0b5b

4.4 Comparison with previous studies

There are several ways to study build failures in the context of software development. Some studies conduct fine-grained investigation of build failures by focusing on a specific cause or context for build failures. For example, Seo *et al.* [183] conducted a study that focuses on build failures caused by source code related issues to understand which source code errors are most responsible for breaking builds. Conversely, studies such as ours or such as the ones led by Rauschet *et al.* [177] and Louet *et al.* [140] analyze build failures on a more general level without focusing on any specific context or root cause. Alternatively, other studies such as Kerzaziet *et al.*’s [119] have also tried to understand build failures by investigating the circumstances that lead to build failures.

Our qualitative results show a lot of similarities with taxonomies proposed in previous studies on build failures. This is mostly due to build failures having common sources of failures regardless of the kind of project or the context in which the build is executed. For example, categories such as “Environment issues” and “Configuration issues” are recurring categories in studies investigating build failures [140, 177, 183].

Our study however differs due to the additional layers of complexities brought by fuzzers which introduce new possible causes of failures not present in previous work. For example, while the taxonomy in the coarse-grained study led by Rauschet *et al.* [177] makes mention of environment issues, their environment issues make no mention of hardware issues or compatibility issues with the compiler. All issues present in non-fuzzing builds can also be found in fuzzing builds. However, fuzzing builds have more possible source of failures related to

using a corpus or configuring/running the fuzzing tool itself. Our taxonomy therefore offers an extended look of possible cause of failures over taxonomies proposed in previous work.

To highlight how our work relate to previous work (and extends it), we compared the result of our analysis with the results proposed in two previous studies on build failures. Specifically, we compare our taxonomy with the coarse-grained taxonomy proposed by Rausch *et al.*'s in their 2017 study [177] and the fine-grained taxonomy proposed by Lou *et al.* in their 2020 study [140]. Based on the descriptions of the taxonomies in the original papers, we tried to match the description of their labels with our own to show all common categories between the taxonomies. As shown in Table 5.4, our taxonomy cover most of their taxonomy while also offerering several new root causes specific to fuzzing.

4.5 Implications

Our manual investigation reveals that fuzzing build failures happen for a wide variety of reasons spanning across multiple domains. For example, the annotators found errors on the project side, errors in the fuzzer itself, errors external to both the fuzzer and the project (e.g., network errors), and errors on the provider side (OSS-Fuzz). Because fuzzing build failures can happen in many different locations, in some cases the cause of the crash may not be due to a mistake from the fuzzing developer. **In fact, our results show that several root causes (e.g., RC1, RC5, RC13, RC23, etc.) of build failures are caused by factors outside of the developers' control which makes the diagnostic process and the fixing process difficult for practitioners.** Compiler related issues (RC1) were a common example of this phenomenon where compiler developers (e.g., rust developers, llvm developers, etc.) would push an update and dozens of fuzzing builds would break on OSS-Fuzz without any intervention or modification by the fuzzing developers or the OSS-Fuzz developers.

Our results demonstrate that diagnosing fuzzing build failures can pose a real challenge for developers conducting fuzzing activities. The challenge of diagnosing a fuzzing build failure is further evidenced by the 41 build logs assigned to RC14 ("Not enough information") where three separate annotators could not confidently determine what was the cause of failure in a failing build log. While this could be due to a lack of project specific

knowledge, the “Not enough information” root cause ranked 8th in terms of frequency out of 25 identified possible root causes. The high frequency of “Not enough information” indicates that it is not uncommon for fuzzing build failures to require a deeper investigation to diagnose the actual root cause of failure.

From Table 5.4, **we find that many root causes responsible for fuzzing build failures are not specific to fuzzing but are common causes of failures for build systems in general** (e.g., dependency issues, network issues, source code issues, command and argument issues, build configuration issues, etc.). As McIntosh *et al.* [148] found in their study on build maintenance, build maintenance can cause significant development overhead to software development activities. In order to minimize the overhead cost of build management on their fuzzing activities, fuzzing practitioners should not only possess fuzzing expertise but also have a good understanding of build systems and experience with build management.

From our manual investigation, we find 25 unique root causes of fuzzing build failures. We find that corpus related issues (99 instances), issues downloading external resources (97 instances), and compiler issues (65 instances) were the most common types of root causes for fuzzing build failures in our sampled data. Our taxonomy reveals that fuzzing build failures can come from many different sources spanning over both the software and hardware. Our taxonomy also shows that fuzzing builds fail not only due to issues specific to fuzzing but also due to issues related to using build systems in general. Additionally, we find multiple cases where fuzzing builds failed due to reasons outside of the fuzzing developers’ control.

5 Discussion

5.1 Why do projects abandon their fuzzing build and how do we prevent it from happening?

The quantitative analysis in Section 3 revealed that a few projects periodically stop their fuzzing activities for significant periods of time. Since OSS-Fuzz is taking on the processing workload and the financial cost of running this processing power instead of the developers,

it is not clear why a project would choose to abandon fuzzing activities. It is therefore relevant to question if the suspension of fuzzing is due to a conscious choice from the projects' developers or if there are other limiting factors that forced these projects to abandon their fuzzing activities. To answer this question, we manually examined the two projects (libra and ClickHouse) that were found to have abandoned their fuzzing activities in Section 3.2.

In the case of the libra project, a discussion ¹⁴ on the OSS-Fuzz GitHub repository between the project's fuzzing maintainers and an OSS-Fuzz developer reveals that the project developers did not always make time to address some of the issues related to fuzzing. After the libra project rebranded itself to "Diem" instead of libra and the fuzzing maintainer was not active anymore, the project was later disabled on OSS-Fuzz. From the official OSS-Fuzz repository, we can also confirm that the project is still disabled as of 2024 by looking at the *project.yaml* file which contains the "disabled" parameter set to true. While looking at *project.yaml* files of other projects to understand why projects get disabled, we found comments mentioning that projects were disabled either because they were test projects, or because the project was being archived.(e.g., opencensus-go project, bazel test project)

For the ClickHouse project, our manual investigation reveals that the initial integration of the project in May of 2020 ¹⁵ caused build failures and the project was therefore disabled until one of the maintainers fixed the issues. A pull request ¹⁶ on the OSS-Fuzz GitHub dating from August 2021 reveals that it took over a year before a contributor decided to get involved again with fuzzing activities for the ClickHouse project. Based on our manual investigation and the fact that the ClickHouse fuzzing build has been failing for several months as of December 2023, we hypothesize that the ClickHouse project faced the same situation as the libra project where the developers did not dedicate time to work on fuzzing activities.

In both the libra and the ClickHouse projects, our manual investigation reveals that large open source projects are very much dependent on having one or multiple of their contributors willing to take on fuzzing activities for the entire project by themselves. Consequently, the lack of such contributor(s) means that these projects cannot be fuzzed and a vulnerability could affect the tens of thousands of dependent projects and users. As highlighted by the

¹⁴<https://github.com/google/oss-fuzz/pull/6560#event-5494529937>

¹⁵<https://github.com/google/oss-fuzz/pull/3800>

¹⁶<https://github.com/google/oss-fuzz/pull/6244>

fuzzing experts surveyed in Nourry et al.’s previous study [165], fuzzing currently has a high barrier of entry and is a very complex field to get into as someone with no prior expertise. While many of the larger open source projects have the luxury of having fuzzing experts among their contributors, we hypothesize that lowering the barrier of entry for fuzzing by simplifying fuzzers and establishing standard fuzzing methodologies could increase the number of open source contributors willing to maintain fuzzing activities for open source projects.

Until the barrier of entry for fuzzing is lowered, however, we hypothesize that automated tools could be one of the main ways that a smaller open source project can sustain fuzzing activities while having limited fuzzing expertise within its contributors.

5.2 Improving fuzzing using other fields’ breakthroughs.

Although we currently have no way to automatically identify who fixes failing builds between OSS-Fuzz developers, project developers or dependent projects’ developers, our taxonomy can give a general idea of where build breakages come from. Assuming that the people responsible for breaking are also the people in charge of fixing it, we can also derive a general idea of who fixes failing builds using our taxonomy. For example, any build failure that happens due to an “environment issue” (other than “RC3: Project environment issue”) or a “command and argument” related issue is likely to be both caused due to an issue on OSS-Fuzz’ size and therefore fixed by OSS-Fuzz developers. Similarly, any root cause related to an issue with a project’s configuration or its source code is likely due to a mistake on the project developers’ side and likely to be fixed by project developers.

One of the main challenges of diagnosing and fixing failing fuzzing builds seems to be that fuzzing build failures can come from a wide variety of sources and contexts which often do not have a relationship between each other (project environment, fuzzing environment, docker environment, platform dependent tools, project dependencies, fuzzer dependencies, external resources to download, fuzzer corpus etc.). While this issue is not unique to fuzzing, several of the root causes found during the manual analysis seem to be caused by factors unrelated to fuzzing. From Table 5.4 in Section 4, we find several root causes of build failures in fuzzing builds that are present in non-fuzzing builds such as source code related errors, failing test cases and dependency issues.

Since many of these root causes occur across a variety of development activities (software development, fuzzing, testing, web development, etc.), several studies on automated build repair have already developed automated tools to address them [101, 142, 161, 215]. It is likely that solutions proposed in previous studies could also be applied in the context of fuzzing. For example, fuzzing builds which happen due to source code errors could benefit from automatic build repair strategies and patch generation tools applied in the context of software development [139, 197]. Dependency and configuration issues could also benefit from automatic dependency versioning repairs [203]. Using the knowledge acquired from other fields' studies, fuzzing developers could potentially improve their fuzzing methodologies without much efforts. Additionally, using tools developed by other fields could possibly reduce the amount of fuzzing build failures or significantly reduce the amount of efforts required to fix failing builds by automating parts of the build fixing process.

5.3 Modeling build failures

As discussed in Section 5.1, open source projects are currently dependent on having fuzzing experts among their contributors. To alleviate some of the dependence of open source projects on open source fuzzing developers, we therefore turn to automation tools to see if open source projects could automate some of the workload incurred by maintaining fuzzing activities. In a previous study investigating log-related issues in Java systems, Hassani et al. [102] demonstrated the possibility of automatically detecting log-related issues in software systems using basic logic rules.

Building upon this idea, we set out to find if it is also possible to automate the diagnosis of build failures in the context of fuzzing. While fuzzing build failures can be quite complex and come from a wide variety of sources as discussed in Section 5.2, we hypothesize that automation can be developed to alleviate some of the fuzzing build management workload. Using the build logs manually labeled for the qualitative analysis, we conduct a basic experiment to test the feasibility of automating the classification of fuzzing build failures by training a machine learning model to recognize common failure patterns in failing build logs.

Preprocessing the logs. We first perform basic text preprocessing by removing extra punctuation and symbols. Since fuzzing builds logs generated by OSS-Fuzz have clear build

steps and a build stops when an error or a crash occurs, we trimmed each build log to only keep the content of the last build step executed during each build. In other words, we only kept the section of each build log that contained the crash inducing error. We did this because the size of the build logs were too large to embed and because we wanted to minimize the amount of text data not relevant to the crash for the model training process.

Embedding the log data. To embed our textual log data, we decided to use OpenAI’s *text-embedding-ada-002* [168]. The main reason behind this choice was that this embedding model allowed up to 8,192 tokens per input. Since many of our build logs remained large even after trimming all sections that were not relevant to the build failure, models that only allow short inputs such as the BERT model [43] (512 tokens) were less suitable for our use case. Since the build process stops when a crash happens, it is more likely that the data indicating the root cause of a build failure is located towards the end of the error section than the start. Therefore, to fit the 8,192 tokens limitation of the embedding model, we tokenized the crashing section of our build logs and kept only the last 8,192 tokens of each log. In one case, the embedding model kept crashing due to one of the failing build log. We therefore removed this data point for the purpose of the classification and used the remaining 676 manually labeled logs.

Visualizing the embedded logs. After embedding our build logs, we used a t-SNE plot to get an overview of our data. A t-SNE plot is a plot that allows us to get a simplified view of our data when dealing with high dimensional and complex data. While a t-SNE plot’s axes do not have a directly interpretable meaning, the formation of small clusters/groupings (as shown in Figure 5.6) indicate to us that some of the data points share similarities between each other. In this case, since we are working with embedded build log data, a t-SNE plot allows us to eas-

Build log embedding distribution using t-SNE

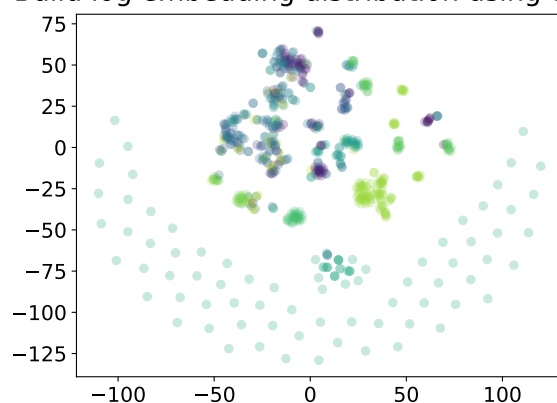


Figure 5.6: Embedded log distribution using t-SNE

ily find out if there are subpopulations or clusters of build logs similar to each other within our dataset. The color of each data point indicates what label (root cause) was assigned to that specific failing log. When several build logs share similar text content, they will appear closer together in the t-SNE plot. Using the color of the points on the plot, we can therefore tell at a glance if logs that share similar text content also share the same root cause or not. Lastly, the groupings in the t-SNE plot also indicate to us that there could be textual patterns specific to each root cause that a model could be trained to recognize and classify.

Modeling and classification. For this experiment, we wanted to see if a model could learn to recognize frequent types of build failure root causes, namely: corpus related issues (RC7), issue downloading external resources (RC8), compiler issues (RC1), project dependency issues (RC9), and finally command and argument related issues (RC12). All failing build logs not belonging to one of the top five most common root causes were labeled as “Other” for the purpose of this experiment. Using OpenAI’s embeddings as an independent variable to find similarities between failed builds logs, we then tried to classify every failed build logs in our manually labeled dataset into one of the 6 possible categories mentioned above.

Table 5.5: Random Forest Classifier 10-Fold cross validation results

Categories	Precision	Recall	F1-score	Testing instances	Correct predictions
(RC7) Corpus related issues	1.000	0.970	0.985	99	96
(RC8) Issue downloading external resources	0.979	0.969	0.974	97	94
(RC1) Compiler issues	0.671	0.815	0.736	65	53
(RC9) Project Dependency issues	0.333	0.017	0.033	58	1
(RC12) Command and argument related issues	0.952	0.833	0.889	48	40
Other	0.808	0.942	0.870	309	291
Total (Macro)	0.761	0.759	0.747	676	575

To predict a failing build’s class, we decided to use random forest (RF) classifiers because RF classifiers have proven to be suitable models to handle overfitting induced by imbalanced classes while also achieving high accuracy [132, 176, 201]. Additionally, a previous study has also shown that RF classifiers can outperform other types of models in many situations [62].

During our experimentation phase, we also tried using an XGBoost classifier to see if it would outperform a random forest classifier. While the results were comparable between the two classifiers, the random forest algorithm slightly outperformed the XGBoost algorithm. In this section, we therefore only present the results obtained using the random forest classifier.

For our use case, we used a maximum tree depth of 20 and a Gini impurity criterion. We used a low maximum depth in order to minimize the chances of overfitting the data. For the criterion selection, we tried several different ones then selected Gini because it achieved the best performance. All other parameters including the number of trees (100 trees) were left as default and can be found in the official scikit-learn documentation.¹⁷ Since our labeled dataset is very small, we conducted a 10-fold cross validation to have a more robust estimation of how our model would perform in a real case scenario. Table 5.5 shows the results we obtained from a 10-Fold cross validation using random forest classifiers on our labeled data.

Our result show that the model was able to achieve an overall F1-score of 74.7% when trying to classify a failing build log into one of 6 possible categories. Looking more closely at the results, we find that there is potential to automate the detection and classification of at least three of the top five most common root causes of fuzzing build failure. The corpus related issues label, the issue downloading external resources label and the command and argument related issues label all have a F1-score of 88% or above. Additionally, the high precision for these three categories indicate that the model is unlikely to produce false positives that could mislead the developers trying to fix their builds. With a precision of 95% for these three categories, we therefore find that there is a real potential to automate these types of build failures for practitioners.

For compiler related failures and dependency related failures, however, our results indicate that the model was not able to recognize failure patterns in the build logs that belonged to these categories. While the model was able to achieve a F1-score of 73.6% for compiler related failures, the 67.1% precision might be too low for real case scenarios and where the model could mislead developers as to why their builds are failing. With only a single instance correctly predicted out of 58, the model was completely unable to recognize dependency related failures.

¹⁷<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

The results achieved by the model overall align with the annotators' manual labeling experience. The model performed well on root causes that were easy for the annotators to label. This can be attributed to some root causes (such as corpus related issues and issue downloading external resources) having very few patterns of failure. Additionally, these root causes often had little amount of unrelated text in the error section of the logs and also clear messages indicating the cause of failure. Alternatively, three distinct failure patterns for compiler issues were found while labeling. It is therefore not surprising that the model would have difficulties achieving high accuracy for compiler related failures. Similarly, build logs belonging to project dependency issues had several completely different patterns of failures and often produced very large build logs with only one or a few single lines indicating that the true cause of failure was due to an issue with a dependency.

Overall, this small experiment demonstrated that there is potential to automate the classification of fuzzing build failures in real case scenarios. While more work is needed for complex failures such as dependency related issues, the clear structure of some of the build logs makes it very easy to recognize and classify automatically. We therefore hypothesize that continuous fuzzing services such as OSS-Fuzz and Fuzzit [63] and CI/CD services that support fuzzing such as Travis CI [34], Circle CI [33] and Jenkins [111] could already benefit from prediction models to automatically diagnose fuzzing build failures.

6 Threats to validity

6.1 Threats to internal validity

Threats to the *internal validity* concern factors internal to our study that could have affected the results we obtained. The manual labeling process conducted during our qualitative analysis was a subjective process based on each author's understanding of a build failure with the error messages that were available in the failing build logs. To mitigate the chance of mislabeling, multiple authors further investigated ambiguous cases to agree on the correct root cause and labeled as "Not enough information" when no agreement could be made or a build failure was not clear enough.

For this study, we chose to manually label only a single build failure per project to avoid

project level biases. However, by only picking a single build failure we might be missing several other types of root causes that should be included in our taxonomy.

6.2 Threats to construct validity

Threats to the *construct validity* concern the relationship between our observation and the theory. Despite having three authors experienced with conducting research studies about build logs and having years of programming experience debugging their own failing builds, the true root cause of a build failure might generate error messages in the failing logs that could mislead the authors towards a different root cause and therefore mislabel a build log. To avoid this situation, the authors thoroughly looked online to find what kind of failures generate the error messages found in the build logs and investigated the GitHub repository of the target project as well as the official OSS-Fuzz GitHub repository to find discussions, issues or pull requests that could reveal the root cause of a build failure.

6.3 Threats to external validity

Threats to the *external validity* refers to the generalizability of our results. For this study we only used open source projects participating to OSS-Fuzz which often have a well established community with experienced developers. Our results may therefore not reflect the fuzzing situation of smaller projects with less experienced fuzzing developers. OSS-Fuzz currently supports state-of-the-art fuzzers which are very complex tools that support dozens of arguments and need significant fuzzing experience to configure properly. Using smaller and simpler fuzzers may therefore not cause as many build failures related to the configuration and the usability of the fuzzer for example. Additionally, because OSS-Fuzz only supports specific programming languages, using fuzzers that were developed using other languages not supported by OSS-Fuzz may pose different build maintenance challenges.

7 Related Work

7.1 Studies on build failures

Build failures have been extensively studied in various contexts such as source code compilation failures, Docker build failures, and Maven/Gradle/Ant build failures just to name a

few. Lou et al. [140] studied the symptoms of build failures and resolution patterns in three widely used build systems. They propose a taxonomy of 50 categories of build failures based on the failure symptoms and find that 67.96% of build issues can be fixed by modifying the build script code. Finally, they find that 20 categories of build failures have clear fix patterns and highlight these patterns. Wu et al. [208] studied build failures in Docker builds. In their study, they investigate over 850,000 Docker builds collected from over 3,000 open-source projects. They find that projects that build more frequently have a lower ratio of broken builds and projects that have a higher ratio of failing builds tend to take longer to fix build failures.

Rauschet *et al.* [177] investigated the cause of build failures in a continuous integration (CI) environment for 14 open-source Java projects. They found that the most common category of build failures (over 80% of build failures) was due to test failures. Their results also show that build failures are often not an isolated incident. In fact, their results show that more than 50% of all build failures follow a previous build failure.

Seo *et al.* [183] led a large scale empirical study at Google to find out the main causes of build failures in Google systems. In their study, they investigated over 26.6 million builds generated over a period of 9 months in Java and C++ projects. By mining error statements from build failures, they were able to find out the frequency at which each type of error appears in build failures and find that symbols that the compiler does not recognize are the most common cause of build failures. They then group error statements into categories of build failures and show that dependency related failures are the most common types of build failures. Their results align with other studies [55,161] that have also studied and found that dependency issues account for a significant portion of build failures.

The main path towards improving build maintenance practices and developing automation tools to alleviate some of the cost and overhead introduced by build maintenance is to first develop a deep understanding of where inefficiencies and build issues come from. Similar to how previous studies have conducted qualitative analysis to understand the nature of build issues in their respective fields (CI [177], build systems [140], etc.), our study's qualitative analysis provides insights as to what root causes are the main culprits of fuzzing build failures and build maintenance overhead. Additionally, our qualitative analysis pro-

vides empirical data to quantify the cost of build maintenance in the context of fuzzing. Our study complements previous work towards understanding and quantifying the impacts of build maintenance on development activities such as fuzzing.

7.2 Empirical studies on fuzzing

In recent years, the use of fuzzers has significantly increased in open source software projects. Consequently, the availability of fuzzing data has also increased and allowed researchers to conduct more empirical studies related to fuzzing practices, fuzzer performances and other fuzzing related qualitative studies.

For instance, Dinget *al.* [46] conducted one of the first empirical studies on OSS-Fuzz by mining and analyzing over 23,000 bugs collected from the official bug tracker [70]. In that study, Dinget *al.* [46] studied the lifecycle of bugs found by OSS-Fuzz fuzzers and found that OSS-Fuzz is effective at finding bugs early on and that developers are quick to patch the issues (especially in the case of serious errors such as buffer overflows). Their study also quantifies which types of bugs and faults are most commonly found by OSS-Fuzz fuzzers and highlight which types of errors tend to be flaky. Kelleret *al.* [116] also investigated the lifecycle of bugs found by OSS-Fuzz. Their findings align with Dinget *al.*'s findings that developers are generally quick to fix bugs once the issues are detected by OSS-Fuzz fuzzers. Kelleret *al.*' Study however finds that the median lifespan for a bug detectable via fuzzing is 324 days which indicates that there are still improvements to be made with respect to the detection of vulnerabilities and fuzzing tools in general.

The increased cost of finding bugs quickly and lower return on investment of dedicating more machines to fuzzing was empirically studied by Böhme *et al.* [25]. In their study, they dedicate four CPU years worth of fuzzing to fuzz over 300 open-source software systems. They find that adding exponentially more machines to find known bugs is much faster but finding unknown bugs is only linearly faster. Their results empirically prove that improving fuzzing tools leads to more significant gains in terms of efficiency, speed, and bug finding capabilities than purely increasing the amount of CPU power dedicated to fuzzing.

As introduced above, previous empirical studies on fuzzing have investigated various aspects related to maintaining fuzzing activities such as fuzzing bug fixing and the cost of

fuzzing over time. Our study complements previous empirical fuzzing work by investigating a new aspect vital to maintaining fuzzing activities namely, managing fuzzing build issues. As McIntosh et al. demonstrated in their 2011 study on build maintenance efforts [148], build maintenance can significantly impact development activities. We therefore hope that our study can help fuzzing developers get a better understanding of the challenges of build maintenance for fuzzing activities and try to plan strategies ahead of time to reduce the overhead introduced by fuzzing build maintenance.

8 Chapter Summary

As more vulnerabilities are introduced into software systems over time, development communities will likely turn to automated solutions such as fuzzing to find these vulnerabilities inside their software systems. With the increased adoption of fuzzing, new fuzzing developers will be faced with the challenges incurred by fuzzing activities starting from the configuration of fuzzers, to building the fuzzers, to running the fuzzers and fixing bugs found while fuzzing.

Our quantitative analysis showed that open source communities are able to maintain their fuzzing infrastructure and fix build issues that arise in a timely manner. In fact, our results show that most failing builds do not have a subsequent failing build indicating that open source developers do care a lot about security and are willing to dedicate their time to keeping their fuzzing build healthy. Our study provides practitioners with relevant insights and information regarding the challenges of build maintenance for fuzzing activities. Using a manual approach, we provide a clear taxonomy of root causes of build failures that fuzzing developers need to be aware of in their daily activities. We also find that a significant portion of fuzzing build failures are caused by issues not specific to fuzzing. We therefore highly encourage fuzzing practitioners to make use of build management tools and methodologies used in contexts other than fuzzing.

For researchers, this study provides valuable empirical fuzzing data on which further studies can build upon. In Section 2.2, we explain the process of mining OSS-Fuzz data so that developers can conduct their own studies. We also provide researchers a dataset of 677 failed fuzzing build logs labeled with the root cause of failure. Finally, in Section 5.3, we provide a proof of concept towards automating build maintenance for fuzzing by

demonstrating the feasibility of classifying fuzzing build failures automatically using machine learning techniques.

Our future work will aim to explore various ways to lower the barrier of entry of fuzzing activities so that projects of all sizes can start fuzzing their software regardless if they have fuzzing experts or not among their contributors. Specifically, we aim to explore how automation techniques can be used to abstract some of the complexity of starting and maintaining fuzzing activities over time.

Chapter 6

Conclusion

Rereferences

- [1] ClusterFuzz. <https://google.github.io/clusterfuzz/>, Last accessed 6 April 2024.
- [2] Go documentation, adding a test. <https://go.dev/doc/tutorial/add-a-test>.
- [3] Go fuzz tests. <https://go.dev/doc/security/fuzz/>.
- [4] Go version 1.18 release notes. <https://go.dev/blog/go1.18>.
- [5] Integrating a go project into oss-fuzz. <https://google.github.io/oss-fuzz/getting-started/new-project-guide/go-lang/>.
- [6] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, page 207–216. Association for Computing Machinery, 1993.
- [7] Anaconda. State of data science report 2022, 2022.
- [8] O. Analytica. Apache software flaw could result in major breaches. *Emerald Expert Briefings*, (oxan-es), 2021.
- [9] R. Arima, Y. Higo, and S. Kusumoto. Toward refactoring evaluation with code naturalness. In *Proceedings of the 26th Conference on Program Comprehension*, page 316–319, 2018.
- [10] E. Arisholm, L. Briand, and E. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

- [11] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring deep state spaces via fuzzing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP'20)*, pages 1597–1612, 2020.
- [12] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. Association for Computing Machinery, 2019.
- [13] Author. State of data science report, 2022.
- [14] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, page 104–113, 2012.
- [15] O. Baysal, I. Davis, and M. W. Godfrey. A Tale of Two Browsers. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, page 238–241, 2011.
- [16] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers and Security*, 2022.
- [17] bitcoin/bitcoin. Issue #19557. <https://github.com/bitcoin/bitcoin/issues/19557>, 2020. Last accessed on 07-12-2022.
- [18] bitcoin/bitcoin. Issue #20088. <https://github.com/bitcoin/bitcoin/issues/20088>, 2020. Last accessed on 07-12-2022.
- [19] bitcoin/bitcoin. Issue #20232. <https://github.com/bitcoin/bitcoin/issues/20232>, 2020. Last accessed on 07-12-2022.
- [20] M. Boehme, C. Cadar, and A. ROYCHOUDHURY. Fuzzing: Challenges and reflections. *IEEE Software*, pages 79–86, 2021.
- [21] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of*

- the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 109–120, 2016.
- [22] M. Böhme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2021.
- [23] M. Böhme and B. Falk. 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*, page 713–724. Association for Computing Machinery, 2020.
- [24] M. Böhme and B. Falk. 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)*, page 713–724, 2020.
- [25] M. Böhme and B. Falk. 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*, ESEC/FSE 2020, page 713–724, 2020.
- [26] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS’17)*, page 2329–2344, 2017.
- [27] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS’16)*, pages 1032–1043, 2016.
- [28] bytecodealliance/wasmtime. Issue #459. <https://github.com/bytecodealliance/wasmtime/issues/459>, 2019. Last accessed on 07-12-2022.
- [29] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.

-
- [30] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15)*, pages 725–741, 2015.
- [31] P. M. Chisnall. Questionnaire design, interviewing and attitude measurement. *Journal of the Market Research Society*, 35(4):392–393, 1993.
- [32] Chromium. Monorail, chromium. <https://bugs.chromium.org/>. Last accessed on 07-12-2022.
- [33] C. CI. Circle ci, 2011.
- [34] T. CI. Travis ci, 2011.
- [35] M. Claes, M. Mantyla, M. Kuuttila, and B. Adams. Abnormal Working Hours: Effect of Rapid Releases and Implications to Work Content. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 243–247, 2017.
- [36] ClickHouse/ClickHouse. Issue #32443. <https://github.com/ClickHouse/ClickHouse/issues/32443>, 2021. Last accessed on 07-12-2022.
- [37] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, pages 37 – 46, 1960.
- [38] D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan. The Impact of Rapid Release Cycles on the Integration Delay of Fixed Issues. *Empirical Software Engineering*, 23(2):835–904, 2018.
- [39] O. Dabic, E. Aghajani, and G. Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.
- [40] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. pages 185–194, 2007.
- [41] W. W. Daniel and C. L. Cross. *Biostatistics: a foundation for analysis in the health sciences*. Wiley, 2018.

- [42] N. V. Database. Log4shell vulnerability.
- [43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186. Association for Computational Linguistics, 2019.
- [44] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, page 404–428, 2006.
- [45] D. Dig and R. Johnson. The role of refactorings in api evolution. pages 389–398, 2005.
- [46] Z. Y. Ding and C. Le Goues. An empirical study of oss-fuzz bugs. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142, 2021.
- [47] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.
- [48] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 141–151, 2006.
- [49] Z. Du, Y. Li, Y. Liu, and B. Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE’22)*, page 2440–2451, 2022.
- [50] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu. Empirical study of restarted and flaky builds on travis ci. In *Proceedings of the 17th International Conference on Mining Software Repositories*, page 254–264. Association for Computing Machinery, 2020.

- [51] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, page 779–790. Association for Computing Machinery, 2014.
- [52] ethereum/solidity. Issue #6201. <https://github.com/ethereum/solidity/issues/6201>, 2019. Last accessed on 07-12-2022.
- [53] ethereum/solidity. Issue #7948. <https://github.com/ethereum/solidity/issues/7948>, 2019. Last accessed on 07-12-2022.
- [54] ethereum/solidity. Issue #6587. <https://github.com/ethereum/solidity/issues/6587>, 2021. Last accessed on 07-12-2022.
- [55] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang. Escaping dependency hell: Finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 463–474, 2020.
- [56] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. USENIX Association, 2020.
- [57] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
- [58] G. Fuzzing. Go fuzzing documentation, 2022.
- [59] K. Gallaba, M. Lamothe, and S. McIntosh. Lessons from eight years of operational data from a continuous integration service: An exploratory case study of circleci. In *Proceedings of the 44th International Conference on Software Engineering*, page 1330–1342. Association for Computing Machinery, 2022.
- [60] A. Garrido and R. Johnson. Challenges of refactoring c programs. page 6–14, 2002.

- [61] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, page 1095–1105, 2014.
- [62] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering*, pages 789–800. Association for Computing Machinery, 05 2015.
- [63] GitLab. Gitlab acquisition of fuzzit, 2019.
- [64] P. Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, pages 70–76, 2020.
- [65] golang/go. Issue #48179. <https://github.com/golang/go/issues/48179>, 2021. Last accessed on 07-12-2022.
- [66] golang/go. Issue #52292. <https://github.com/golang/go/issues/52292>, 2022. Last accessed on 07-12-2022.
- [67] Google. Getting started, bug disclosure guidelines, oss-fuzz. <https://google.github.io/oss-fuzz/getting-started/bug-disclosure-guidelines/>. Last accessed on 07-12-2022.
- [68] google. Getting started, oss-fuzz. <https://google.github.io/oss-fuzz/getting-started/>. Last accessed on 07-12-2022.
- [69] Google. Announcing oss-fuzz: Continuous fuzzing for open source software, 2016.
- [70] O.-F. Google. Chrome monorail bug tracker, 2023.
- [71] google/oss fuzz. Issue #5526. <https://github.com/google/oss-fuzz/issues/5526>. Last accessed on 07-12-2022.
- [72] google/oss fuzz. Issue #153. <https://github.com/google/oss-fuzz/issues/153>, 2016. Last accessed on 07-12-2022.

- [73] google/oss fuzz. Issue #243. <https://github.com/google/oss-fuzz/issues/243>, 2017. Last accessed on 07-12-2022.
- [74] google/oss fuzz. Issue #244. <https://github.com/google/oss-fuzz/issues/244>, 2017. Last accessed on 07-12-2022.
- [75] google/oss fuzz. Issue #273. <https://github.com/google/oss-fuzz/issues/273>, 2017. Last accessed on 07-12-2022.
- [76] google/oss fuzz. Issue #638. <https://github.com/google/oss-fuzz/issues/638>, 2017. Last accessed on 07-12-2022.
- [77] google/oss fuzz. Issue #815. <https://github.com/google/oss-fuzz/issues/815>, 2017. Last accessed on 07-12-2022.
- [78] google/oss fuzz. Issue #823. <https://github.com/google/oss-fuzz/issues/823>, 2017. Last accessed on 07-12-2022.
- [79] google/oss fuzz. Issue #1333. <https://github.com/google/oss-fuzz/issues/1333>, 2018. Last accessed on 07-12-2022.
- [80] google/oss fuzz. Issue #1344. <https://github.com/google/oss-fuzz/issues/1344>, 2018. Last accessed on 07-12-2022.
- [81] google/oss fuzz. Issue #1675. <https://github.com/google/oss-fuzz/issues/1675>, 2018. Last accessed on 07-12-2022.
- [82] google/oss fuzz. Issue #2094. <https://github.com/google/oss-fuzz/issues/2094>, 2019. Last accessed on 07-12-2022.
- [83] google/oss fuzz. Issue #2558. <https://github.com/google/oss-fuzz/issues/2558>, 2019. Last accessed on 07-12-2022.
- [84] google/oss fuzz. Issue #2635. <https://github.com/google/oss-fuzz/issues/2635>, 2019. Last accessed on 07-12-2022.

-
- [85] google/oss fuzz. Issue #3014. <https://github.com/google/oss-fuzz/issues/3014>, 2019. Last accessed on 07-12-2022.
 - [86] google/oss fuzz. Issue #3082. <https://github.com/google/oss-fuzz/issues/3082>, 2019. Last accessed on 07-12-2022.
 - [87] google/oss fuzz. Issue #3227. <https://github.com/google/oss-fuzz/issues/3227>, 2020. Last accessed on 07-12-2022.
 - [88] google/oss fuzz. Issue #3548. <https://github.com/google/oss-fuzz/issues/3548>, 2020. Last accessed on 07-12-2022.
 - [89] google/oss fuzz. Issue #3432. <https://github.com/google/oss-fuzz/issues/3432>, 2021. Last accessed on 07-12-2022.
 - [90] google/oss fuzz. Issue #6147. <https://github.com/google/oss-fuzz/issues/6147>, 2021. Last accessed on 07-12-2022.
 - [91] google/oss fuzz. Issue #6871. <https://github.com/google/oss-fuzz/issues/6871>, 2021. Last accessed on 07-12-2022.
 - [92] google/oss fuzz. Issue #6885. <https://github.com/google/oss-fuzz/issues/6885>, 2021. Last accessed on 07-12-2022.
 - [93] google/oss fuzz. Issue #7138. <https://github.com/google/oss-fuzz/issues/7138>, 2022. Last accessed on 07-12-2022.
 - [94] google/oss fuzz. Issue #7198. <https://github.com/google/oss-fuzz/issues/7198>, 2022. Last accessed on 07-12-2022.
 - [95] google/oss fuzz. Issue #7581. <https://github.com/google/oss-fuzz/issues/7581>, 2022. Last accessed on 07-12-2022.
 - [96] google/syzkaller. Issue #57. <https://github.com/google/syzkaller/issues/57>, 2016. Last accessed on 07-12-2022.

- [97] google/syzkaller. Issue #1622. <https://github.com/google/syzkaller/issues/1622>, 2020. Last accessed on 07-12-2022.
- [98] H. Green and T. Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, page 1070–1081, 2022.
- [99] M. Greiler, A. van Deursen, and M.-A. Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, pages 244–254, 2012.
- [100] J. Gu, X. Luo, Y. Zhou, and X. Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, page 1418–1430, 2022.
- [101] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. IEEE/ACM, 2018.
- [102] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, page 3248–3280, 2018.
- [103] S. Herbold, J. Grabowski, and H. Neukirchen. Automated refactoring suggestions using the results of code analysis tools. pages 104 – 109, 2009.
- [104] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, page 837–847, 2012.
- [105] A. Humayun, Y. Wu, M. Kim, and M. A. Gulzar. Naturalfuzz: Natural input generation for big data analytics. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1592–1603. Association for Computing Machinery, 2023.
- [106] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale. Software testing: Survey of the industry practices. In *Proceedings of the 41st International Convention on Information*

- and Communication Technology, Electronics and Microelectronics (MIPRO'18)*, pages 1449–1454, 2018.
- [107] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In *2019 IEEE International Conference on Software Maintenance and Evolution*, pages 186–190, 2019.
- [108] G. Inc. Oss-fuzz architecture. <https://google.github.io/oss-fuzz/architecture/>.
- [109] S. Inc. Heartbleed vulnerability, 2014.
- [110] ISO/IEC. Iso 25000 portal.
- [111] Jenkins. Jenkins.io, 2011.
- [112] Y. Jiang and B. Adams. Co-evolution of infrastructure and source code: an empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, page 45–55. IEEE Press, 2015.
- [113] Jung-Min Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, 2002.
- [114] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, page 1–10, 2010.
- [115] S. Karus and M. Dumas. Predicting coding effort in projects containing xml. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, page 203–212, 2012.
- [116] B. Keller, A. Meneely, and B. Meyers. What happens when we fuzz? investigating oss-fuzz bug history. In *Proceedings of the 20th International Conference on Mining Software Repositories*, pages 207–217. Association for Computing Machinery, 2023.

- [117] M. Kelly, C. Treude, and A. Murray. 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)*, pages 1–6, 2019.
- [118] N. Kerzazi and F. Khomh. Factors impacting rapid releases: An industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 98–107, 2014.
- [119] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.
- [120] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, 20(2):336–373, 2015.
- [121] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 179–188, 2012.
- [122] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160, 2011.
- [123] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 371–372, 2010.
- [124] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.
- [125] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, pages 633–649, 2014.

- [126] T. Klooster, F. Turkmen, G. Broenink, R. t. Hove, and M. Böhme. Effectiveness and scalability of fuzzing techniques in ci/cd pipelines, 2022.
- [127] E. Kula, A. Rastogi, H. Huijgens, A. Van Deursen, and G. Gousios. Releasing fast and slow: An exploratory case study at ING. In *Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 785–795, 2019.
- [128] G. Lacerda, F. Petrillo, M. Pimenta, and Y. Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, page 110610, 2020.
- [129] C. Lemieux and K. Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 475–485. Association for Computing Machinery, 2018.
- [130] T. Lethbridge, S. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10:311–341, 07 2005.
- [131] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. A qualitative study of the benefits and costs of logging from developers’ perspectives. *IEEE Transactions on Software Engineering*, 47(12):2858–2873, 2021.
- [132] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, page 1831–1865, 2017.
- [133] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan. Maff: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering (ICSE’22)*, page 1–12, 2022.
- [134] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

- [135] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, pages 1199–1218, 2018.
- [136] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566. IEEE, 2018.
- [137] B. Lin, C. Nagy, G. Bavota, and M. Lanza. On the impact of refactoring operations on code naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 594–598, 2019.
- [138] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, page 672–684. IEEE/ACM, 2022.
- [139] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang. History-driven build failure fixing: How far are we? In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 43–54, 2019.
- [140] Y. Lou, Z. Chen, Y. Cao, D. Hao, and L. Zhang. Understanding build issue resolution in practice: Symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 617–628, 2020.
- [141] Z. Lubsen, A. Zaidman, and M. Pinzger. Using association rules to study the co-evolution of production & test code. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 151–154, 2009.
- [142] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117. 2018, 2018.
- [143] S. Mallisery and Y.-S. Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Computing Survey*, pages 1–38, 2023.

- [144] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen. On rapid releases and software testing: A case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.
- [145] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47:2312–2331, 2021.
- [146] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, pages 1–31, 2017.
- [147] N. McDonald, S. Schoenebeck, and A. Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proceedings of the ACM on Human-Computer Interaction*, 3, 2019.
- [148] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, page 141–150. Association for Computing Machinery, 2011.
- [149] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–10, 2009.
- [150] H. D. Menendez and D. Clark. Hashing fuzzing: Introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering*, 48(9):3540–3553, 2022.
- [151] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE’22)*, page 1343–1355, 2022.
- [152] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 1615–1629, 2023.

- [153] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, pages 375–407, 2010.
- [154] M. Michlmayr, B. Fitzgerald, and K.-J. Stol. Why and how should open source projects adopt time-based releases? *IEEE Software*, 32(2):55–63, 2015.
- [155] V. M. Mika, F. Khomh, B. Adams, E. Engstr, and K. Petersen. On Rapid Releases and Software Testing. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 20–29, 2013.
- [156] W. Ming, W. Yongcong, X. Yifan, and J. Hai. Evaluating seed selection for fuzzing javascript engines. *Empir. Softw. Eng.*, 2023.
- [157] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Associating code clones with association rules for change impact analysis. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 93–103, 2020.
- [158] M. Moroz and K. Serebryany. Guided in-process fuzzing of Chrome components, 2016. <https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html>, Last accessed 6 April 2024.
- [159] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. 2007.
- [160] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? pages 287–297, 2006.
- [161] S. Mukherjee, A. Almanza, and C. Rubio-González. Fixing dependency errors for python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 439–451. Association for Computing Machinery, 2021.

- [162] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, page 287–297, 2009.
- [163] E. C. Neto, D. A. d. Costa, and U. Kulesza. Revisiting and improving szz implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–12, 2019.
- [164] H. L. Nguyen and L. Grunske. BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE’22)*, 2022.
- [165] O. Nourry, Y. Kashiwa, B. Lin, G. Bavota, M. Lanza, and Y. Kamei. The human side of fuzzing: Challenges faced by developers during fuzzing activities. pages 1–26. Association for Computing Machinery, 2023.
- [166] O. Nourry, Y. Kashiwa, B. Lin, G. Bavota, M. Lanza, and K. Yasutaka. Replication package for the paper: The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities, May 2023.
- [167] obgm/libcoap. Issue #213. <https://github.com/obgm/libcoap/issues/213>, 2018. Last accessed on 07-12-2022.
- [168] OpenAI. Gpt text-embedding-ada-002, 2022.
- [169] G. OSS-Fuzz. Oss-fuzz, 2016.
- [170] G. OSS-Fuzz. Oss-fuzz build logs storage api, 2023.
- [171] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3, 2019.
- [172] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension*, page 176–185, 2017.

- [173] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP'18)*, pages 697–710, 2018.
- [174] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, page 26–33, 2018.
- [175] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, page 1–26, 2023.
- [176] S. Quach, M. Lamothe, B. Adams, Y. Kamei, and W. Shang. Evaluating the impact of falsely detected performance bug-inducing changes in jit models. *Empirical Software Engineering*, 2021.
- [177] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, page 345–355. Association for Computing Machinery, 2017.
- [178] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, page 428–439, 2016.
- [179] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, pages 861–875, 2014.
- [180] J. Scarsbrook, M. Utting, and R. Ko. Typescript's evolution: An analysis of feature adoption over time. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories*, pages 109–114, 2023.
- [181] secdev/scapy. Issue #1398. <https://github.com/secdev/scapy/issues/1398>, 2018. Last accessed on 07-12-2022.

- [182] secdev/scapy. Issue #3347. <https://github.com/secdev/scapy/issues/3347>, 2021. Last accessed on 07-12-2022.
- [183] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers’ build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, page 724–734. Association for Computing Machinery, 2014.
- [184] K. Serebryany. OSS-Fuzz - Google’s continuous fuzzing service for open source software, 2017. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany>, Last accessed 6 April 2024.
- [185] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, page 1981–1993, 2013.
- [186] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna. 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*, page 347–362, 2017.
- [187] A. Silva, G. Carneiro, F. Brito e Abreu, and M. Monteiro. Frequent releases in open source software: A systematic review. *Information*, 8(3):109, 2017.
- [188] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 858–870, 2016.
- [189] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 269–279, 2017.
- [190] L. Sousa, D. Cedrim, W. Oizumi, A. Bibiano, A. Oliveira, A. Garcia, D. Oliveira, and M. Kim. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. 10 2020.

- [191] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.
- [192] K. Stroggylos and D. Spinellis. Refactoring—does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, page 10, 2007.
- [193] P. Thongtanunam, W. Shang, and A. E. Hassan. Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering*, 24(2):937–972, 2019.
- [194] G. N. Tonietto, S. A. Malkoc, and S. M. Nowlis. When an Hour Feels Shorter: Future Boundary Tasks Alter Consumption by Contracting Time. *Journal of Consumer Research*, 45(5):1085–1102, 2018.
- [195] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinianian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494, 2018.
- [196] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.
- [197] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, 28:1–29, 2019.
- [198] M. Valiev, B. Vasilescu, and J. Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 644–655. Association for Computing Machinery, 2018.
- [199] C. Vassallo, G. Grano, F. Palomba, H. Gall, and A. Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15, 2019.

- [200] C. Vassallo, F. Palomba, and H. C. Gall. Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In *International Conference on Software Maintenance and Evolution*, pages 564–568, 2018.
- [201] A. Verma and X. Dong. Detection of ventricular fibrillation using random forest classifier. *Journal of Biomedical Science and Engineering*, 2016.
- [202] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, page 1–12, 2006.
- [203] H. Wang, S. Liu, L. Zhang, and C. Xu. Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 198–210, 2023.
- [204] A. Wei, Y. Deng, C. Yang, and L. Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering (ICSE’22)*, page 995–1007, 2022.
- [205] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-scale automated refactoring using clangmr. In *2013 IEEE International Conference on Software Maintenance*, pages 548–551, 2013.
- [206] M. Wu, K. Chen, Q. Luo, J. Xiang, J. Qi, J. Chen, H. Cui, and Y. Zhang. Enhancing coverage-guided fuzzing via phantom program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1037–1049. Association for Computing Machinery, 2023.
- [207] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*, page 1634–1645. Association for Computing Machinery, 2022.

- [208] Y. Wu, Y. Zhang, T. Wang, and H. Wang. An empirical study of build failures in the docker context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, page 76–80. Association for Computing Machinery, 2020.
- [209] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.
- [210] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, 2006.
- [211] Q. Yan, M. Huang, and H. Cao. A survey of human-machine collaboration in fuzzing. In *Proceedings of the 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC'22)*, pages 375–382, 2022.
- [212] J. Yun, F. Rustamov, J. Kim, and Y. Shin. Fuzzing of embedded systems: A survey. 55(7), 2022.
- [213] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, pages 325–364, 2011.
- [214] M. Zalewski. American fuzzy lop (2.52b).
- [215] C. Zhang, B. Chen, J. Hu, X. Peng, and W. Zhao. Buildsonic: Detecting and repairing performance-related configuration smells for continuous integration builds. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2023.
- [216] P. Zhang, B. Ren, H. Dong, and Q. Dai. Cagfuzz: Coverage-guided adversarial generative fuzzing testing for image-based deep learning systems. *IEEE Transactions on Software Engineering*, 48(11):4630–4646, 2022.

- [217] C. Zhou, M. Wang, J. Liang, Z. Liu, C. Sun, and Y. Jiang. Visfuzz: Understanding and intervening fuzzing with interactive visualization. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, pages 1078–1081, 2019.
- [218] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang. Minerva: Browser api fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1135–1147, 2022.
- [219] X. Zhu and M. Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*, pages 2169–2182, 2021.
- [220] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54:1–36, 2022.
- [221] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.