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

Olivier Nourry*, Masanari Kondo*, Mahmoud Alfadel†, Shane McIntosh†, Yasutaka Kamei*
*Kyushu University, Japan — oliviern@posl.ait.kyushu-u.ac.jp, {kondo,kamei}@ait.kyushu-u.ac.jp
†Software REBELs, University of Waterloo, Canada — {malfadel,shane.mcintosh}@uwaterloo.ca

***Abstract*—Fuzz testing (or fuzzing) is a software testing technique aimed at identifying software vulnerabilities. Recently, the Go community added native support for fuzz testing into their standard library. Using that feature, developers can write unit tests to perform deterministic and fuzz testing of their software systems against unexpected inputs. Although the availability of support makes fuzz testing more accessible for the Go community at large, little is known about the degree to which Go developers adopt fuzz testing during software development. Therefore, in this paper, we set out to study the evolution of fuzz testing practices in open-source Go projects. More specifically, we strive to understand whether the introduction of support for fuzz testing in the Go standard library has led to the adoption of fuzz testing as part of the standard testing processes of Go projects. To achieve our goal, we study 1) to what extent fuzz tests are used in open-source Go projects, 2) who writes and maintains fuzz tests in Go projects, and finally, 3) how tightly coupled are fuzz tests with source code (as compared to non-fuzz tests). We find that fuzz testing only represents 3.15% of testing functions in open-source projects. Our results also suggest that fuzz testing development is not being conducted as part of standard testing activities. For developers contributing to fuzzing, we find that a median of only 12.50% of their testing-related commits contain fuzz tests. Finally, we perform a qualitative analysis and find that fuzz testing is mostly used by critical software systems, such as blockchain technologies or network infrastructure projects, to test the most critical features of their systems (e.g., data processing functions, database endpoints). Our results lead us to conclude that fuzz testing is best used in combination with deterministic testing (e.g., unit testing) where fuzzing is used to thoroughly test important features, and deterministic testing is used to test other features.**

I. Introduction

Anaconda's 2022 yearly survey on the state of data science [1] showed that the fear of vulnerabilities (such as the Log4j vulnerability[1]) 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 [2, 3]. 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-2019-16411[4]).

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 [4, 5], 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 growing interest in fuzzing [2, 3] within the open-source community, 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)[5] within the Go programming language itself to empower developers to write fuzz tests just as easily as unit tests. Additionally, the efforts of the open-source community to make the adoption of fuzzing easier have motivated the addition of Go features to facilitate the adoption of OSS-Fuzz (issue #50192)[6] and have also led OSS-Fuzz developers to add native Go fuzzing support [6].

Since unit tests do not scale well to test against large amounts of abnormal inputs, the increased accessibility of fuzzing now provides developers with the opportunity to adopt fuzz testing as part of their standard software testing practices. With the considerable security benefits provided by combining deterministic testing, such as unit testing, with scalable approaches, such as fuzzing, we believe that the research community has an important role to play 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.** We therefore set out to study the co-evolution of fuzz testing

[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
[5]https://github.com/golang/go/issues/44551
[6]https://github.com/golang/go/issues/50192

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 [7] and aim to answer the following preliminary questions:

(PQ1) Do developers use fuzz testing?
(PQ2) Is fuzzing test code more prone to change than non-fuzzing test code?

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

(RQ1) Is fuzz testing development conducted as part of the standard testing process?
(RQ2) Who conducts fuzz testing in open-source projects?

Based on the preliminary and research questions mentioned above, we summarize the findings of this paper as follows:

- (PQ1) 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.
- (PQ2) 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.
- (RQ1) 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.
- (RQ2) 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 require rigorous testing. From the number of developers that have contributed to the development of fuzz tests, we also conclude that more work needs to be done to lower the barrier of entry to fuzz testing.

II. Background and Related Work

In this section, we define key concepts and then situate our work with respect to the literature.

A. Background

Challenges in the adoption of 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 [8]. 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. interviewed 25 senior practitioners to learn more about the challenges of testing plug-in-based systems [9]. Their study reveals that even experienced developers face difficulties writing tests for their software systems.

As the adoption of fuzzing into the software testing process increases over time [10, 11, 12], software testers and developers will need to manage the challenges incurred by fuzz testing. In our prior work, we conducted a survey with fuzzing experts that revealed over 22 distinct challenges of conducting fuzzing activities [13]. We found that the usability of fuzzers (i.e., setting up, building, or using a fuzzer) is the key challenge that experts are facing when conducting fuzz testing. Thus, making fuzzing more accessible could lower barriers to the adoption 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 [14]. To ease 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.[7] 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 set out to understand 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 target of the test is the `GenKeyPair` function, which is designed to generate a pair of encryption keys. As we can see, the structure to write Go fuzz tests closely resembles 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 unit testing, the initial values provided to fuzz tests will also be used to generate new inputs for further testing.

B. 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 [7]. They find that build maintenance introduces considerable 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.

[7] https://github.com/golang/go/issues/44551#issuecomment-785176317

[8] Retrieved 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 example[8]

Zaidman et al. studied the co-evolution of production and test code in both industrial and open source projects [15]. 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. Jiang and Adams studied the co-evolution of infrastructure code and source code in 265 OpenStack projects [16]. 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 [17]. 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. Mazinanian et al. [18] 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 [19]. Their observations align with the results of Mazinanian 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 IV-A) 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 the conclusions of the previous studies, i.e., we also find 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 [20]. Their results show that fuzzing is an effective technique to find bugs and that developers typically fix detected bugs quickly. Yet they also find that some flaky bugs detected by fuzzing are problematic to address or reproduce. Moreover, they find that fuzzing campaigns tend to find large numbers of bugs in short timeframes requiring developers to deal with high numbers of bug reports at once.

Böhme et al. conducted a four cpu year-long fuzz campaign to study the increasing cost of fuzzing over time [21]. 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. Our prior work mined fuzzing-related GitHub issues of open-source software repositories [13]. After manually labeling these issues, we 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 our prior work had previously been highlighted in an industrial setting by Liang et al., where a team of developers at Huawei tried to start fuzzing one of their libraries [22].

This paper makes the following contributions: (1) we conduct a large-scale empirical study on 11,341 projects to study the usage and adoption of native Go fuzzing, (2) we shed light on how open-source communities conduct testing activities, and (3) we propose a list of candidate functions for developers to fuzz based on our investigation of how fuzz testing is being applied in practice.

III. Studied Data

Since the goal of our study is to investigate the usage and evolution of fuzz testing in OSS projects, we first need to find projects that contain fuzz tests. In this section, we describe our

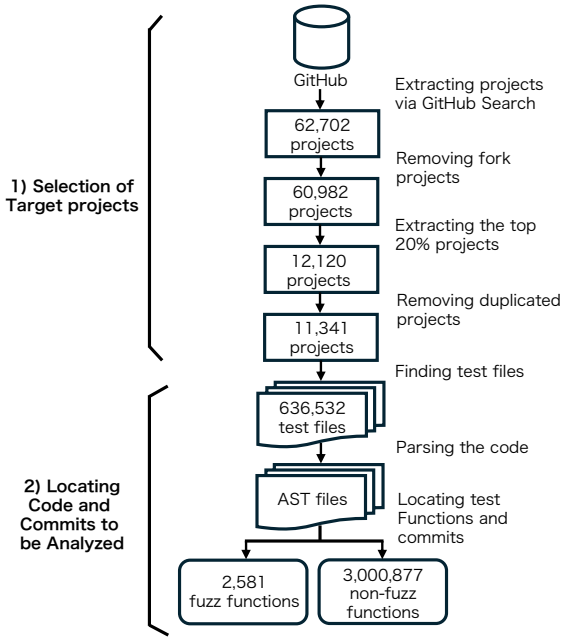

Fig. 1. Overview of the data collection process

approach for selecting projects and locating code and commits to be analyzed within the studied projects. Figure 1 provides an overview of the data collection process.

A. Selection of Target Projects

We begin with the GitHub search tool [23] to find projects. To collect all projects, we apply the language filter, where we specify that the projects must be written in Go (i.e., the primary language of the project is Go). This produces a list of 62,702 projects (as of March 2024). We then filter out projects that are forks, which reduced the list to 60,982 projects.

Next, since we are interested in how open-source projects use fuzzing, we need to mitigate the likelihood of toy projects being included in our dataset. Therefore, we select only those projects within the top 20% (12,120 projects) in terms of number of commits (x $>$ 246 commits). Because larger projects tend to have larger communities, we also assume that large projects are more likely to have open-source contributors conducting fuzz testing within their development team. By selecting only the large projects, we therefore increase our odds of finding fuzz tests within the studied projects.

While inspecting our data for inconsistencies, we observe duplicated projects due to project renaming. For these cases, although two projects have 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 deduplicate our data, we therefore remove any project in our dataset whose URL redirects to the latest up-to-date repository. This process further reduces the number of projects to 11,341. We then clone these repositories and proceed with the code parsing process.

B. 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 [24], all test code must be located in a file with the "*_test.go*" suffix. To find test code in our studied projects, we therefore search for all files ending in "*_test.go*". In total, we detect 636,532 test files in our dataset.

Parsing the code. After locating the test files, we use the *parser package* of the Go standard library[9] to parse all Go test files in our dataset. The parser produces Abstract Syntax Trees (ASTs) in JSON format, which show all test function names along with their return and parameter types. Additionally, the parser logs the first and last line numbers of each function in the test files. The parsing process is applied to all test files for every revision in the history of our studied projects.

Locating test functions. Next, we set out to detect test functions (both non-fuzzing and fuzzing test functions) within the test files. To write a fuzz test, developers must follow the naming conventions that is specified in the official Go documentation [25], i.e., the function name of a fuzz test must start with the word "*Fuzz* (e.g., FuzzTestOne)". Moreover, the fuzz target must be a function call to the *(*testing.F).Fuzz* function with **testing.T* being the type of the first parameter.

To locate fuzz tests, we traverse the ASTs that we obtain from parsing the test files in search of functions that satisfied the naming constraints. Since the parser logs the name of all functions in the test files, we also search for functions that satisfy the naming constraints for non-fuzzing tests, such as unit tests (the function name starts with the word "*Test*").

Labeling commits. After locating fuzzing tests/code, we extract the starting and end lines of each test function for every revision from the ASTs. For each commit, we then use git to extract the names of the changed files along with the line numbers where the changes are recorded. From there, we match the line numbers from the AST output and the "diff" output to label all commits as "non-fuzzing", "fuzzing", or "source" according to whether the changes modify fuzzing code, non-fuzzing test code, or code outside of test files. Note that these categories are not mutually exclusive, and one commit may have multiple labels.

IV. Preliminary Analysis

A. 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 III-B, we were able to find the number of test files, non-fuzz test functions, and fuzz test functions in our studied projects. In

[9]https://pkg.go.dev/go/parser

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 most projects still do not make use of native Go fuzzing features 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 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.

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 percentage of testing activities in our studied projects.

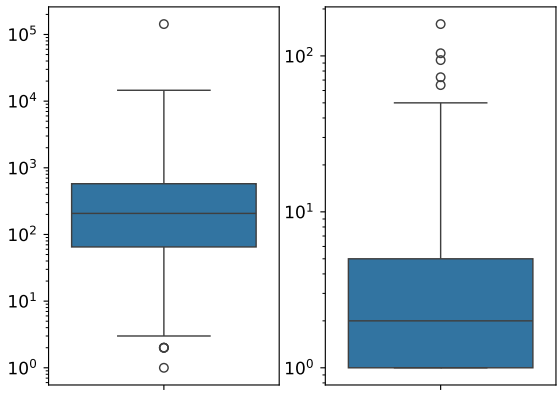

Fig. 2. Number of non-fuzz tests (left) and fuzz tests (right) per project containing at least one fuzz test (log scaled)

> 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 percentage of total testing functions. These results indicate that more work needs to be done to spread the adoption of fuzzing among open-source communities.

B. 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 it requires 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 III-B, 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 III-B. 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 then calculated 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 III-B, 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.

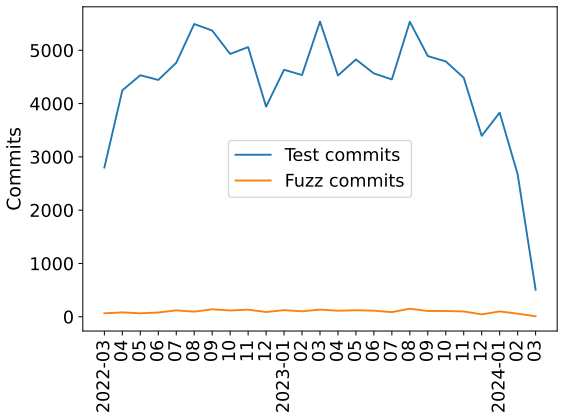

Fig. 3. Number of commits modifying non-fuzzing test functions and number of commits modifying fuzzing test functions per month since fuzz tests were added to the standard library. Note: The drop at the start of 2024 is because the data had not been made available yet.

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 due in part to fuzz tests not requiring 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 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 to 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 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 on non-fuzz testing than fuzz testing.

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

Motivation. In Section IV-B, 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. After removing commits older than the introduction of native Go fuzzing, we use association rules [26, 27, 28, 29] 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 [26]. In other words, they are rules used to find associations and relationships between different types of data points. Table I shows how we calculated the association rules used in this paper.

TABLE I
Association Rule Calculations

Rule	Calculation
Support(X)	# type X revisions / # total revisions
Conf(X –>Y)	Support(X,Y) / Support(X)
Lift(X–>Y)	Conf(X–>Y) / 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 → Test)* would be measured as (1/5) = 0.2 and the *Conf (Test → 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 → Test)* $\neq$ *Conf (Test → 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 indicates that we are observing fewer instances than we would expect from

TABLE II
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 → Test	0.369
	Source → Fuzz	0.007
	Test → Source	0.861
	Test → Fuzz	0.015
	Fuzz → Source	0.909
	Fuzz → Test	0.750
Lift	Source → Test	1.239
	Source → Fuzz	1.204
	Test → Fuzz	2.393

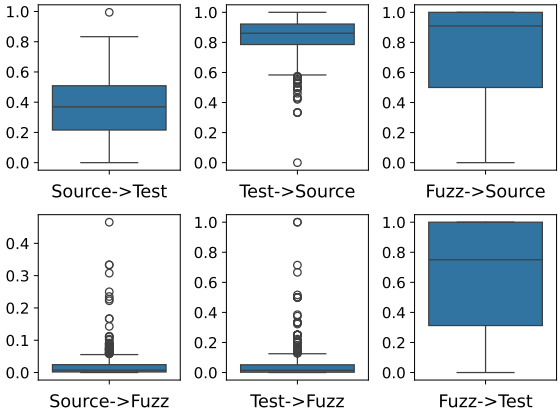

Fig. 4. Distribution of confidence values across studied projects.

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 →Y) = Lift(Y→X))

Results. Table II 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 → Test) value reveals 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 → 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 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 Conf(Test → Fuzz) value being twice as large as the Conf(Source → 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 activity in our studied projects. This pattern is further highlighted by the Conf(Fuzz → Source) and Conf(Fuzz → Test) values (0.909 and 0.750 respectively) and the distribution of confidence values in Figure 4 which show that fuzz test changes seem to be accompanied by both source code changes and standard test code changes indiscriminately.

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.

VI. 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 [13].

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 As we did in Section V, we first removed commits older than the introduction of native Go fuzzing. From the mining process described in Section III-B, 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, deletes, 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 III 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 to unit tests.[10]

[10]https://github.com/golang/go/issues/44551#issuecomment-785176317

TABLE III
MEDIAN NUMBER OF DEVELOPERS PARTICIPATING TO STANDARD TESTING ACTIVITIES, 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

We find that both standard testing and fuzz testing rely on a few contributors to take on most of the testing workload. Table III 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 III, 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).

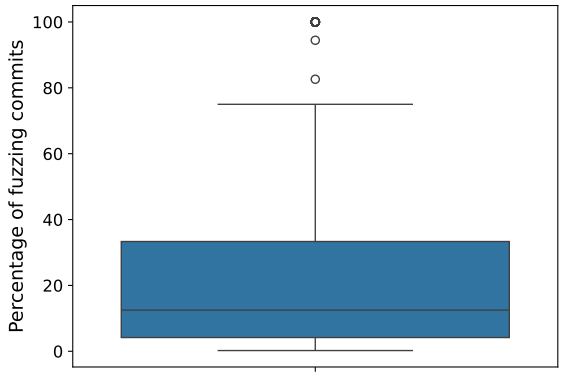

Fig. 5. Percentage of testing commits that modify fuzzing code for developers conducting fuzzing activities.

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 percentage of testing commits (i.e., commits that change either a non-fuzz test or a fuzz test.) contain changes to fuzzing test code. Figure 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 fewer 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 who focus on testing. Our results suggest that fuzzing development is not conducted by fuzzing specialists but rather by contributors dedicated to testing activities in general.

VII. Discussion

A. 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 Liang et al. [22] 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 [21] has also shown that long fuzz sessions are extremely inefficient to run. Due to these limitations, it is possible that 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, it still has its 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 V 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 development and a custom database driver.

TABLE V
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
mtoohey31/iter	50	Go package for iterator functions
gravitational/teleport	50	Network infrastructure platform
scylladb/scylla-go-driver	47	Experimental, high performance ScyllaDB 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 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.

B. 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 V 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 IV 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.

TABLE IV
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.

From Table IV, 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 that 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*[11] 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.

C. 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' results are consistent with our quantitative results where we 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 [30, 31]. For those not familiar with fuzz testing, we recommend fuzzing functions such as the ones listed in Table IV.

[11]https://github.com/sourcegraph/zoekt

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 the number of developers that conduct fuzz testing.

VIII. 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 concern the relationship between theory and observation. To find and mine Go projects, we used the GitHub search tool published by Dabic et 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. Additionally, we identified fuzz test functions based on their arguments as specified in the official Go documentation. It is however possible that

developers might have used aliased imports which would cause some of our identified functions

IX. Conclusion

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 provide researchers and software engineers with 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.

X. Replication Package

We provide all required scripts to replicate the data mining process, the data filtering process, and the data handling to obtain our results. The scripts can be found at the following link: https://anonymous.4open.science/r/ICSME2024-GoFuzzEvolutionReplication-64E8/README.md

XI. Acknowledgements

We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council (NSERC) of Canada. We also gratefully acknowledge the financial support of: (1) JSPS for the KAKENHI grants (JP21H04877, JP22K18630, JP22K17874, JP24K02921), and Bilateral Program grant JPJSBP120239929; and (2) the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

References

[1] Author, "State of data science report," 2022. [Online]. Available: https://www.anaconda.com/state-of-data-science-report-2022

[2] 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.

[3] "ClusterFuzz," https://google.github.io/clusterfuzz/, Last accessed 6 April 2024.

[4] Google, "Announcing oss-fuzz: Continuous fuzzing for open source software," 2016. [Online]. Available: https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[5] ——, "Google, oss-fuzz," https://google.github.io/oss-fuzz/.

[6] G. OSS-Fuzz, "Integrating a go project into oss-fuzz," https://google.github.io/oss-fuzz/getting-started/new-project-guide/go-lang/.

[7] 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*. Association for Computing Machinery, 2011, p. 141–150.

[8] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, pp. 633–649, 2014.

[9] 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*, 2012, pp. 244–254.

[10] 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.

[11] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, pp. 70–76, 2020.

[12] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Survey*, pp. 1–38, 2023.

[13] 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." Association for Computing Machinery, 2023, pp. 1–26.

[14] "Go version 1.18 release notes," https://go.dev/blog/go1.18.

[15] 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*, pp. 325–364, 2011.

[16] 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*. IEEE Press, 2015, p. 45–55.

[17] 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*. Association for Computing Machinery, 2014, p. 779–790.

[18] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in java," *Proc. ACM Program. Lang.*, pp. 1–31, 2017.

[19] 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*, 2023, pp. 109–114.

[20] 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)*, pp. 131–142, 2021.

[21] 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*. Association for Computing Machinery, 2020, p. 713–724.

[22] 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)*. IEEE, 2018, pp. 562–566.

[23] 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*. IEEE, 2021, pp. 560–564.

[24] "Go documentation, adding a test," https://go.dev/doc/tutorial/add-a-test.

[25] "Go fuzz tests," https://go.dev/doc/security/fuzz/.

[26] 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*. Association for Computing Machinery, 1993, p. 207–216.

[27] 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*, 2004, pp. 563–572.

[28] 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)*, 2020, pp. 93–103.

[29] 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*, 2009, pp. 151–154.

[30] 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*. Association for Computing Machinery, 2018, p. 644–655.

[31] 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*. IEEE/ACM, 2022, p. 672–684.