

An Empirical Study of Fuzz Harness Degradation

Philipp Görz
Joschua Schilling
Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany

Marcel Böhme
MPI-SP
Germany

Abstract

The purpose of continuous fuzzing platforms is to enable fuzzing for software projects via *fuzz harnesses*—but as the projects continue to evolve, are these harnesses updated in lockstep, or do they run out of date? If these harnesses remain unmaintained, will they *degrade* over time in terms of coverage achieved or number of bugs found? This is the subject of our study.

We study Google’s OSS-Fuzz continuous fuzzing platform containing harnesses for 510 open-source C/C++ projects, many of which are security-critical. A harness is the glue code between the fuzzer and the project, so it needs to adapt to changes in the project. It is often added by a project maintainer or as part of a, sometimes short-lived, testing effort.

Our analysis shows a consistent overall fuzzer coverage percentage for projects in OSS-Fuzz and a surprising longevity of the bug-finding capability of harnesses even without explicit updates, as long as they still build. However, we also identify and manually examine individual cases of harness coverage degradation and categorize their root causes. Furthermore, we contribute to OSS-Fuzz and Fuzz Introspector to support metrics to detect harness degradation in OSS-Fuzz projects guided by this research.

1 Introduction

In recent years, fuzzing has become a popular technique among security practitioners. The method has an impressive ability to find software faults, uncovering over 10,000 vulnerabilities and 36,000 bugs across 1,000 open-source software projects via Google’s OSS-Fuzz project as of August 2023 [22, 46]. To fuzz software projects efficiently and effectively, they must be correctly integrated into the fuzzer. In practice, this is achieved by manually creating so-called *fuzzing harnesses*, which allow a fuzzer to run and provide input to the program under test correctly. As the creation of well-fitting harnesses is critical for fuzzing but requires substantial manual effort [36, 42], various attempts [6, 14, 28–31, 39, 54–57] have been made to automate the process.

The correct integration of important open source projects and the creation of suitable harnesses are so critical for successful fuzzing campaigns that Google’s OSS-Fuzz project offers a financial reward system for developers that integrate their projects with OSS-Fuzz via harnesses. Furthermore, Google specifically rewards harnesses that cover at least

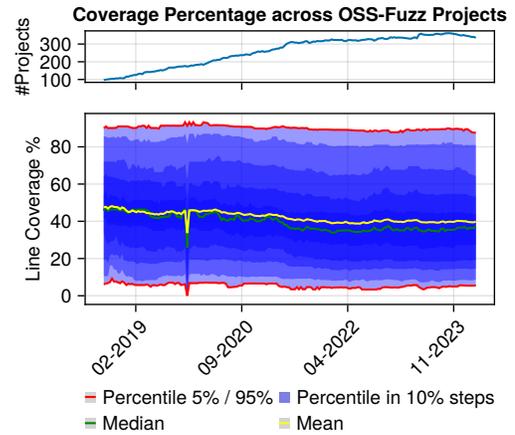


Figure 1. Fuzz coverage across C/C++ projects in OSS-Fuzz.

50% of the project’s source code [21]. The importance of well-fitting harnesses is also highlighted, especially in long-lasting fuzzing campaigns such as OSS-Fuzz, as a coverage plateau during fuzzing is often caused by an inadequate harness design [17].

However, how does the effectiveness of the provided harness develop after the initial onboarding over time? Modern software projects are dynamic, as they are usually in continuous development and rely on agile development methods [4, 27, 51]. The effect of varying rates of evolution within an ecosystem of software projects is well-studied under software degradation [2, 5, 8, 10]. Yet, the phenomenon of harness degradation remains overlooked, particularly in the fuzzing community, despite the potential decrease in capability to find security-critical bugs in important projects.

In this paper, we conduct the first comprehensive empirical study on the scale, impact, and causes of harness degradation in real-world software. To this end, we analyze the coverage achieved and bugs found from on-boarding until today for 29,019 harness versions across 433 projects integrated into OSS-Fuzz.

We find that the coverage of C/C++ projects in OSS-Fuzz stays quite stable on average, as can be seen in Figure 1, which indicates a surprising longevity of fuzz harnesses, as long as the harnesses still build. However, we also find that there is still a high variance, as there are fuzz harnesses that

improve or degrade over time. Unsurprisingly we can confirm that projects with less coverage find fewer bugs, as for projects with less than 35% coverage a bug is found for every 14,474 changed lines of code, but projects with higher coverage find a bug for every 3,985 changed lines. In fact, projects that manage to improve their coverage are rewarded with a burst of found bugs. Furthermore, we investigate the causes for coverage drops, we find that these are usually caused by added code (project internal or external), partial build failures, and errors during coverage measurements. Finally, we are working with the Fuzz Introspector and OSS-Fuzz teams to implement metrics to detect harness degradation, to avoid cases where maintainers believe their projects to be well fuzzed while in reality there has been a silent degradation.

In summary, we make the following key contributions towards investigating fuzz harness degradation in practice:

- We systematically analyze and quantify harness degradation and the effect of harness updates in OSS-Fuzz based on coverage percentage and bug-finding capability.
- We manually analyze cases of harness coverage degradation in OSS-Fuzz and categorize them into common causes.
- Based on these causes, we contribute to Fuzz Introspector introducing new metrics developed to alert of possible harness degradation.
- We provide our complete dataset¹, scraping code, case study notes, and analysis notebook² to facilitate further research on this topic.

2 Background

We begin by presenting the essential background information. Focusing on what fuzzers and fuzzing harnesses are.

Fuzzers. Fuzzers are dynamic testing tools that execute a target program with randomized inputs. This method has established itself as an effective and practical way to detect bugs and security vulnerabilities. Modern general-purpose fuzzers generally use an evolutionary approach, where inputs are slightly modified (“mutated”). The inputs that perform well, such as reaching new parts of the code or triggering new behavior, are kept to be mutated again. To kickstart this process, it is beneficial to have a comprehensive initial set of inputs that the fuzzer can start with, this is also called a *seed corpus*. While the set of inputs found during fuzzing is called a *corpus* [35, 49, 53].

Fuzzing Harnesses. The term fuzzing harness (fuzz harness for short) is derived from software testing terminology, where a test harness describes a collection of test stubs and test drivers, which are required to execute a test suite [9]. Similarly, a fuzz harness defines the standardized entry point

into the program under test. As test stubs are less relevant for fuzzing, the term fuzz driver is often used interchangeably with fuzz harness [54].

To provide a flexible way of generating new inputs, most general-purpose fuzzers settled on a standardized function [38]. This function takes a byte vector and its length as inputs. That byte vector needs to be passed to the target program in a syntactically and semantically correct way, which is the goal of the fuzz harness. Thus, a typical fuzz harness function transforms this byte vector into data structures and calls functions from the target program. If needed, the harness includes calls to initialization and cleanup functions. So, the harness function must encode the business logic required to interact with the target application correctly. Thus, the harness needs to be updated to keep up with potential changes in the program’s code base. This makes fuzz harnesses interesting potential points of failure, which we study in this paper.

OSS-Fuzz. [22] OSS-Fuzz is a project developed and maintained by Google that provides open-source projects with the infrastructure to fuzz their code. However, this requires open-source projects to integrate with the OSS-Fuzz infrastructure. This entails developing fitting fuzz harnesses for their projects, setting up project-specific build configurations, and providing meta information [19]. To incentivize open-source projects to integrate with OSS-Fuzz, a bounty program is offered [21].

OSS-Fuzz supports projects written in several programming languages and supports the x86_64 or i386 architectures. After successful integration with OSS-Fuzz, the current version of a project is continuously built and fuzzed daily. OSS-Fuzz relies on a scalable, distributed infrastructure to use different fuzzers and sanitizers and provide automatic reporting for the project’s maintainers. This includes detected bugs and detailed daily coverage results to facilitate further fuzzing [19]. To our knowledge, the OSS-Fuzz project also keeps the corpus of previous runs available for the project maintainers and uses it, or a subset, as the seed corpus for future runs. Finally, crucially for our research, we collect publicly available data from the OSS-Fuzz reaching from 2016 – 10 to 2024 – 10, which enables a long-term study such as the one conducted in this paper.

3 Fuzz Harness Degradation

To evaluate the impact of fuzz harness degradation, we observe two central metrics of fuzzer evaluation, code coverage and bug-finding capability [33, 45]. As we are interested in harness performance over time, we consider these metrics over different harness versions, as a harness evolves. Furthermore, we are interested in the reasons for harness degradation. To this end, we manually perform case studies on code coverage drops and classify common causes. Finally, based on these causes we implement metrics to warn maintainers

¹Dataset: doi:10.5281/zenodo.14000867

²Repository: <https://github.com/CISPA-SysSec/fuzz-harness-degradation>

Table 1. Number of data points for each step of the data preparation phase.

	Raw	Clean	Filter
① Projects	510	491	433
② Coverage	665,165	574,840	527,085
③ Commits	8,297,824	2,556,947	2,495,172
④ Harness Changes	42,412	29,923	29,019
⑤ Monorail Bugs	68,486	53,460	50,493
⑥ Crash Revisions	104,494	83,863	79,530
⑦ Fuzz Introspector	975,213	712,094	652,495
⑧ Fuzz Intro. Harness	1,307,652	929,224	894,238

of possible harness degradation, with the future goal of studying the effect of these new metrics. In total, we investigate four research questions:

- RQ1 What are the immediate effects of harness updates?
- RQ2 What is the rate of degradation of code coverage over time?
- RQ3 Does the bug-finding capability of harnesses degrade over time?
- RQ4 What are common causes for coverage drops?
- RQ5 What are practical ways to detect harness degradation?

3.1 Data

To answer these research questions, we require data that covers large parts of the lifespan for fuzz harnesses, optimally containing many fuzz harnesses and versions. Additionally, we need bug data and the availability of artifacts to study individual cases of harness degradation. To our knowledge, OSS-Fuzz is the only project that meets these criteria. Thus, to investigate our research questions, we collect and pre-process several datasets related to OSS-Fuzz to create one cohesive dataset³ as follows.

We prepare each dataset in three steps, corresponding to the three columns in Table 1. First, we collect all available data in the **Raw** step. Next, we remove all unusable or irrelevant data in the **Clean** step. This includes data that does not fall into the date range when the project was first added to OSS-Fuzz or up to a week after the last commit to a project and other cleanups explained individually below. Finally, we filter out projects for which we do not have valid harness data in the **Filter** step. As each dataset also requires individual actions, we specify them separately as below. The number of datapoints after each step can be found in Table 1.

3.1.1 ① Projects. Shows the number of projects for each step, based on whether we could collect all the required data. Note that only C/C++ projects are used in our dataset.

3.1.2 ② Coverage. The daily coverage reports per project and harness. **Raw** OSS-Fuzz publishes one coverage report per day per project and harness. We do not count a datapoint if no data is available on a particular day. However, we do track the absence of data. This data is collected from the HTML coverage report⁴, which contains line, function, and region coverage as provided by Clang’s source-based code coverage feature [37]. **Clean** We removed around 20k empty coverage reports before the projects were first added to the OSS-Fuzz repo.

3.1.3 ③ Commits. The commits to the default branch. **Raw** We try to clone every git project based on the URL provided in the OSS-Fuzz project .yaml under the key main_repo. If the repository cannot be cloned we exclude it from our analysis, which is the case for non-git-based repositories and projects that specify a GitHub organization instead of a repository. In total, we gather commit data for 491 projects. We collect every commit to the project’s default branch, including the number of added/removed lines per file, as provided by git. Some projects consist of multiple repositories that include dependencies and sometimes fuzzing-related code. Commits to these separate repositories are not included in the commit count of a project, as this would require analyzing the Dockerfile and differentiating between libraries and project code that happens to be in a separate repository. **Clean** We remove all commits that happened before the project was added to OSS-Fuzz or are no longer part of OSS-Fuzz.

3.1.4 ④ Harness Changes. Commits that are classified as a harness change. **Raw** We use heuristics based on the commits in OSS-Fuzz and the project’s repository to classify the commits that introduce or update fuzzer harnesses. If the commit is part of OSS-Fuzz, we include all commits that change a file that is not the project.yaml, Dockerfile (or Jenkins file, which was used earlier) to avoid simple changes of metadata or dependency updates to count as a harness update.

For commits to the project’s repository, we include changes to files with C/C++ file extensions, where the file path includes "fuzz", however, excluding "fuzzy" as some projects use fuzzy logic. Additionally, changes to C/C++ files containing the fuzzing functions LLVMFuzzerTestOneInput and LLVMFuzzerInitialize are also included as harness changes. This analysis is implemented with tree sitter [1]. **Clean** As for the commit data, remove all data points before the project’s introduction into OSS-Fuzz. **Filter** There is one situation in which these heuristics can produce false positives: If the harness is part of a larger file containing normal code, these changes are counted as harness changes. As the code in that

³Dataset: doi:10.5281/zenodo.14000867

⁴ https://storage.googleapis.com/oss-fuzz-coverage/<project>/reports/<date>/linux/file_view_index.html
i.e.: https://storage.googleapis.com/oss-fuzz-coverage/sudoers/reports/20250226/linux/file_view_index.html

file can also contain support functions for the harness, we cannot simply constrain our analysis to the harness function. During our manual analysis, we detected only one project where this is the case, which we filtered out. To avoid false negatives, that is, a harness update is not detected, we investigate all projects with suspicious gaps of harness updates compared to commits. We detected two projects that moved fuzzing harnesses into a separate repository, which we filtered out. Additionally, two projects only contain fuzzing examples; as these are not relevant to our study, we filter them out as well.

3.1.5 ⑤ Monorail Bugs. Monorail is a issue tracker⁵ containing bugs found via fuzzing or build of the project in OSS-Fuzz. Note that Monorail has been deprecated and replaced with a new issue tracker⁶ for which we do not support data gathering, our last datapoint is on the 2024 - 9 - 11. [Raw] Three types of bugs are reported: security bugs, general bugs, and build failures (that break the project’s fuzzer build on OSS-Fuzz). We collect meta information on bugs such as the date of the bug, labels, and comments. Note that security-relevant bugs are hidden for 90 days, so we cannot include these vulnerabilities in our dataset. [Clean] We remove all bug reports for projects that do not have harness changes. We also checked that no reports before integration into OSS-Fuzz are available. Unexpectedly, there are 25 such reports, which we remove from the dataset. We believe that these were manually created.

3.1.6 ⑥ Crash Revisions. Date (and commit) ranges for bugs in Monorail. [Raw] For many bugs in Monorail, automated comments are added that specify the revision which was used when this bug was found or during what date (commit) range a regression is introduced. Furthermore, the project and fuzzer version in which the crash was found are specified. These specified crash revisions are generated for security and general bug issues. For fixed bugs, a time (and commit) range is added to indicate when the fix occurred. [Clean] The cleaning is done identically to the procedure applied to monorail bugs.

3.1.7 ⑦ Fuzz Introspector. Fuzz Introspector reports created daily. [Raw] Fuzz Introspector [23] is a tool to analyze the quality of fuzzing and identify potential issues with the fuzzing setup. Most OSS-Fuzz projects have already been integrated with Introspector in the Open Source Fuzzing Introspection project [24], whose reports we include in our analysis. We collect all JSON reports for each day⁷. Note that these are only available if the project can be built with the Introspector instrumentation. While we cache the full

JSON report, we collect the following data: harness count, total complexity, complexity reached, total functions, reached function count, and code coverage function count. [Clean] All reports before the project is added to OSS-Fuzz are removed. While there are some, they are all empty.

3.1.8 ⑧ Fuzz Intro. Harness. Daily Fuzz Introspector reports per harness. [Raw] Additionally to the combined data discussed above, Fuzz Introspector also provides data on a per-harness basis. If a report is available, we collect for each harness per day the following data: the harness name, corpus size (the number of inputs found during fuzzing), the number of functions reached during fuzzing, and statically calculated during the instrumentation by Fuzz Introspector: the total number of reachable basic blocks, total reachable cyclomatic complexity, the total number of reachable files, and the number of reachable functions. [Clean] Again, we remove all reports of the time before a project is included in OSS-Fuzz.

Based on this combined dataset we investigate the research questions. Beginning with an analysis of coverage changes over time.

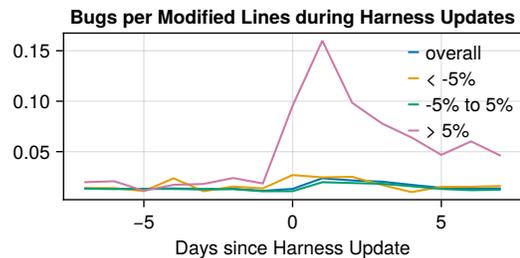


Figure 2. Bug finding relative to harness updates.

3.2 RQ1: What are the immediate effects of harness updates?

We hypothesize that harnesses slowly degrade over their lifetime and that this can be counteracted by maintenance in the form of harness updates. Which cause an immediate increase in coverage and a burst of new bugs after the introduction of the new harness version. To verify our hypothesis, we study the specific events of harness updates and the immediate effects these changes have.

First, we start with the effects on coverage by comparing the maximum coverage in the 7 days before a harness update, with the maximum coverage after a harness update.

Our results show that contrary to our original hypothesis, harness updates do not have a vastly positive effect on coverage. Instead, we find a rather low mean coverage increase of only 0.26%, while the median coverage change is exactly at 0.0%. We observe a rather high variance, with a standard deviation of 4.34, which we attribute to the stark differences between projects in our data set.

⁵<https://bugs.chromium.org/p/oss-fuzz/issues/list>

⁶<https://issues.oss-fuzz.com/issues>

⁷<https://oss-fuzz-introspector.storage.googleapis.com/<project>/introspector-report/<date>/summary.json>

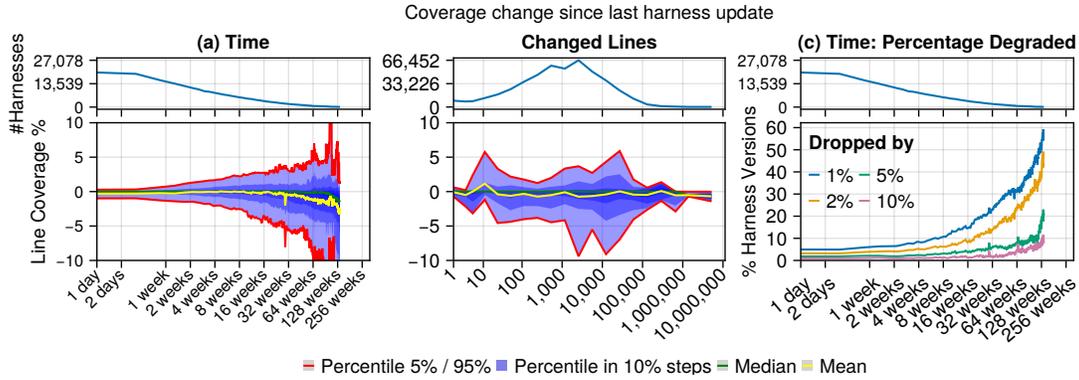


Figure 3. Relative coverage change since a harness update, aggregated across all projects.

To filter out minor changes, we divide harness updates into three categories. We consider a harness improved, if the project coverage increases at least 5%. Similarly, we consider a harness degraded if coverage decreases by 5% or more. Everything in between is considered as a maintained harness, which is the predominant category with 92.36% of harness updates. While 5.54% improve and 2.1% of harness updates actually degrade coverage.

To analyze the effects these types of harness updates have on the bug finding capability of the projects, we collect the amount of bugs detected in the week before and after a harness update. This data can be seen in Figure 2 which shows the code churn adjusted rate of bugs detected for the earlier mentioned categories of harness updates. The data shows that our hypothesis of initial bug bursts can be trivially confirmed for improved harnesses. For degraded and maintained harnesses, the correlation is less pronounced.

Harness updates have a surprisingly minor effect on coverage. We find only a slight mean increase of 0.26%. We can see initial bug bursts after harness updates, especially when coverage increases.

3.3 RQ2: What is the rate of degradation of code coverage over time?

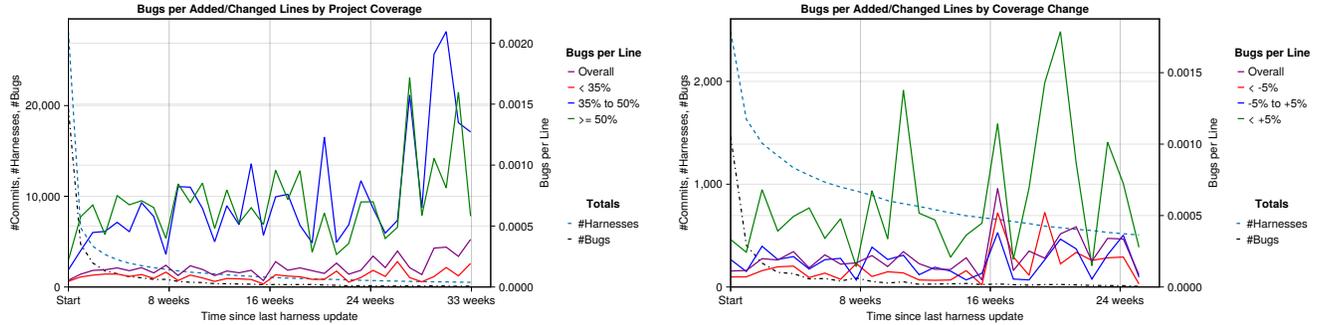
We hypothesize that the fuzzing coverage degrades over time if the project changes faster than the fuzzing harness is maintained. To investigate this hypothesis, we need to identify changes in the project code related to fuzzer maintenance, which we do as described in Section 3.1.4. We show the coverage changes since the last harness update in Figure 3. Note that we use the highest coverage found during the first three days, to compensate for possibly slow coverage saturation and delays until the fuzzer uses a new harness update. We want to emphasize that we are interested in the degradation of a harness over time, not the impact of the harness update itself (which we discussed in Section 3.2). Furthermore, we exclude harnesses that completely break and do not count

them as zero coverage, which would strongly influence the results. Note that this influence has already been investigated by Nourry et al. which found that around 12% of builds to be broken over time [41].

As seen in Figure 3(a), we can only observe a slight downward trend for the mean coverage. However, there is a large variance as the harnesses get older. To avoid showing possibly misleading data, we cut off the tail end where #Harnesses falls below 100. We expected the downwards trend to be much larger and therefore consider the results a surprising, but positive result. While it is expected that some projects suffer a decrease in coverage over time, it is, in our opinion, unexpected to see a similar number of projects that have an upswing over time. This keeps the median rather neutral and just when the mean seems to reach a significant downtrend, we are reaching the limit of our data.

How is it possible that the coverage percentages increase without a harness change? We have observed two common scenarios: Newly added code that can already be reached by the existing harnesses and thus is directly covered as soon as it is added to the project. This leads to a larger share of covered code, thus increasing the relative line coverage. The second scenario includes bug fixes or code changes that unlock previously blocked parts of a program. This enables a fuzzer to get deeper into the code and increase coverage without the need for harness updates. However, as we are concerned with harness degradation and not improvements, we leave a more thorough investigation to future research.

For the first plot, we use time as a simple proxy for project churn. However, the number of changed lines is a more precise approximation, which is shown in Figure 3(b). Similarly to the analysis by time, the data shows a high variance and a neutral mean and median result. Indeed, we also investigated a stratified analysis separating projects by activity and updates of the fuzzing harness. However, looking at these subsets of the data, the results remain similar, and the variance is still very high. Thus, we provide these results as part of the dataset and do not discuss them further.



(a) Bugs per changed lines of code for projects with: under 35%, between 35% and 50%, and over 50% coverage. (b) Bugs per changed lines for projects where the maximum coverage changed from the past to the next 7 days. Coverage changes are divided into >5% coverage increase, >5% coverage decrease and everything in between.

Figure 4. Found bugs per changed lines since the last harness update

This result likely stems from the maturity of the projects in OSS-Fuzz, as only relatively mature projects with a significant user base are added to OSS-Fuzz. For less mature projects with more active development, we would expect to see significantly more degradation. Additionally, this highlights the effectiveness of fuzzers as this result demonstrates the ability of fuzzers to successfully cover code influenced by code churn without manual intervention.

While our analysis does not show an overall coverage decline due to harness degradation in the mean and median over all projects, this does not mean that no harnesses in OSS-Fuzz are affected by harness degradation. To measure the number of harnesses that decrease in coverage over time, we instead visualize the number of harnesses that fall below a threshold compared to their initial coverage. This data can be seen in Figure 3(c). The data shows that around 5% of harness versions degrade after half a year.

The coverage over time is surprisingly stable for OSS-Fuzz projects, mainly due to the large number of stable projects. However, individual projects can still have large increases and decreases in coverage, where 5% or more decreased coverage is reached by 5% of projects after half a year.

3.4 RQ3: Does the bug-finding capability of harnesses degrade over time?

We hypothesize that if a harness is not kept up to date, its bug finding capability will decrease over time due to project churn.

To investigate whether harnesses that are not updated find fewer bugs over time, we discuss Figure 4. As previously, these plots are relative to the time a harness has been updated. A harness counts as active until it is updated, at which point a "new" harness version starts again at the beginning. This

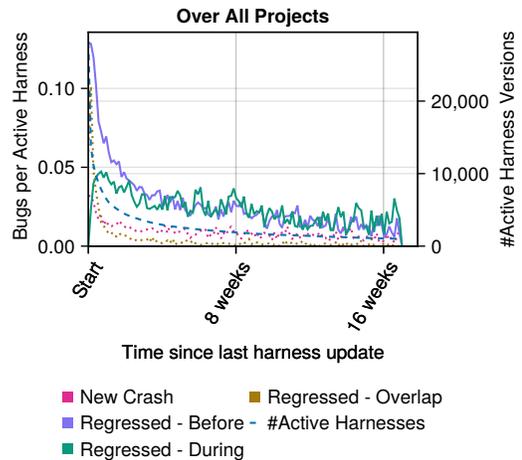


Figure 5. Bugs by crash revision type over time since the last harness update across all projects.

is why the number of harnesses reduces over time. Note that we again cut off the plot when reaching 500 harnesses to avoid misleading results.

In Figure 4a we can see the ratio of bugs found per modified (added/changed) line of code. As established in previous research [13], coverage has a strong correlation with the ratio of bugs found, which means that projects with good coverage are able to uncover more bugs via fuzzing. However, according to our data this relationship does not seem to hold linearly as projects with coverage above 35% perform rather similarly to projects above 50% coverage. This can have a number of reasons, maybe the used fuzzers just reach their limits or projects with over 50% coverage could also have more thorough tests outside of fuzzing, thus, reducing the relative number of bugs found. In any case, projects below 35% coverage, show a significant decrease in bug detection performance, as expected.

Notably, our data does not show a significant decrease in code churn-adjusted bug finding capability, over the first half year of a harness version after which our data starts to be too sparse. We again see that coverage plays a large role even if not as a relative increase.

Next, we are also interested in whether changes to the relative coverage percentage during the lifetime of a harness version changes the ratio of bugs found. We already observed a positive effect by harness updates in Section 3.2. However, does increasing coverage through project updates alone also improve bug finding? The code churn adjusted bug ratio split by recent coverage changes is shown in Figure 4b. It shows the number of bugs found per modified line, split by whether coverage increased (>5%), decreased (>5%) or stayed neutral (in between) comparing the maximum code coverage of the past and next 7 days throughout the lifetime of a harness. Again, the data shows bug burst for coverage increases, so the improved bug finding is not necessarily linked to a harness update. Notably, this is also true in the other direction, where coverage reductions also severely reduce bug finding capability, even when compared against no coverage change. Again confirming that coverage changes also strongly influence bug finding capability, even over the lifetime of harnesses. In summary, we can see that the fuzzing performance overall remains quite effective. On average, this remains true, even when the projects keep evolving without updating the harness.

We now investigate the second part of the hypothesis, that bugs are found due to updating the harness, which the earlier version misses. Looking at the crash revision data as described in Section 3.1.6, we can differentiate between two types of bugs: First, bugs that existed during the previous harness version but were found only after an update. The second category is bugs, which are introduced and found, while the harness version is already active. To create this separation, we group all Monorail bugs by the first crash revision mentioned in their reports. Some reports have "Fixed" as the first entry, these are excluded from analysis, as we do not know when these bugs were found. Similarly, bugs without crash revision data are also excluded. This leaves the revision types "New Crash" and "Regression". A "New Crash" revision contains the version fuzzed when the bug was found. "Regression" contains a range of possible commits where this bug was introduced, which we separate into three groups those where the range ends before the current harness version ("Before"), where it overlaps with the harness version ("Overlap"), and where the range is contained in the current harness version's lifetime ("During"). This visualization can be seen in Figure 5. Note that this figure is adjusted for active harness versions, and we again cut at 100 active harnesses to remove a possibly misleading tail end of the data.

We can only use "Before" and "During" to answer whether a harness update was required to find missed bugs. That is because "New Crash" does not give us information as to when

the bug was introduced, and for "Overlap" it is uncertain if the bug belongs to the "Before" or "During" group. Still, for completeness, we show all four groups. We would expect bugs of type "Crash", "Before", and "Overlap" to have a large initial burst which then reduces over time as the new harness version is explored. "During" should reflect the frequency of new bugs being introduced, similar to "Crash". Moving on to the interpretation of figure Figure 5, we can mostly confirm these expectations. As expected, an initial bug burst in all categories is visible in the first days after the harness update. While "During" starts as expected from none, it also has a slight increase within the first days of the harness lifecycle compared to later points in time. Note, however, that this can be likely explained by the heightened project activity within the first days of the harness update. Noteworthy is also that the number of regression bugs is constantly vastly higher than that of new bugs. This confirms the results by Zhu and Böhme [58], who found that about 77% of new bugs discovered are regression bugs introduced with a recent commit.

We observe an initial bug burst once the harnesses are updated. This is caused by bugs not found by the previous harness version, as well as active development shortly after harnesses are changed. However, unexpectedly, the number of bugs found by a harness version does not decrease automatically due to degradation. Instead, the degradation is limited to projects with degrading coverage.

3.5 RQ4: What are common causes for coverage drops?

Even though our data shows that overall C/C++ projects in OSS-Fuzz maintain their coverage quite well, our results also show a significant decrease of bug finding capability when coverage drops. As such it is important to identify cases where this happens, especially when coverage decreases silently.

To evaluate causes for coverage drops, we must first identify relevant instances of harness degradation to study. Thus, we filter for coverage drops that reduce coverage by 5% points or more, comparing the month before and after the drop. We chose 5% points to get a clear signal that a harness degradation occurred and to keep the number of cases to study manageable. This filter will help us focus on cases that result in long-term harness degradation. Additionally, we filter for days with a 5% coverage drop, allowing us to focus the case studies on the days where the drop actually happened. However, several projects with unstable coverage exist, where these two filters still retain many uninteresting cases, as they are just repeated up and downswings. To remove these cases, we additionally filter for maximum coverage of the previous month, dropping by 5% over the following month.



Figure 6. Reports and coverage for Curl (a, b, c) and Serenity (d).

Note that we are excluding harnesses that no longer built at all. While this can be seen as an extreme form of harness degradation, which results in zero coverage, it is also easily detected, and thus not interesting to study. This results in 267 instances of harness degradation, all of which we manually analyze by comparing coverage reports on the day before and on the coverage drop, if available, we also include the Fuzz Introspector reports for these days. Additionally, during our investigation, we documented other related cases, reaching a total of 308 case studies. As part of our analysis, we categorize each instance into one of the following categories:

3.5.1 Harness Build Failure. (Cases: 24) We identified multiple instances where harnesses no longer build correctly. This can be caused by a dependency no longer building accurately, a misconfiguration of the compilation environment (such as compiler flags), or a variety of similar reasons that have been studied in more depth before [41, 42]. The cases we investigated do not lead to a full but only a partial build failure that still severely impact fuzzer performance. These usually surface as a small drop of the total number of covered lines but a large drop in the coverage percentage. Note that these cases are sometimes not reported to Monorail as they are typically configuration mistakes or errors handled by the build script, so a build is not even attempted; thus, a build failure cannot be reported.

For example, Curl [15] experienced around one year of ineffective fuzzing, see Figure 6(a, b, c), which started⁸ at the end of 2022 and lasted until the issue was resolved in early 2024. The reason was that OpenSSL [43] was no longer building⁹. However, that did not completely stop the build process, as some harnesses were still working.

3.5.2 Project Code Added / Code Churn. (Code added: 44, code churn: 24)

In these cases, the project is developed further, increasing code size or changing semantics (Code Churn). However,

the harnesses are not updated to reflect these changes appropriately. This is what we initially imagined as the typical reason for harness degradation. However, many projects in OSS-Fuzz are quite mature, and this effect can mainly be observed in projects with active development, which leads to a relatively low number of observed cases. This degradation usually surfaces as an increase in total lines of covered code and a code coverage drop or as a coverage drop only.

One example of a project that degrades over time is the Serenity project[47], see Figure 6d. Initially, some parts seem well fuzzed, but the harnesses are not updated for newly added code as the project ages, leading to harness degradation.

3.5.3 External Code in Coverage Report. (Cases: 45) Within this category, we classify projects that include code not part of the project proper in the coverage report, such as external libraries or third-party code. This is important for coverage measurement, which is a potentially error-prone process [45], as this additional code distorts the coverage measurement of the actual program under test.

One extreme example is a 255 line project¹⁰ (grpc-httpjson-transcoding [26]), shown in Figure 8. Towards the end of 2021, the Protobuf [44] and googletest[18] libraries with 42k lines of code¹¹ are added to this project, as can be seen in Figure 8c. This caused a drop in relative coverage by 93% points. This happens again in mid-2023¹², which adds another ~50k lines of code, as shown in Figure 8d. Overall, any nuance in the coverage over time is lost due to adding these external libraries, as is shown in Figure 8a.

These cases can be observed by a huge relative change in total lines covered and a large drop in coverage percentage. While the core project code can remain well-fuzzed, it will be difficult to detect actual harness degradation and other performance issues in the future, as the coverage report is (predominantly) influenced and distorted by the external code.

⁸before: <https://storage.googleapis.com/oss-fuzz-coverage/curl/reports/20221130/linux/src/report.html>

after: <https://storage.googleapis.com/oss-fuzz-coverage/curl/reports/20221201/linux/src/report.html>

⁹<https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=53965>

¹⁰ <https://storage.googleapis.com/oss-fuzz-coverage/grpc-httpjson-transcoding/reports/20210922/linux/proc/self/cwd/report.html>

¹¹ <https://storage.googleapis.com/oss-fuzz-coverage/grpc-httpjson-transcoding/reports/20210927/linux/proc/self/cwd/report.html>

¹² <https://storage.googleapis.com/oss-fuzz-coverage/grpc-httpjson-transcoding/reports/20230701/linux/proc/self/cwd/report.html>

3.5.4 Corpus Size Decrease and Low Corpus Size. (Size decreases: 59, low counts: 84)

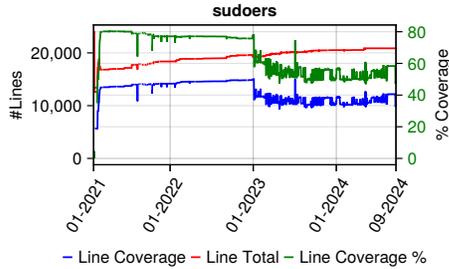


Figure 7. Sudoers coverage over time.

We noticed several cases where the corpus created by the fuzzer drops to a small fraction of its previous size. For example, one harness for Sudo [48] decreases from $\sim 12k$ corpus entries¹³ to 21¹⁴ at the start of 2023, as shown in Figure 7. In general, once this drop happens, a small corpus size and unstable coverage percentage will remain, as also manifests for the "sudoers" project.

We have investigated the code related to the coverage measurement and identified that for projects with this phenomenon the libFuzzer either detects a OOM, which is limited to 2GB, or is timing out, however, libFuzzer does not report these incidents with a non-zero returncode causing the coverage measurement to be reported as successful while only partially completing the coverage run. We have submitted a patch to OSS-Fuzz but as of the time of writing, have not received a response, as such we are not able to provide an accurate number of such cases. However, even though this is an artifact of OSS-Fuzz, in cases where the fuzzer produces a tiny corpus, we argue that fuzzing is still ineffective. To illustrate, imagine a project with a seed corpus that covers 80% of the project. However, the project uses a data integrity check, stopping the fuzzer from creating new accepted mutated inputs. This project will pass the required coverage measurements based on the seed corpus alone, but the fuzzing process is ineffective and no interesting inputs beyond the seed corpus will be found.

3.5.5 Others. We encounter some (6) intermittent clang coverage tool [37] errors causing nonsensical coverage reports. These explain some single-day variations in coverage. Additionally, some projects intended the removal of harnesses (3), which still cause a reduction in code coverage. Additionally, in some (14) cases we are missing the required familiarity with the project to be certain of how to classify them correctly.

¹³ https://storage.googleapis.com/oss-fuzz-coverage/sudoers/reports/20230104/linux/src/sudo/plugins/sudoers/regress/fuzz/fuzz_sudoers.c.html#L190

¹⁴ https://storage.googleapis.com/oss-fuzz-coverage/sudoers/reports/20230105/linux/src/sudo/plugins/sudoers/regress/fuzz/fuzz_sudoers.c.html#L190

Projects with Broken Harnesses. During our analysis, we encountered 28 projects that seem to have ineffective harnesses, some of which the maintainers might not be aware of. At the time of writing we are still in the process of informing the projects of these findings.

We identify four common causes for coverage drops. Harness build failures, added project code, added external code, and errors during coverage measurement.

3.6 RQ5 What are practical ways to detect harness degradation?

As we have now identified common causes of coverage drops, we can continue to explore how to detect harness degradation. We aim to identify practically useful metrics to monitoring over time. This is in line with the approach of Fuzz Introspector and OSS-Fuzz, which already provide certain metrics to identify problems in the fuzzing pipeline. Thus, to provide context, we first discuss already existing metrics: (1) The project and the fuzzing harness should build and run successfully. Failures are reported as build errors in the issue tracker for OSS-Fuzz. (2) Bugs resulting from found crashes are reported to the issue tracker. As crashes can severely impede fuzzer effectiveness, these can be seen as a type of notification. (3) Introspector reports a "blocked fuzzer" if the achieved code coverage is too low compared to what is possible based on static analysis. (4) Introspector also reports functions as blockers if they lead to unexplored code with large complexity, and the fuzzer cannot get through them. This is less of a metric and more a guidance for maintainers on what to focus on to improve fuzzing.

While all of these are important to monitor for maintainers, the number of slowly degrading OSS-Fuzz projects, as previously discussed and seen in figure 3, shows that the currently implemented monitoring mechanisms are insufficient. Therefore, to complement these signaling metrics, we propose a set of additions as discussed in the following. These novel metrics are based on our previous analysis results, especially the findings of RQ4 as described in section 3.5.

(1) Improve coverage measurement error reporting. Currently, timeouts and cases where the coverage measurement runs out of memory are not reported to maintainers. Based on our analysis, this is a fundamental issue, which affects approximately at least 10% of harnesses. Exact numbers can be provided, once this change is integrated. Our correspondence with maintainers revealed that this has been a surprising behavior for all maintainers we had correspondence with. (2) Statefulness of harnesses is currently not detected. This can be another cause of inconsistent coverage results. Note that this has been a planned feature of OSS-Fuzz before. At the time of writing, we have a working prototype but have not received a response regarding the integration into OSS-Fuzz. (3) A closely related metric is the corpus size in relation

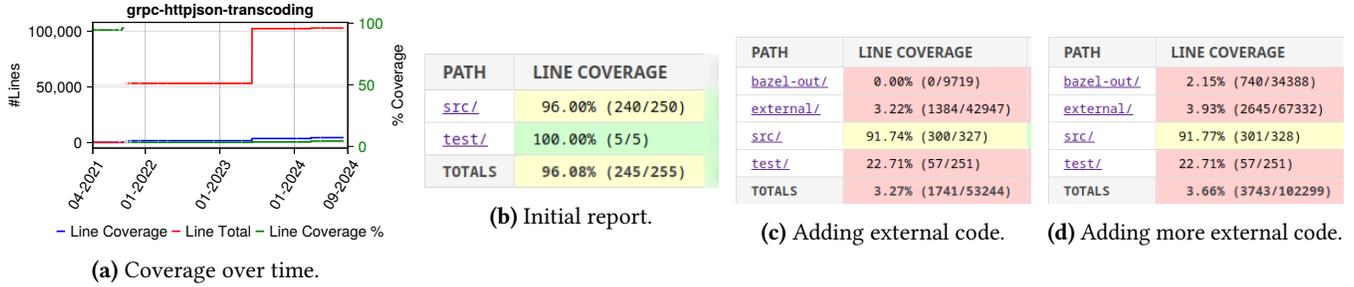


Figure 8. The coverage performance and report of `grpc-httpjson-transcoding`.

to covered code or covered complexity. While coverage is a useful metric, it does not allow assessing how well the covered code is fuzzed. For example, a single seed corpus entry may cover 80% of the code, but there could be a data integrity check stopping the fuzzer from generating any new related inputs. So, while the coverage percentage is at an acceptable level, based on the seed corpus alone, the project is effectively not being fuzzed. We propose a very simple and practical metric, the number of corpus entries compared to the complexity or lines covered. As we expect this metric to suffer from some level of noise, we suggest setting a rather conservative threshold.

None of the existing or so far proposed metrics consider the temporal aspect, as in how the fuzzing harness performance evolves over time. Thus, we add the following metrics to detect harness degradation issues which are easily visible over time:

(4) Keep track of project harnesses used in the past and inform the maintainers if one is no longer available. If removal was intended, this can act as a confirmation; if not, it should be reported. While OSS-Fuzz warns of project-wide build failures, it misses cases where individual harnesses stop building. Alternatively, a list of expected harnesses (or removed and renamed harnesses) could be kept up-to-date by the maintainers. (5) We noticed that most harnesses have very stable coverage results over time, even down to individual lines. Based on this insight, a metric that compares the coverage results over time is useful as a catch-all detection tool. Note that harnesses that do not have stable results either have: a broken coverage measurement or stateful harnesses, which should be addressed in any case; or have had project code changes, for which we want to detect possible degradation. This approach can also account for the case of harness degradation where new code is added or changed, but the harness is not updated. Note that this coverage drop should ideally be judged in the long term; small changes over time still add up. Therefore, a reasonable threshold needs to be chosen that avoids spurious alarms and is high enough to warrant action by the maintainer. Finally, to make this coverage-based metric meaningful, it is best to classify code as intended to be fuzzed by a harness or not. While it

is true that all code can have security implications, including third-party library code, it is usually better to fuzz such code separately. To ensure this, exclusion detection can be performed by static reachability analysis, which is already included in Fuzz Introspector.

At the time of writing, we are actively working with the Fuzz Introspector and OSS-Fuzz teams to integrate them. While we have received positive feedback so far, not all of our pull requests have been accepted at the time of writing. To provide full transparency, we include an exhaustive list of our pull requests in the aforementioned repository. Note, however, that there is no feasible way for us to effectively anonymize our official pull requests.

To counteract the measured degradation of fuzzing performance in OSS-Fuzz projects, we propose to monitor certain metrics, which include statefulness, rapid changes in corpus size, and coverage over time. We implement these metrics as part of OSS-Fuzz and Fuzz Introspector to make them available for public use to support the maintainers and contributors of OSS-Fuzz projects. The metrics will detect all cases of coverage drops discussed in RQ4, but also provide alerts for more specialized issues.

4 Threats to Validity

We reflect on possible issues and mitigations regarding generalizability, methodological mistakes, and the selection of experiments from which we draw our conclusions.

External validity. Refers to the degree to which our results can be generalized to other fuzzing harnesses outside of those included in our analysis. We analyzed all OSS-Fuzz projects (i) that are written in C/C++ (ii) for which we could access their git repository (iii) and where we could detect harnesses (iv). See Section 3.1 for more details. In the following, we discuss each point and how it influences generalizability.

(i) Projects accepted in OSS-Fuzz “must have a significant user base and/or be critical to the global IT infrastructure” [20]. Thus, we expect projects included in OSS-Fuzz to be open source, have an incentive for securing the project,

and usually be quite mature. Indeed, while not all harnesses are optimal, many projects spend significant effort on integrating fuzzing. Many projects in OSS-Fuzz are mature and have relatively little new functionality added, as we also observed in Section 3.3. Additionally, OSS-Fuzz offers a bounty program to incentivize integration and well-maintained harnesses [21]. Regarding the effects on generalizability, we expect these selection criteria to favor mature projects with an incentive to create effective fuzzer harnesses. However, we expect that time spent on security and fuzzing by open-source maintainers depends on their priorities [50], which should be motivated by the OSS-Fuzz reward program [21]. As a result, we expect a comparatively high effort to be spent on fuzzing that smaller projects might not be able to replicate. (ii) While the choice of C/C++ projects is to focus our research, we expect that generalizing to memory-safe languages will be difficult, especially regarding bug-finding capability. Also, we identified dependency management of C/C++ as an issue that negatively influences harness maintenance, which might differ from other languages. (iii) The choice to focus on Git repositories is reasonable, as this still allowed us to get data for most projects in OSS-Fuzz (491 / 510). However, it is plausible that projects not using Git might have a different engineering culture, which could influence generalizability. (iv) Our method of detecting harness changes (see section 3.1.4) likely introduces some bias. This is because projects that separate fuzzer harnesses from the main project presumably see the development of the fuzzing components as independent of the project’s code. This is an error-prone approach, as this can lead to code version mismatches and quicken harness degradation. However, during a manual investigation, we only found two projects that use separate repositories, which are excluded from our data as described in Section 3.1.4.

Internal validity. Refers to the degree to which our study minimizes potential methodological mistakes. While we cannot guarantee the absence of errors in our experiments, we follow a twofold approach to mitigate this issue. First, we combine our quantitative analysis with manual analyses. Second, we open-source our experimental infrastructure, scripts, and case study notes¹⁵.

Construct validity. Refers to the degree to which our study measures what we intend to measure. We mitigate this threat by using multiple metrics to measure and assess harness degradation and the associated effects. Note that we observed a coverage measurement error in the OSS-Fuzz infrastructure, as described in Section 3.5.4.

5 Related Work

Effectiveness of fuzzing. Ding and Le Goues [16] analyze the life cycle of bugs found in OSS-Fuzz and identify spikes

of rapid bug discovery (punctuated equilibria) amidst long periods of low bug-finding activity. At around the same time, Zhu and Böhme [58] also studied the lifecycle of bugs discovered by OSS-Fuzz and found that after an initial burst of bugs found as soon as projects are integrated into OSS-Fuzz, new bugs continue to be found *at a constant rate* throughout the lifetime of the project in OSS-Fuzz. At this point about 77% of new bugs discovered are regressions (bugs introduced with a recent commit). The rate of regression bugs increases after the initial burst of found bugs. Other studies [11, 34] demonstrate diminishing returns even within a fuzzing campaign as it continues for a long time. In contrast, we study whether harnesses continue to be maintained within OSS-Fuzz and how fuzzer effectiveness may degrade over time as a result.

Nourry et al. [41] analyze build failures in OSS-Fuzz and found that only 5% of builds for the median project fail, and that 80% of those failures are fixed within one day, which shows that fuzzing mostly remains functional over a project’s lifetime. As reasons for build failures, they identify environment, project dependency, and configuration issues, amongst others. Nourry et al. [42] studied Github Issues to understand which challenges developers face. They found, for instance, that developers find writing good harnesses hard, or that projects might suddenly fail to build for project-unspecific reasons, like changes to the fuzzing framework.

Automatic harnessing. Writing good fuzzing harnesses has been known to be a tedious and difficult part of fuzzer integration [36, 42] while further automation is seen as a central challenge in modern-day fuzzing [12, 52]. This resulted in various attempts to automate the process, including static analysis [14, 55, 56], guidance by runtime data [29, 31, 57], or a combination of both dynamic and static approaches [54]. Others suggest using artifacts such as unit tests [30] or client code for libraries [6, 28]. Finally, LLMs have been proposed [39] to generate fuzz harnesses. This includes efforts by Google’s OSS-Fuzz project, which recently adapted techniques to automatically generate fuzzing harnesses via LLMs [25]. The result of our study motivates the development of automatic harness *maintenance* techniques that should reactively handle the degradation or build failures of harnesses.

Test suite degradation. The degradation over time of unmaintained software components [2, 5, 8, 10, 40] or of unmaintained test suites [3, 7, 32] has been well-studied in software engineering. Test suites specifically do not only degrade like other software components; they also degrade due to individual test cases becoming obsolete. Fuzzing as a testing technique is, by design, less prone to this kind of degradation, as test cases can automatically be added or removed by a fuzzer if the program under test changes. Our findings further underline the robustness of fuzzing as a method of test case generation specifically regarding the aspect of harness degradation.

¹⁵Repository: <https://github.com/CISPA-SysSec/fuzz-harness-degradation>

6 Conclusion

In this paper we study fuzzing harness degradation by collecting a dataset based on OSS-Fuzz and analyzing coverage and bug-finding capability over time. Furthermore, we manually investigate coverage percentage drops and collect a set of common causes. Additionally, we contribute to OSS-Fuzz via implementations of metrics to detect harness degradation based on this research.

Our results confirm that fuzzing is effective and efficient in finding bugs [12, 22]. Our research shows that fuzzing harnesses have a surprisingly long effective lifetime where bugs are still found. However, they perform even better when actively maintained; this includes keeping harnesses updated when code changes, fixing harnesses that no longer build, and fixing bugs found by fuzzers, as fuzzers can get stuck on these bugs. To detect decreased fuzzer performance, we propose to monitor certain metrics, which include statefulness, rapid changes in corpus size, and coverage over time.

References

- [1] *Tree-sitter. A Parser Generator Tool. Documentation*, 2024. URL <https://tree-sitter.github.io/>.
- [2] Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. An empirical study of design degradation: How software projects get worse over time. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.
- [3] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology*, 73:66–80, 2016.
- [4] Samer Atawneh. The analysis of current state of agile software development. *Journal of Theoretical and Applied Information Technology*, 97(22):3197–3028, 2019.
- [5] Ahmed Baabad, Hazura Binti Zulzalil, Salmi Binti Baharom, et al. Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709, 2020.
- [6] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985, 2019.
- [7] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.
- [8] Alessandro Bianchi, Danilo Caivano, Filippo Lanubile, and Giuseppe Visaggio. Evaluating software degradation through entropy. In *Proceedings Seventh International Software Metrics Symposium*, pages 210–219. IEEE, 2001.
- [9] International Software Testing Qualifications Board. *Test Harness Definition*. International Software Testing Qualifications Board, 2024. URL https://glossary.istqb.org/en_US/term/test-harness.
- [10] Andrea Bobbio, Matteo Sereno, and Cosimo Anglano. Fine grained software degradation models for optimal rejuvenation policies. *Performance Evaluation*, 46(1):45–62, 2001.
- [11] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 713–724, 2020.
- [12] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2020.
- [13] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1621–1633, 2022.
- [14] Sheung Chi Chan, Adam Korczynski, and David Korczynski. *Oss-fuzzgen: Automated fuzzing of open source java projects*. 2023.
- [15] curl Project. *curl: command line tool and library for transferring data with URL syntax*. <https://github.com/curl/curl>. Accessed: 2024-10-19.
- [16] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142, 2021. doi: 10.1109/MSR52588.2021.00026.
- [17] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin IP Rubinstein. Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, pages 47–55, 2023.
- [18] Google. *googletest: GoogleTest - Google Testing and Mocking Framework*. <https://github.com/google/googletest>, . Accessed: 2024-10-27.
- [19] Google. *OSS-Fuzz: Setting up a new project*. Google, . URL <https://google.github.io/oss-fuzz/>.
- [20] Google. *OSS-Fuzz: Accepting New Projects*. Google, . URL <https://google.github.io/oss-fuzz/getting-started/accepting-new-projects/>.
- [21] Google. *OSS-Fuzz: Reward Program*. Google, . URL <https://bughunters.google.com/about/rules/open-source/5097259337383936/oss-fuzz-reward-program-rules>.
- [22] Google. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Google, 2017. URL <https://github.com/google/oss-fuzz>.
- [23] Google. *Fuzz Introspector Repository*. Google, 2024. URL <https://github.com/ossf/fuzz-introspector>.
- [24] Google. *Fuzz Introspector: Fuzzing Introspection of OSS-Fuzz projects*. Google, 2024. URL <https://introspector.oss-fuzz.com/>.
- [25] Google. *OSS-Fuzz-Gen: AI-Powered Fuzzing: Breaking the Bug Hunting Barrier*. Google, 2024. URL <https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html>.
- [26] grpc-httpjson-transcoding Project. *grpc-httpjson-transcoding*. <https://github.com/grpc-ecosystem/grpc-httpjson-transcoding>. Accessed: 2024-10-19.
- [27] Rashina Hoda, Norsaremah Salleh, and John Grundy. The rise and evolution of agile software development. *IEEE software*, 35(5):58–63, 2018.
- [28] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.
- [29] Seungho Jeon and Jung Taek Seo. Sp-fuzz: Fuzzing soft plc with semi-automated harness synthesis. In *International Conference on Information Security Applications*, pages 282–293. Springer, 2023.
- [30] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692. IEEE, 2023.
- [31] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.
- [32] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *2009 International Conference on Software Testing Verification and Validation*, pages 201–209. IEEE, 2009.
- [33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

- [34] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben ten Hove, and Marcel Böhme. Effectiveness and scalability of fuzzing techniques in ci/cd pipelines. *arXiv preprint arXiv:2205.14964*, 2022.
- [35] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.
- [36] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566. IEEE, 2018.
- [37] LLVM. *Source-based Code Coverage. Clang Documentation*. LLVM, 2024. URL <https://clang.llvm.org/docs/SourceBasedCodeCoverage.html>.
- [38] LLVM. *libFuzzer – A library for coverage-guided fuzz testing*. LLVM, 2024. URL <https://llvm.org/docs/LibFuzzer.html>.
- [39] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. *arXiv preprint arXiv:2312.17677*, 2023.
- [40] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt Von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16th european conference on software maintenance and reengineering*, pages 277–286. IEEE, 2012.
- [41] Olivier Nourry, Yutaro Kashiwa, Weiyei Shang, Honglin Shu, and Yasutaka Kamei. My fuzzers won’t build: An empirical study of fuzzing build failures. *ACM Transactions on Software Engineering and Methodology*.
- [42] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. The human side of fuzzing: Challenges faced by developers during fuzzing activities. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–26, 2023.
- [43] OpenSSL Project. *Openssl: Tls/ssl and crypto library*. <https://github.com/openssl/openssl>. Accessed: 2024-10-19.
- [44] protobuf Project. *protobuf*. <https://github.com/protocolbuffers/protobuf>. Accessed: 2024-10-19.
- [45] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024.
- [46] Kostya Serebryany. *OSS-Fuzz-Google’s Continuous Fuzzing Service for Open Source Software*, 2017.
- [47] serenity Project. *serenity: The Serenity Operating System*. <https://github.com/SerenityOS/serenity>. Accessed: 2024-10-27.
- [48] sudo Project. *sudo: Utility to execute a command as another user*. <https://github.com/sudo-project/sudo>. Accessed: 2024-10-19.
- [49] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [50] Shao-Fang Wen, Mazaher Kianpour, and Stewart Kowalski. An empirical study of security culture in open source software communities. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 863–870, 2019.
- [51] Dave West, Tom Grant, Mary Gerush, and David D’Silva. Agile development: Mainstream adoption has changed agility. *Forrester Research*, 2(1):41, 2010.
- [52] Qian Yan, Minhuan Huang, and Huayang Cao. A survey of human-machine collaboration in fuzzing. In *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, pages 375–382. IEEE, 2022.
- [53] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISA Helmholtz Center for Information Security, 2024. URL <https://www.fuzzingbook.org/>. Retrieved 2024-07-01 16:50:18+02:00.
- [54] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. {APICraft}: Fuzz driver generation for closed-source {SDK} libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828, 2021.
- [55] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. Automata-guided control-flow-sensitive fuzz driver generation. In *USENIX Security Symposium*, pages 2867–2884, 2023.
- [56] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327. IEEE, 2021.
- [57] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. Daisy: Effective fuzz driver synthesis with object usage sequence analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 87–98. IEEE, 2023.
- [58] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

A Supplementary Materials

A.1 Coverage Since Harness Update

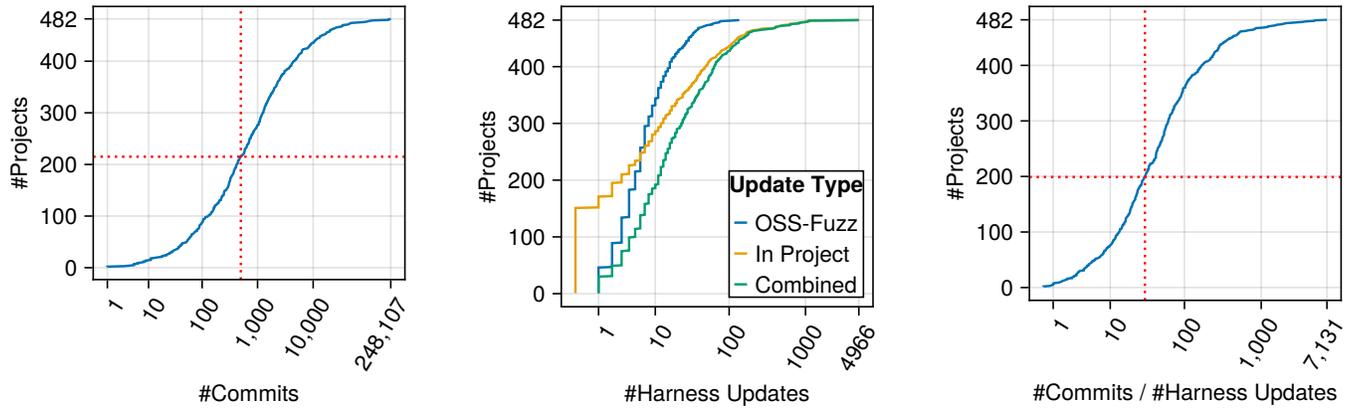


Figure 9. Cumulative number of projects, respectively, by number of commits, harness updates, and ratio of commits to harness updates. Dotted lines in red mark the cutoff points.

As the projects are quite diverse in their activity level and number of harness updates. We separate the projects twice, once based on the number of commits, to approximate activity (with the cutoff point at 500 commits), and again by those with a high ratio of harness updates to commits, to approximate harness maintenance (with the cutoff point at 30 commits per harness update on average). This results in four quadrants, those with high activity and high harness maintenance, high activity and low maintenance, and so on. The following plots show combinations of these quadrants, such as those with high and low maintenance or high and low maintenance for a high activity. Comparing to the overall result, the trends seem a bit clearer. For example, Figure 10 (high maintenance ratio projects) seems to trend better than Figure 11 (low ratio). However, overall we are not quite convinced that variance is small enough to truly interpret these results.

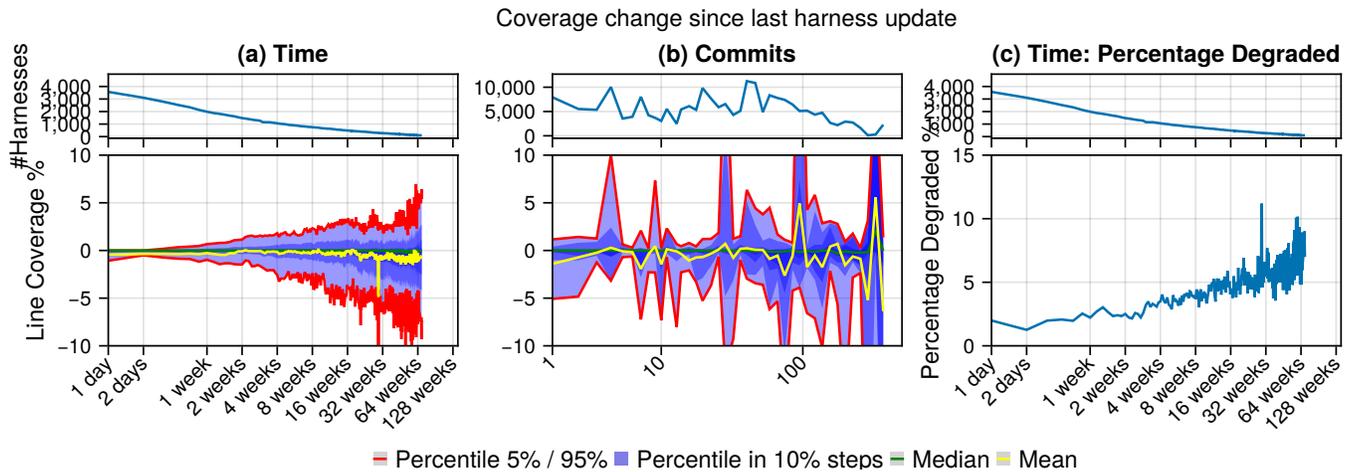


Figure 10. Coverage since the last harness update for projects with a high ratio of harness updates.

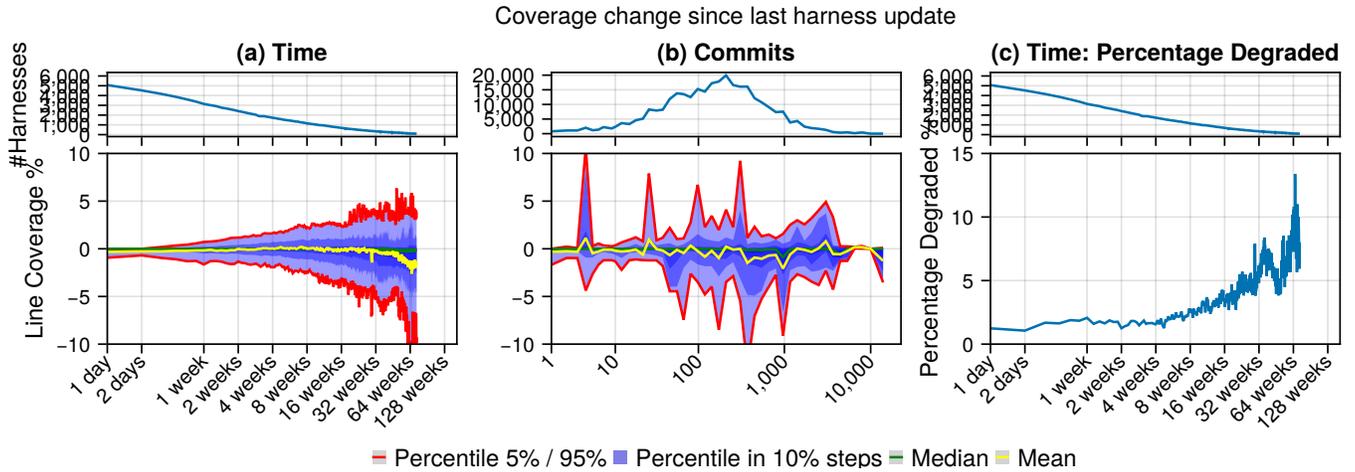


Figure 11. Coverage since the last harness update for projects with a low ratio of harness updates.

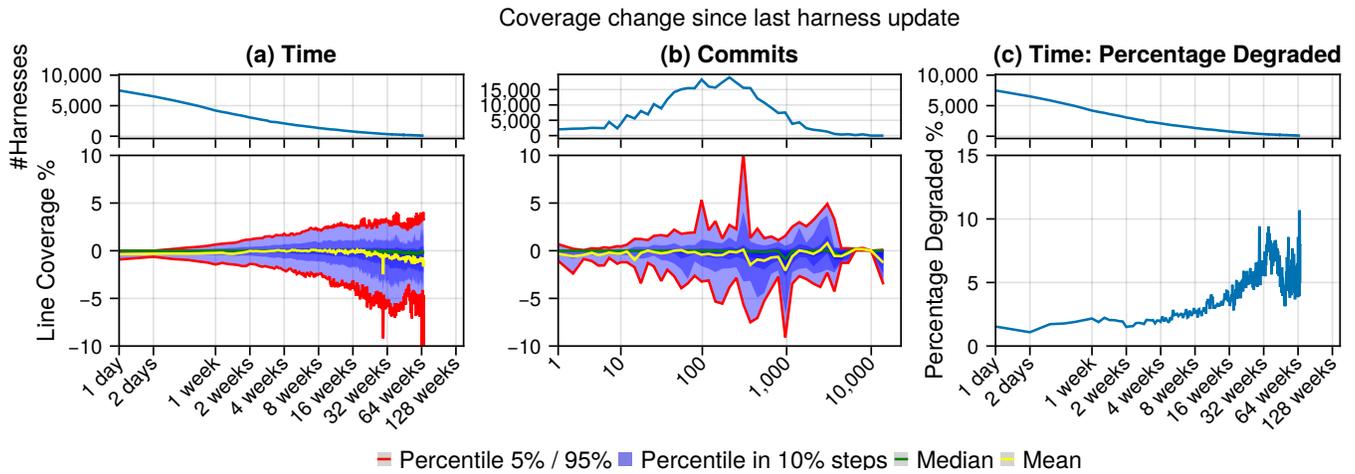


Figure 12. Coverage since the last harness update for projects with a high number of commits.

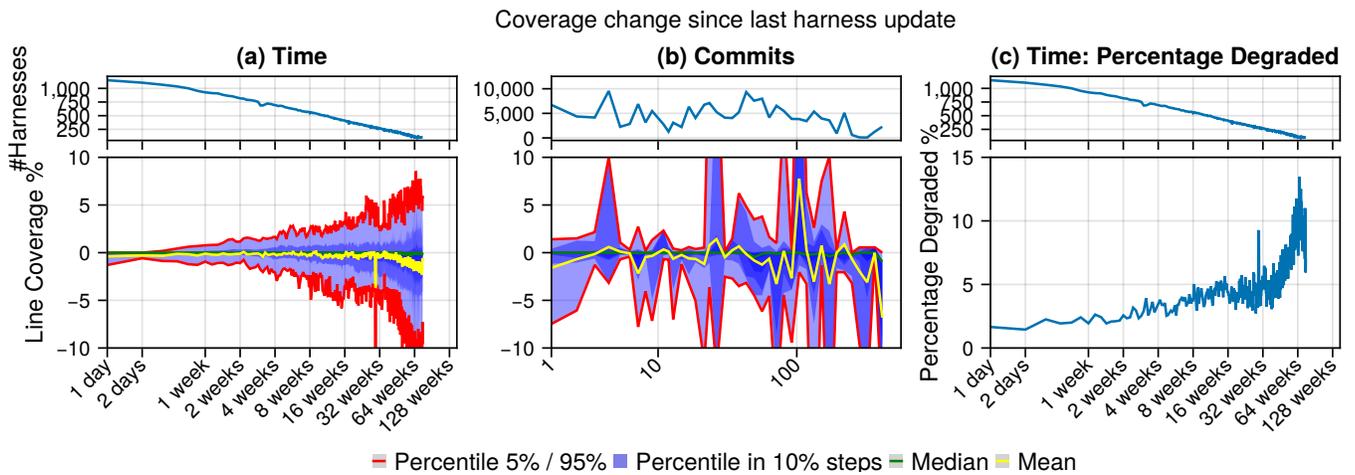


Figure 13. Coverage since the last harness update for projects with a low number of commits.

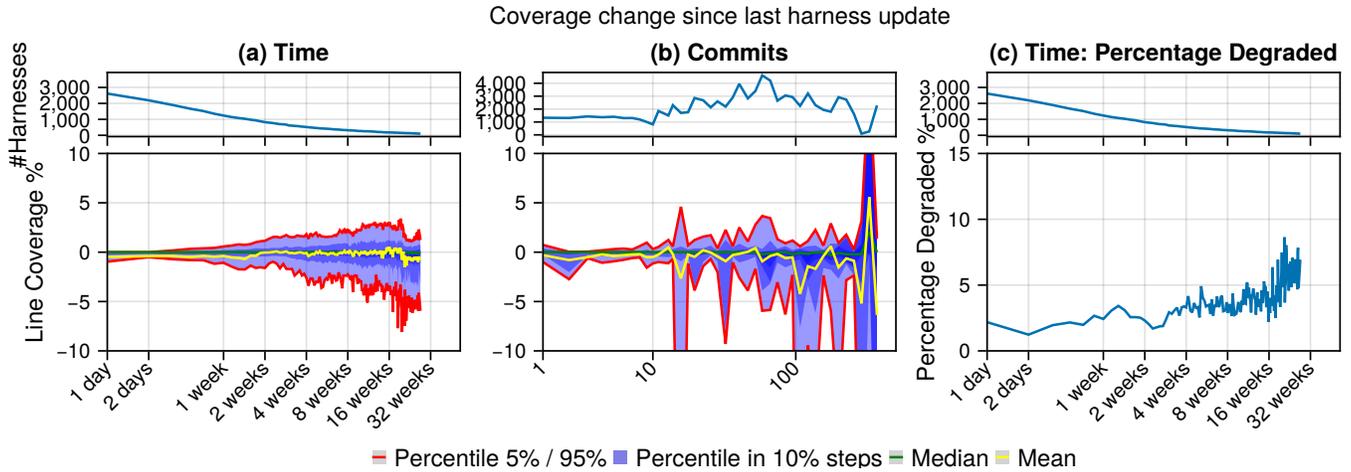


Figure 14. Coverage since the last harness update for projects with a high number of commits and a high ratio of harness updates.

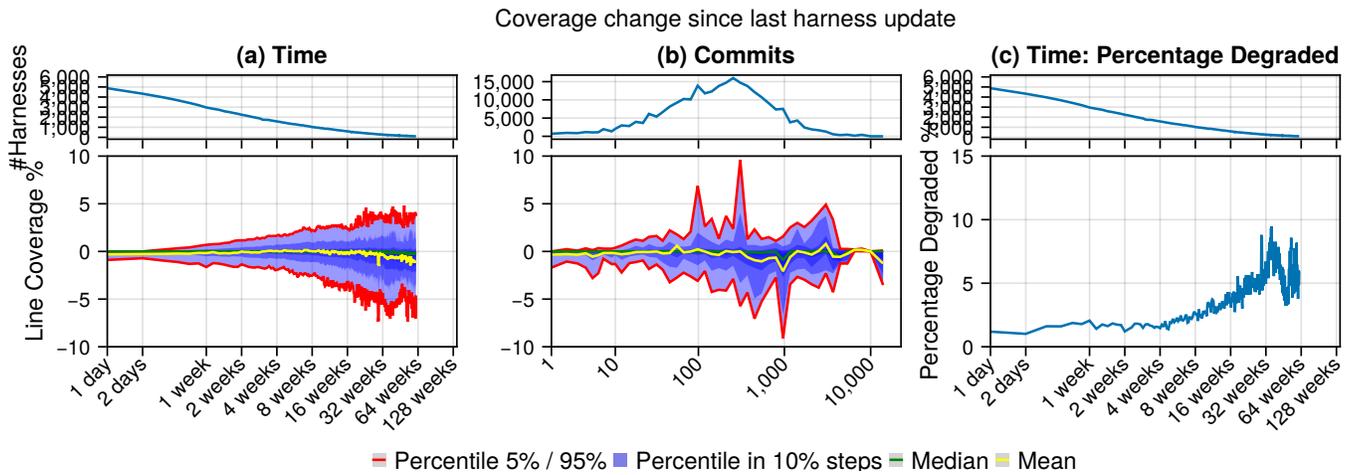


Figure 15. Coverage since the last harness update for projects with a high number of commits and a low ratio of harness updates.