My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures

OLIVIER NOURRY, Kyushu University, Japan
YUTARO KASHIWA, Nara Institute of Science and Technology, Japan
WEIYI SHANG, University of Waterloo, Canada
HONGLIN SHU, Kyushu University, Japan
YASUTAKA KAMEI, Kyushu University, Japan

Fuzzing is an automated software testing technique used to find software vulnerabilities that works by sending large amounts of inputs to a software system to trigger bad behaviors. In recent years, the open-source software ecosystem has seen a significant increase in the adoption of fuzzing to avoid spreading vulnerabilities throughout the ecosystem. While fuzzing can uncover vulnerabilities, there is currently a lack of knowledge regarding the challenges of conducting fuzzing activities over time. Specifically, fuzzers are very complex tools to set up and build before they can be used.

We set out to empirically find out how challenging is build maintenance in the context of fuzzing. We mine over 1.2 million build logs from Google's OSS-Fuzz service to investigate fuzzing build failures. We first conduct a quantitative analysis to quantify the prevalence of fuzzing build failures. We then manually investigate 677 failing fuzzing builds logs and establish a taxonomy of 25 root causes of build failures. We finally train a machine learning model to recognize common failure patterns in failing build logs. Our taxonomy can serve as a reference for practitioners conducting fuzzing build maintenance. Our modeling experiment shows the potential of using automation to simplify the process of fuzzing.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*; *Software maintenance tools*; • **General and reference** → **Empirical studies**; • **Security and privacy** → *Vulnerability management*.

Additional Key Words and Phrases: Fuzzing, Empirical Study, Build Maintenance

ACM Reference Format:
Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, Honglin Shu, and Yasutaka Kamei. 2024. My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures. 1, 1 (August 2024), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 Hearthbleed vulnerability [27] and the Log4Shell vulnerability [10]. 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

Authors' addresses: Olivier Nourry, oliviern@posl.ait.kyushu-u.ac.jp, Kyushu University, Japan; Yutaro Kashiwa, yutaro.kashiwa@is.naist.jp, Nara Institute of Science and Technology, Japan; Weiyi Shang, wshang@uwaterloo.ca, University of Waterloo, Canada; Honglin Shu, shu.honglin.167@s.kyushu-u.ac.jp, Kyushu University, Japan; Yasutaka Kamei, kamei@ait.kyushu-u.ac.jp, Kyushu University, Japan.

ACM XXXX-XXXX/2024/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [27], Google launched the OSS-Fuzz [50] 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 [17].

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 [3, 36, 40, 64]. In 2020, a Shonan meeting [4] 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 [32, 59, 60], 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 [13, 29] and empirically find out more about current fuzzing practices and challenges. In 2023, Nourry et al. [47] 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 languages 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.) [14, 18, 54, 55, 61], we set out to learn more about build failures in the context of fuzzing using openly availably 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 [25, 43, 44, 52], 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 [2, 63]. For software testing, a fuzz target can be a piece of source code that expects a specific input to be executed [17]. 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.

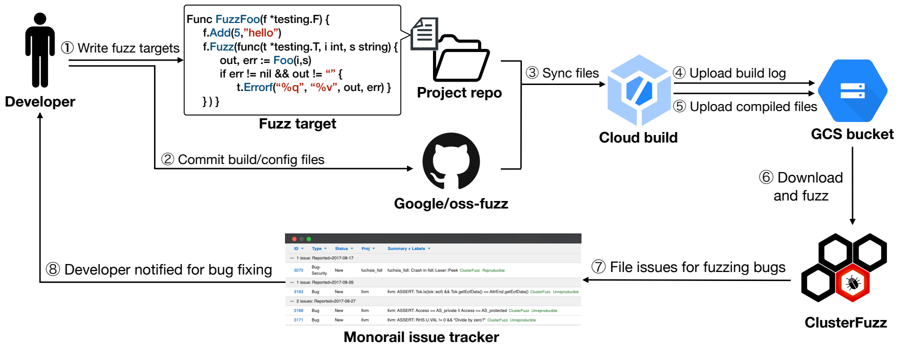
If a build process successfully completes, the fuzzer is then executed and starts continuously sending inputs to the fuzz target(s). If the developer provided a corpus to the fuzzer, the inputs contained in the corpus will be used to try to crash the target system and also to generate new inputs via mutations. These mutations can be done using a variety of strategies. For example, the popular AFL++ fuzzer [16] 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.

[1] https://google.github.io/oss-fuzz/getting-started/continuous-integration/
[2] https://github.com/google/oss-fuzz/issues/3014



```
Func FuzzFoo(f *testing.F) {
    f.Add(5,"hello")
    f.Fuzz(func(t *testing.T, i int, s string) {
        out, err := Foo(i,s)
        if err != nil && out != "" {
            t.Errorf("%q", "%v", out, err) }
    }) }
```

ID	Type	Status	Proj	Summary + Labels
— 1 issue: Reported=2017-08-17				
3070	Bug-Security	New	fuchsia_fidl	fuchsia_fidl: Crash in fidl::Lexer::Peek ClusterFuzz Reproducible
— 1 issue: Reported=2017-08-26				
3163	Bug	New	llvm	llvm: ASSERT: Tok.is(tok::eof) && Tok.getEofData() == AttrEnd.getEofData() ClusterFuzz Unreproducible
— 2 issues: Reported=2017-08-27				
3168	Bug	New	llvm	llvm: ASSERT: Access == AS_private ll Access == AS_protected ClusterFuzz Unreproducible
3171	Bug	New	llvm	llvm: ASSERT: RHS.U.VAL != 0 && "Divide by zero?" ClusterFuzz Unreproducible

Fig. 1. OSS-Fuzz architecture

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 external service, and using OSS-Fuzz' bug issue tracker are all additional challenges incurred by using an external provider rather than fuzzing locally. Figure 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 [26], 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 (e.g., 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.

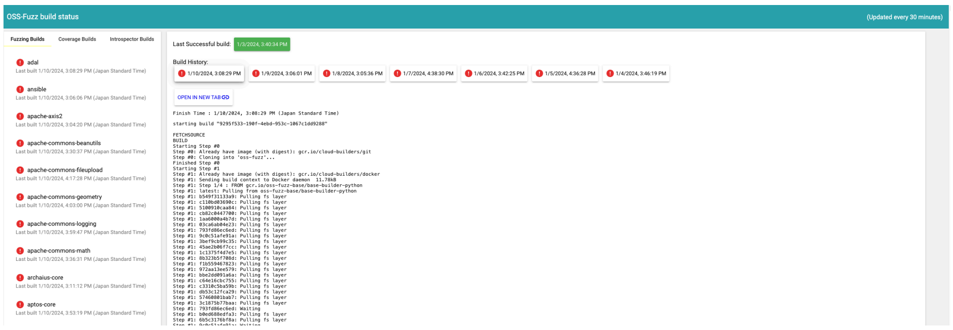
OSS-Fuzz' process flow. Once all the build files and configuration files are uploaded to the OSS-Fuzz repository, the OSS-Fuzz service will use a cloud builder[4] to build the project using the provided build/config files (③). At the end of the build process, the resulting build log will be uploaded 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 infrastructure [5] will download the fuzz targets, start fuzzing the project and

[3]https://github.com/google/oss-fuzz
[4]https://cloud.google.com/build/docs/cloud-builders
[5]https://google.github.io/clusterfuzz/

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 [21] 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., LLVM project 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 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 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.



adal
Last built 1/10/2024, 3:08:29 PM (Japan Standard Time)
ansible
Last built 1/10/2024, 3:06:06 PM (Japan Standard Time)
apache-axis2
Last built 1/10/2024, 3:04:20 PM (Japan Standard Time)
apache-commons-beanutils
Last built 1/10/2024, 3:30:37 PM (Japan Standard Time)
apache-commons-fileupload
Last built 1/10/2024, 4:17:28 PM (Japan Standard Time)
apache-commons-geometry
Last built 1/10/2024, 4:03:00 PM (Japan Standard Time)
apache-commons-logging
Last built 1/10/2024, 3:59:47 PM (Japan Standard Time)
apache-commons-math
Last built 1/10/2024, 3:36:31 PM (Japan Standard Time)
archaius-core
Last built 1/10/2024, 3:11:12 PM (Japan Standard Time)
aptos-core
Last built 1/10/2024, 3:53:19 PM (Japan Standard Time)

Last Successful build: 1/3/2024, 3:40:34 PM

Build History: 1/10/2024, 3:08:29 PM | 1/9/2024, 3:06:01 PM | 1/8/2024, 3:05:36 PM | 1/7/2024, 4:38:30 PM | 1/6/2024, 3:42:25 PM | 1/5/2024, 4:36:28 PM | 1/4/2024, 3:46:19 PM

```
Finish Time : 1/10/2024, 3:08:29 PM (Japan Standard Time)

starting build "9295f533-190f-4ebd-953c-1067c1dd9288"

FETCHSOURCE
BUILD
Starting Step #0
Step #0: Already have image (with digest): gcr.io/cloud-builders/git
Step #0: Cloning into 'oss-fuzz'...
Finished Step #0
Starting Step #1
Step #1: Already have image (with digest): gcr.io/cloud-builders/docker
Step #1: Sending build context to Docker daemon  11.78kB
Step #1: Step 1/4 : FROM gcr.io/oss-fuzz-base/base-builder-python
Step #1: latest: Pulling from oss-fuzz-base/base-builder-python
Step #1: b549f31133a9: Pulling fs layer
Step #1: c110bd03690c: Pulling fs layer
Step #1: 5100910caa84: Pulling fs layer
Step #1: cb82c0447700: Pulling fs layer
Step #1: 1aa6000a4b7d: Pulling fs layer
Step #1: 03ca6ab04e23: Pulling fs layer
Step #1: 793fd86ec6ed: Pulling fs layer
Step #1: 9c0c51afe91a: Pulling fs layer
Step #1: 3bef9cb99c35: Pulling fs layer
Step #1: 45ae2b06f7cc: Pulling fs layer
Step #1: 1c1375f4d7e5: Pulling fs layer
Step #1: 8b323b5f708d: Pulling fs layer
Step #1: f1b559467823: Pulling fs layer
Step #1: 972aa13ee579: Pulling fs layer
Step #1: bbe2dd091a6a: Pulling fs layer
Step #1: c64e16cbc755: Pulling fs layer
Step #1: c3310c5ba59b: Pulling fs layer
Step #1: db53c12fca29: Pulling fs layer
Step #1: 57460801bab7: Pulling fs layer
Step #1: 3c1875b77baa: Pulling fs layer
Step #1: 793fd86ec6ed: Waiting
Step #1: b0ed688edfa3: Pulling fs layer
Step #1: 6b5c3176bf8a: Pulling fs layer
```

Fig. 2. OSS-Fuzz public panel to OSS-Fuzz projects' fuzzing builds status

As stated 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. As shown in Figure 2, a dashboard [51] showing projects being fuzzed by OSS-Fuzz is publicly available online. Using this dashboard, anyone can view a build log from each of the last 7 days for each project and the date of the last passing build. 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 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

[6]https://google.github.io/oss-fuzz/getting-started/new-project-guide/#build_frequency
[7]https://cloud.google.com/storage/docs/json_api

projects' fuzzing builds, we were finally able to mine the entire history of fuzzing builds logs from OSS-Fuzz' GCS bucket.

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 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 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 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 reposiories, we first isolated all build logs where the extracted project URL contained the substrings "fuzz" or "corpus". To ensure that we were not

[8] https://github.com/guidovranken/cryptofuzz-corpora

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 [55, 61], 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 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 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 project[9] because Facebook officially shut down the project and sold it to a private

[9] https://github.com/libra/libra

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 project [10] 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 dashboard [11], 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.

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. [42] 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. [47], 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 3) and the first passing build (build log 6 in Figure 3) following the initial build failure as shown in Figure 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.

[10]https://github.com/ClickHouse/ClickHouse
[11]https://oss-fuzz-build-logs.storage.googleapis.com/index.html

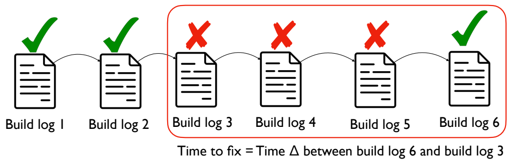


Fig. 3. Approach to calculate the time to fix a build failure in a failing build sequence.

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 activites 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 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 4 shows an overview of the time required to fix failing fuzzing builds in hours. As shown in Table 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 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

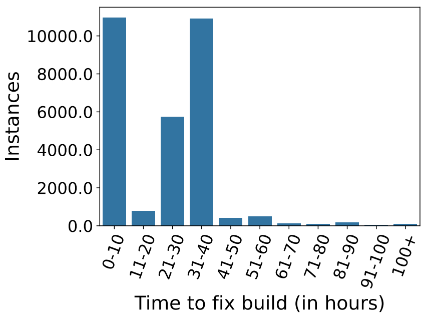


Fig. 4. Time to fix a fuzzing build failure in hours

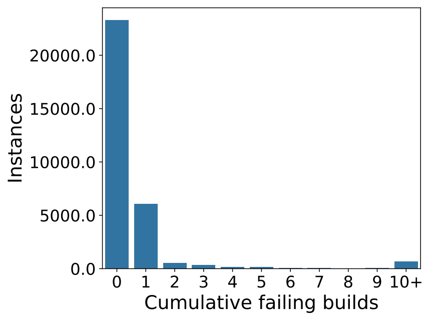


Fig. 5. Number of subsequent failing builds

Table 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)		
≤ 12h	≤ 24h	≤ 48h	0	≤ 1	>10
40.00%	75.00%	95.00%	73.85%	93.00%	2.12%

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. Our results further confirm that most projects are keeping up with their fuzzing activites when we calculate the proportion of failing builds with more than 10 subsequent falling builds. As shown in Table 3, we find that in only 2.12% cases builds fail more than 10 times before it is fixed. Figure 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.

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. We chose to establish our taxonomy from the ground up to avoid missing new types of build failures due to referring to taxonomies proposed in previous work. Moreoever, we wanted to avoid biasing the annotators during the labeling process by making them aware of categories established in previous studies.

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 anaylsis, 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 [33, 41], we designed the manual labeling methodology as follows.

Establish the base taxonomy. The three annotators first conducted a trial run by individually 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 merged the labels of all three iterations into one set then calculated Cohen's Kappa [9] to measure the inter-agreement ratio between the authors. The resulting coefficient is a value ranging between -1 and +1 and has been used in past software engineering studies to calculate the agreement during labeling tasks [1]. A value of 0 implies that the agreement ratio was due to chance, a value higher than 0 indicates that the agreement is higher than what we would expect from pure chance, and a lower value means that the agreement is lower than what we would expect from pure chance. In our case, we obtained a 0.78 coefficient which indicates that the annotators had a good inter-agreement during labeling.

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

Table 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.

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

Table 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 [38]	Coarse-Grained Prev. Study [54]
Environment issue (114)	*RC1*: Compiler issues (65)	Others	crash
	RC2: Coverage file and directory issues (26)		
	RC3: Project environment issues (10)		crash
	RC4: Network issues (7)	Server connection error	crash
	RC5: Hardware issues (5)	Memory issue	crash
	RC6: Permission issues (1)	Execution permission error	
Corpus related issues (99)	*RC7*: Corpus related issues (99)		
Issues downloading external resources (97)	*RC8*: Issues downloading external resources (97)		
Project dependency issues (58)	*RC9*: Project dependency issues (58)	Dependency resolution	dependency
Build and configuration issues (58)	*RC10*: Project configuration and build file issues (28)	Parse	buildconfig
	RC11: Coverage build configuration and file issues (25)	Parse	buildconfig
	RC12: Fuzzer build script issues (5)		
Project source code related issues (56)	*RC13*: Source code related project compilation errors (39)	Compiler error	compile
	RC14: Missing source code files (17)	Missing file	
Command and argument related issues (48)	*RC15*: Command and argument related issues (48)		
Runtime issues while fuzzing (43)	*RC16*: Runtime issues while fuzzing (43)		
Not enough information (43)	*RC17*: Not enough information (43)		unknown
Fuzz target issues (47)	*RC18*: Sanitizer errors (24)	Other run-time error	
	RC19: Broken fuzz target (21)		
	RC20: Missing fuzz target (2)		
Miscellaneous (14)	*RC21*: Input causes unusual fuzzer crashes or behaviors (7)		
	RC22: Failing test cases (3)	Test run-time error	testfailure
	RC23: Missing OSS-Fuzz scripts (2)		
	RC24: Unusual crash from the target binary (1)	External executable error	
	RC25: Regression in the fuzzer causes build crash (1)		

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 breakages that 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 a 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 (e.g., 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 (e.g., 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 builds 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 #4781[12], fixing PR #4783[13]). 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 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, UndefinedBehaviorSanitizer, 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

[12]https://github.com/google/oss-fuzz/issues/4781
[13]https://github.com/google/oss-fuzz/pull/4783

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 "RC21: 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 (RC22) or OSS-Fuzz scripts required during the build process were missing (RC23). Finally, we found another unique case where the target binary crashed before any input could be sent to it (RC24). This case differed from RC16 (Runtime issues while fuzzing) 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. [55] 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 Rausch et al. [54] and Lou et al. [38] analyze build failures on a more general level without focusing on any specific context or root cause. Alternatively, other studies such as Kerzazi et al.'s [30] 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 failure 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 [38, 54, 55].

Our study however differs due to the additional layers of complexities brought by fuzzers which introduce new possible causes of failure not present in previous work. For example, while the taxonomy in the coarse-grained study led by Rausch et al. [54] 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 sources of failure related to using a corpus or configuring/running the fuzzing tool itself. Our taxonomy therefore offers an extended look of possible causes of failure over taxonomies proposed in previous work.

To highlight how our work relate to previous work (and extends it), in Table 4 we compare the results 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 [54] and the fine-grained taxonomy proposed by Lou et al. in their 2020 study [38]. 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 4, our taxonomy covers 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 project 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 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. [42] 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.

Our results can provide some insights into the build fixing process. 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 a build (e.g., OSS-Fuzz developers, projects developers, etc.) are also the people in charge of fixing it, we can 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 caused due to an issue on OSS-Fuzz' side 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.

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 June 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 projects were 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

[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

of thousands of dependent projects and users. As highlighted by the fuzzing experts surveyed in Nourry et al.'s previous study [47], 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.

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 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 [22, 39, 45, 62]. 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 [37, 56]. Dependency and configuration issues could also benefit from automatic dependency versioning repairs [58]. 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. [23] 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 classication 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* [49]. 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 [11] (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 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 easily 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.

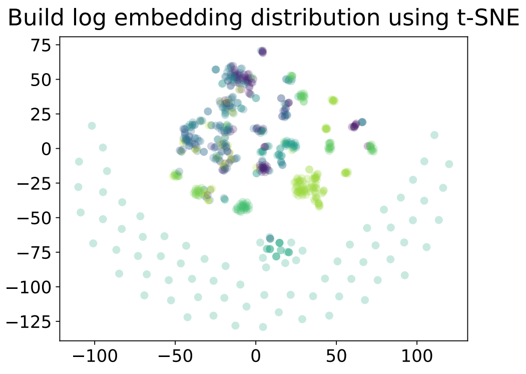


Fig. 6. Embedded log distribution using t-SNE

Confirming the presence of failure text patterns. To learn more about textual patterns of failure in the failed logs, we tried to find logs whose failure patterns were representative of a root cause. Since points (or logs) that are closer to each other on the t-SNE plots should have similar content, we decided to use the t-SNE coordinates to define a "centroid" for each root cause. We defined this theoretical center by calculating the mean coordinates of all points that shared the same root cause. We then calculated the euclidian distance between the centroid of each root cause and its corresponding points. Finally, we took the five points (or builds logs) that were closest to this center, manually reviewed them and wrote down the text patterns that allowed us to classify the failure. We repeated this process for the five most common root causes (RC1, RC7, RC8, RC9, RC12), as well as two root causes specific to fuzzing builds (RC16, RC19) since we wanted to know more about fuzzing specific patterns of failure. Table 5 shows failure patterns representative of the selected root causes. These failure patterns were extracted from logs that were closest to the centroid of each root cause.

Table 5. Table showing a representative build failure text pattern for the selected root causes and a corresponding example. The top part shows the failure pattern and the lower part shows a real example.

Label	Pattern	Example
RC1: Compiler issues	checking whether the C compiler works... no configure: error: in {location where compiler failed}: configure: error: C compiler cannot create executables	checking whether the C compiler works... no configure: error: in '/src/libxml2': configure: error: C compiler cannot create executables
RC7: Corpus related issues	[/corpus/{name of corpus}] End-of-central-directory signature not found. Either this file is not a zipfile, or it constitutes one disk of a multi-part archive.In the latter case the central directory and zipfile comment will be found on the last disk(s) of this archive.	[/corpus/fuzz_reader.zip] End-of-central-directory signature not found.Either this file is not a zipfile, or it constitutes one disk of a multi-part archive.In the latter case the central directory and zipfile comment will be found on the last disk(s) of this archive.
RC8: Issue downloading external resources	CommandException: No URLs matched. Do the files you're operating on exist?	CommandException: No URLs matched. Do the files you're operating on exist?
RC9: Project: Dependency issues	clang-14: error: no such file or directory: '{missing_dependency}'	clang-14: error: no such file or directory: '/src/librdkafka/mklove/deps/dest/usr/lib/libz.a'
RC15: Command and argument related issues	Bad syntax used for argument	[-] PROGRAM ABORT : [0mBad syntax used for -t[1;91m
RC16: Runtime issue while fuzzing	Error occured while running {name of fuzz target}: [...] {error code} libFuzzer: run interrupted; exiting	Error occured while running ./hevc_dec_fuzzer: INFO: Seed: 2661026770 INFO: Loaded 1 module 65 inline 8-bit counters): 65 [0x7d53fc, 0x7d543d), INFO: Loaded 1 PC tables (65 PCs): 65 [0x57c130,0x57c540), MERGE-OUTER: 16912 files, 0 in the initial corpus, 0 processed earlier MERGE-OUTER: attempt 1 MERGE-OUTER: attempt 2 MERGE-OUTER: attempt 3 ==28== libFuzzer: run interrupted; exiting
RC19: Broken Fuzz target	BAD BUILD: {name of fuzz target} [...] Broken fuzz targets ({number of broken targets}) {broken target 1} {broken target 2} ... {broken target N} ERROR: {X}% of fuzz targets seem to be broken	BAD BUILD: /tmp/not-out/llvm-isel-fuzzer–wasm32-O2 seems to have either startup crash or exit: [...] Broken fuzz targets 19 /tmp/not-out/llvm-opt-fuzzer–x86_64-gvn /tmp/not-out/llvm-opt-fuzzer–x86_64-irce /tmp/not-out/llvm-isel-fuzzer–aarch64-O2 /tmp/not-out/llvm-opt-fuzzer–x86_64-simplifycfg ... tmp/not-out/llvm-opt-fuzzer–x86_64-loop_predication /tmp/not-out/llvm-isel-fuzzer–wasm32-O2 ERROR: 70.37037037037037% of fuzz targets seem to be broken.

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 6. 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 [34, 53, 57]. Additionally, a previous study has also shown that RF classifiers can outperform other types of models in many situations [19]. During our experimentation phase, we also tried using an XGBoost classifier and an MLP classifier to see if it would outperform a random forest classifier. We chose these two alternatives because MLP classifiers are widely used for text embedding classifications [12, 24, 31] and XGBoost has proven to perform well for text classification tasks[6, 35]. While the results were comparable between the three classifiers, the random forest algorithm performed the best out of them. 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. We used a low maximum depth in order to minimize the chances of overfitting the data. All other parameters including the number of trees (100 trees), and the impurity criterion (Gini) 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 6 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

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

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.

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 [20] and CI/CD services that support fuzzing such as Travis CI [8], Circle CI [7] and Jenkins [28] 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 generalizibility 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. [38] 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. [61] 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.

Rausch et al. [54] 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. [55] 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 [15, 46] 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 [54], build systems [38], 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 provides 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, Ding et al. [13] conducted one of the first empirical studies on OSS-Fuzz by mining and analyzing over 23,000 bugs collected from the official bug tracker [21]. In that study, Ding et al. [13] 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. Keller et al. [29] also investigated the lifecycle of bugs found by OSS-Fuzz. Their findings align with Ding et al.'s findings that developers are generally quick to fix bugs once the issues are detected by OSS-Fuzz fuzzers. Keller et 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. [5]. 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 [42], 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 CONCLUSION

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. We also aim to explore how automation techniques can be used to abstract some of the complexity of starting and maintaining fuzzing activities over time. Specifically, we aim to investigate the process of fixing fuzzing build failures in order to deliver a practical solution or tool that can automate fuzzing build diagnosis and repair.

9 REPLICATION

To facilitate future work, we have made available online the result of our manual labeling process in our replication package [48].

10 ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of: (1) JSPS for the KAKENHI grants (JP21H04877, JP22K18630), and Bilateral Program grant JPJSBP120239929; (2) the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship; and (3) JST for the PRESTO grant JPMJPR22P3.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, 385–395.
[2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 748–758.
[3] Craig Beaman, Michael Redbourne, J. Darren Mummery, and Saqib Hakak. 2022. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers and Security* (2022).
[4] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* (2021), 79–86.
[5] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 713–724.
[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 785–794.
[7] Circle CI. 2011. Circle CI. https://circleci.com/
[8] Travis CI. 2011. Travis CI. https://www.travis-ci.com/
[9] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* (1960), 37 – 46. https://api.semanticscholar.org/CorpusID:15926286
[10] National Vulnerability Database. [n. d.]. Log4Shell Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2021-44228

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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*. Association for Computational Linguistics, 4171–4186.
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *North American Chapter of the Association for Computational Linguistics*.
[13] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *Proceedings of the 18th International Conference on Mining Software Repositories*. Association for Computing Machinery, 131–142.
[14] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, 254–264.
[15] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. 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*. 463–474.
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. USENIX Association.
[17] Go Fuzzing. 2022. Go Fuzzing Documentation. https://go.dev/doc/security/fuzz/
[18] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. 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*. Association for Computing Machinery, 1330–1342.
[19] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *Proceedings of the 37th International Conference on Software Engineering*. Association for Computing Machinery, 789–800.
[20] GitLab. 2019. GitLab acquisition of Fuzzit. https://about.gitlab.com/blog/2020/10/22/fuzzit-acquisition-journey/
[21] Oss-Fuzz Google. 2023. Chrome Monorail Bug Tracker. https://bugs.chromium.org/p/oss-fuzz/issues/list
[22] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 1078–1089.
[23] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering* (2018), 3248–3280.
[24] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 328–339.
[25] Ahmad Humayun, Yaoxuan Wu, Miryung Kim, and Muhammad Ali Gulzar. 2023. NaturalFuzz: Natural Input Generation for Big Data Analytics. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, 1592–1603.
[26] Google Inc. [n. d.]. OSS-Fuzz Architecture. https://google.github.io/oss-fuzz/architecture/.
[27] Synopsys Inc. 2014. Heartbleed Vulnerability. https://heartbleed.com/
[28] Jenkins. 2011. Jenkins.io. https://www.jenkins.io/
[29] Brandon Keller, Andrew Meneely, and Benjamin Meyers. 2023. What Happens When We Fuzz? Investigating OSS-Fuzz Bug History. In *Proceedings of the 20th International Conference on Mining Software Repositories*. Association for Computing Machinery, 207–217.
[30] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *International Conference on Software Maintenance and Evolution*. IEEE, 41–50.
[31] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations*.
[32] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 475–485.
[33] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2021. A Qualitative Study of the Benefits and Costs of Logging From Developers' Perspectives. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2858–2873.
[34] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* (2017), 1831–1865.
[35] Qian Li, Hao Peng, Jianxin Li, Congying Xia, Renyu Yang, Lichao Sun, Philip S. Yu, and Lifang He. 2022. A Survey on Text Classification: From Traditional to Deep Learning. *ACM Transactions on Intelligent Systems and Technology* (2022), 1–41.

[36] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* (2018), 1199–1218.
[37] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-Driven Build Failure Fixing: How Far Are We?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.
[38] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. 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)*. 617–628.
[39] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, 106–117.
[40] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47 (2021), 2312–2331.
[41] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. 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, Article 72 (2019).
[42] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, 141–150.
[43] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1615–1629.
[44] Wen Ming, Wang Yongcong, Xia Yifan, and Jin Hai. 2023. Evaluating seed selection for fuzzing JavaScript engines. *Empirical Software Engineering* (2023).
[45] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 439–451.
[46] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451.
[47] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023).
[48] Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, and Yasutaka Kamei. 2024. Replication package for the paper: My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures. https://github.com/posl/FuzzingBuildLogs-ReplicationPackage
[49] OpenAI. 2022. GPT text-embedding-ada-002. https://openai.com/blog/new-and-improved-embedding-model
[50] Google OSS-Fuzz. 2016. OSS-Fuzz. https://google.github.io/oss-fuzz/
[51] Google OSS-Fuzz. 2023. Oss-Fuzz build logs storage api. https://oss-fuzz-build-logs.storage.googleapis.com/index.html
[52] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. *ACM Transactions on Software Engineering and Methodology* (2023), 1–26.
[53] Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. 2021. Evaluating the impact of falsely detected performance bug-inducing changes in JIT models. *Empirical Software Engineering* (2021).
[54] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. 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*. Association for Computing Machinery, 345–355.
[55] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, 724–734.
[56] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* 28 (2019), 1–29.
[57] Anurag Verma and Xiaodai Dong. 2016. Detection of Ventricular Fibrillation Using Random Forest Classifier. *Journal of Biomedical Science and Engineering* (2016).
[58] Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. 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*. 198–210.

[59] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. 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*. Association for Computing Machinery, 1037–1049.

[60] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 1634–1645.

[61] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. An Empirical Study of Build Failures in the Docker Context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, 76–80.

[62] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. 2023. 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.

[63] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. 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*. 1135–1147.

[64] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54 (2022), 1–36.