Continuous Fuzzing

Klooster, Thijs; Turkmen, Fatih; Broenink, Gerben; Hove, Ruben Ten; Bohme, Marcel

Published in:
Proceedings - 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT 2023

DOI:
10.1109/SBFT59156.2023.00015

Publication date:
2023

Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines

Thijs Klooster
TNO, The Netherlands
ORCID 0000-0001-7151-1332

Fatih Turkmen
University of Groningen, The Netherlands
ORCID 0000-0002-6262-4869

Gerben Broenink
TNO, The Netherlands
ORCID 0000-0002-3660-8179

Ruben ten Hove
TNO, The Netherlands
ORCID 0000-0002-9137-9145

Marcel Böhme
MPI-SP, Germany
ORCID 0000-0002-4470-1824

2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT) | 979-8-3503-0182-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBFT59156.2023.00015

***Abstract*—While fuzzing can be very costly, it has proven to be a fundamental technique in uncovering bugs (often security related) in many applications. A recent study on bug reports from OSS-Fuzz observed that recent code changes are responsible for 77% of all reported bugs, stressing the importance of continuous testing. With the increased adoption of CI/CD practices in software development, it is only natural to look for effective ways of incorporating fuzzing into continuous security testing. In this paper, we study the effectiveness of fuzz testing in CI/CD pipelines with a focus on security related bugs and seek optimization opportunities to triage commits that do not require fuzzing. Through experimental analysis, we found that the fuzzing effort can be reduced by 63% in three of the nine libraries we analyzed (55% on average). Additionally, we investigate the correlation between fuzzing campaign duration and the effectiveness of fuzzers in vulnerability discovery: a significantly shorter fuzzing campaign facilitates a faster pipeline for developers, while it can still uncover important bugs. Our findings suggest that continuous fuzzing is indeed beneficial for secure software development processes, and that there are many opportunities to improve its effectiveness.**

I. Introduction

In the past decade, fuzzing has become one of the most successful automated vulnerability discovery techniques [24]. A fuzzer automatically generates inputs for a given Program Under Test (PUT) until the program crashes or a sanitizer terminates the PUT. A sanitizer is an automatic test oracle that crashes the PUT for inputs that expose security flaws, such as buffer overflows. Google uses 30k cores to continuously fuzz all Google products in ClusterFuzz [16]. Microsoft recently introduced fuzzing as a service in Project OneFuzz [27]. More than 600 open source projects are continuously fuzzed by Google's OSSFuzz [13]. In the six years since its launch, OSS-Fuzz has been continuously running and found more than 40k bugs, a lot of which are security bugs. Once a project has been sufficiently fuzzed, most new bugs that can be found are the result of recent code changes [33].

In this paper, we explore the challenges and opportunities of incorporating fuzzing into the continuous integration / continuous deployment (CI/CD) pipelines. In such pipelines, each pull request triggers an automatic build of the new code which often runs automated regression tests. A *pull request* (PR) is submitted by an author to integrate new code into the main branch of the code repository of the PUT. If the automated build fails, the PR authors and reviewers are informed accordingly. The integration of fuzzing into CI/CD allows for the detection of bugs right after they are introduced, and before they reach the main branch [4], [9], [11].

Traditionally, fuzzing is considered resource and time intensive. In academia, the recommended fuzzing campaign duration is at least *24 hours* [21] for effective fuzzing. In the context of CI/CD, resources and time are limited. *10 minutes* is considered a reasonable build time (which also includes testing) [6], [19], [31]. Furthermore, Github Actions currently charges 0.48 USD per hour [7], aborts automated builds after six hours [8], and only limited resources are available.

Hence, we explore opportunities to save resources by skipping fuzzing when we know for sure that no (new) bugs can be found as per the previous fuzzing sessions. To provide empirical evidence, we collected the most recent commits (ranging from 241 to 7847) of nine open-source libraries. We analyzed them with a checksum-based method to help determine if fuzzing can be avoided after some of the commits. We also analyzed the trade-off between fuzzing campaign duration and the number of bugs found for reasonable values of campaign duration. We used `Magma` [18], a state-of-the-art fuzzing benchmark with security-critical bugs and vulnerabilities. We hypothesize that shorter but continuous fuzzing campaigns (suitable for CI/CD pipelines) should still positively impact vulnerability discovery. This is based on the fact that fuzzing a non-fuzzed target can uncover a significant amount of vulnerabilities with relatively few resources [1]. Our work provides empirical evidence that:

- Considerable fuzzing effort can be avoided, thereby saving computational resources, when source code changes bundled in commits are carefully scrutinized.
- Continuous but shorter fuzzing campaigns can still uncover important bugs.

One of the most practical implications of our work is that the CI/CD bug discovery process can significantly benefit from fuzzing, after certain optimizations and trade-offs are applied.

979-8-3503-0182-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBFT59156.2023.00015

II. Background and Motivation

In this section, we summarize fuzzing as an automated testing technique and motivate the need for its incorporation into CI/CD pipelines.

A. Fuzzing in CI/CD

Fuzzing is a dynamic testing technique that, starting from one or more seeds, executes a PUT with repeatedly mutated inputs until some "interesting" behavior is observed [21]. A fuzzer records the observed program behaviour and saves the inputs that lead to the observed behaviour for future use. The fuzzing process generally stops when the user intervenes, a predetermined timeout is reached, or a bug is found.

Fuzzers are commonly used in vulnerability discovery. They are classified according to their knowledge of the PUT in terms of input format and program structure, and the strategies they employ in generating inputs (e.g., mutational fuzzers). A *black-box fuzzer* assumes no knowledge of the PUT, a *grey-box fuzzer* assumes partial knowledge and often leverages code instrumentation to glean that knowledge, and a *white-box fuzzer* employs sophisticated analysis techniques (e.g., symbolic execution). In order for a fuzzer to test the robustness of some functionality within a PUT, a *fuzzing harness* needs to be written, which acts as the main entry point for the fuzzer to reach the specific functionality. A single library can have many fuzzing harnesses, each providing access to different kinds of functionality of the software. In what follows, we use the term *fuzz target* to indicate a compiled fuzzing harness (i.e., the executable file resulting from the compilation of the harness).

A *CI/CD pipeline* refers to the largely automated process of integrating committed changes to the code, testing and then moving the code from the commit stage to the production stage [28]. Frequent but small (merge) commits are encouraged [6], [10]. Indeed, a large scale analysis conducted by Zhao et al. [32] determined that around 21 (median) commits are made per day for 575 open source projects. Performing automated testing on each code change enables earlier feedback [6] on bugs since failed stages and errors can be traced to sources (and the developers) more quickly, thanks to the limited scope [20].

However, running long fuzzing campaigns after each commit does not scale. Since only some commits include code changes that require fuzzing, methods to identify such commits are needed to incorporate fuzzers into CI/CD pipelines.

Moreover, determining a good timeout for fuzzing (i.e., campaign duration), is crucial for effective testing. While there are different proposals, a commonly accepted fuzzing campaign duration is 24 hours [21]. However, in a CI/CD setting, we are limited in terms of the available resources [26] and running an automated build for 24 hours on each commit is impractical. Indeed, in a CI/CD setting the testing time that is considered "reasonable" is much shorter. For instance, Fowler [6] proposes 10 minutes as a guideline. Thus, striking a balance between fuzzer effectiveness and testing duration in the CI/CD pipelines is of practical concern.

The idea of continuous fuzzing (in the CI/CD sense) is not new. For instance, Google offers `CIFuzz` [14] as part of `OSS-Fuzz` that runs fuzzers for 10 minutes (max 6 hours) over 30 day old/public regressions of (selected) open source projects. Similarly, GitLab has coverage-guided fuzzing (and Web API fuzzing) integrated to its CI/CD offerings that can be run either 10 minutes or 60 minutes [9] in its own pipeline stage. However, there is no empirical study investigating this trade-off between bug finding and campaign length that can inform this decision.

B. Research Questions

In our work, we are interested in the dimensions that will have the most impact on the performance of the fuzzing effort in CI/CD pipelines. More specifically, we want to answer the following questions:

RQ.1 ***How can fuzz testing be adapted to fit CI/CD pipelines given their clashing ambitions in computational resource usage?***

RQ.2 ***What is a reasonable fuzzing campaign duration that is compatible with CI/CD testing timelines but is still effective in finding security vulnerabilities?***

III. Benchmarks

In order to investigate the effectiveness of fuzzers while varying the fuzzing campaign duration in the CI/CD setting, we need a fuzzing platform that enables the evaluation of multiple fuzzers and supports good quality benchmark suites (i.e., target programs). While there exist several such platforms, we chose to employ `Magma` [18] due to the following reasons:

- `Magma` is a state-of-the-art fuzzing benchmark with an emphasis on providing the *ground-truth* for bugs and their precise location in the PUT. This is achieved by reverting the fixes for known, real-world bugs which have existed in their respective software repositories in the past.
- The benchmark includes 138 bugs in total, spanning 11 distinct vulnerability types (CWE-IDs) over nine distinct open source software projects that are widely used.
- The benchmark provides instrumented targets which make a distinction whether a bug was *reached*, *triggered*, or *detected* to enable more accurate evaluation of fuzzer effectiveness. A bug is *reached* when the faulty line of code is executed, *triggered* when the fault condition is satisfied, and *detected* when the fuzzer observes the faulty behaviour.
- `Magma` supports many different fuzzers out-of-the-box, including `AFL++`, `libFuzzer`, and `Honggfuzz`, which are used in Google's `OSS-Fuzz` platform as well [13].

We considered all (nine) available libraries/PUTs in `Magma`, which are listed in Table I along with various CI/CD aspects of their online repositories. For each library, the number of fuzzing targets that can be used for benchmarking varies. Since the `Magma` benchmark is created by reintroducing known bugs that existed in the library in the past, the number of bugs per library varies as well. These software libraries are still receiving commits to this day, which means they are

TABLE I: CI/CD details and Magma parameters of the nine libraries used in the experiments.

Library	Commits	Branches	Repo size (MB)	Language	Magma fuzz targets	Magma bugs
libsndfile	3083	23	36	C	1	18
libtiff	3953	4	17	C	2	14
libpng	4098	8	48	C	1	7
libxml2	5353	12	52	C	2	17
lua	5420	5	13	C	1	4
poppler	7206	20	24	C++	3	22
sqlite3	26610	1128	146	C	1	20
openssl	31163	21	662	C	5	20
php	128239	407	615	C	4	16

still actively developed, and fuzzing them has real-world and practical benefits.

While there are several metrics that are being used for fuzzer evaluation in practice, *the number of crashes* is arguably the most used. However, Klees et al. [21] suggest that the number of crashes metric can dramatically overestimate the number of underlying bugs. Since Magma uses ground-truth based metrics (the *number of bugs reached, triggered,* and *detected*), this issue is mitigated. However, Magma is intended to be used for standard fuzzer evaluation, not specifically in the context of continuous fuzzing. Therefore, we have extended the benchmark with certain modifications which we summarize next.

A. Setup

The Magma platform orchestrates the individual fuzzing campaigns, monitors the findings, and outputs the results. The platform also allows for the specification of parameters of the fuzzing campaigns, e.g., which fuzzers to use, which fuzz targets to use, and the amount of repeated trials. Since the platform is based on Docker, it was fairly easy to adapt it to the needs of our experiments. In our setup, we employ ensemble fuzzing (which is not supported out-of-the-box in Magma) in a continuous manner, through shorter campaigns. In ensemble fuzzing, different fuzzers collaborate on the same fuzz target whereby individual strengths of the fuzzers are exploited. Continuous ensemble fuzzing can be very beneficial when integrated into CI/CD pipelines, seeing as OSS-Fuzz has managed to find 40k bugs in five years [15] while employing such an ensemble fuzzing approach. Even though the utilization of multiple fuzzers increases the resource usage, the empirical evidence provided by OSS-Fuzz suggests that it is an effective way of fuzzing in continuous practices. Therefore, this will be our starting point from which we aim to reduce resource usage based on fuzz targets that have not been affected by code changes. Whenever there are targets that do not have to be fuzzed, this actually results in three fewer CPU cores needed per target (when using an ensemble setup of the three fuzzers that OSS-Fuzz uses).

In standard fuzzing sessions that take up to 24 hours or even longer, ensemble fuzzing requires fuzzers to share the same working directory, such that their progress can be shared throughout the whole of the campaign. With continuous fuzzing, the average per-commit campaign duration would be much shorter. Therefore, it suffices to synchronize the fuzzers *after* every commit. The synchronization process combines the corpora and minimizes the result to obtain the new seed corpus for the next commit that will get pushed. Figure 1 shows the overall design of ensemble fuzzing in the context of continuous integration. Upon receiving a new commit, the harnesses will be compiled into targets, which in turn will be used to minimize the corpus. The minimized corpus is used by every fuzzer in the setup, where every fuzzer contributes towards an updated corpus by submitting newly found interesting inputs. This updated corpus will be re-used in future commits, effectively allowing fuzzers to collaborate on each fuzz target. Corpus sharing is implemented this way in ClusterFuzzLite and in GitLab's coverage-guided fuzzing [9], [11]. In summary, a continuous fuzzing setup where different fuzzers can collaborate on the same targets effectively can be designed by applying the ensemble fuzzing strategy within continuous integration, sharing and minimizing the corpus between each commit.

IV. Experiments

We conducted an extensive set of experiments in order to seek answers to the questions **RQ.1** and **RQ.2**. In this section, we elaborate upon the experiments and present the results. The source code of the extended Magma benchmark as well as the source code used for the experiments can be found online[1].

A. Fuzz Target Selection

In an effort to optimize the resource usage in continuous fuzzing, the first experiment aims to find out what proportion of the fuzzing campaigns should be started by default. In other words, we are interested in understanding if there are any commits that should not trigger a fuzzing campaign due to the code changes that do not affect certain fuzz targets.

One option of determining whether a fuzz target is different from that of the preceding commit, is by calculating the checksum of both targets. Calculating a checksum can be done over the source code of the target although it is still not trivial to trace which lines of code affect which target given the checksums. Instead, calculating the checksum of the fuzz target itself (being an executable file) would present even a simpler solution and reveal whether the target is different between the respective commits. Before the checksum can be calculated, the fuzzing harnesses have to be compiled into targets. This does require some time and resources, in contrast to when we would be able to do it without compiling the harnesses. Fortunately, the compilation of source code

[1]https://github.com/kloostert/CICDFuzzBench

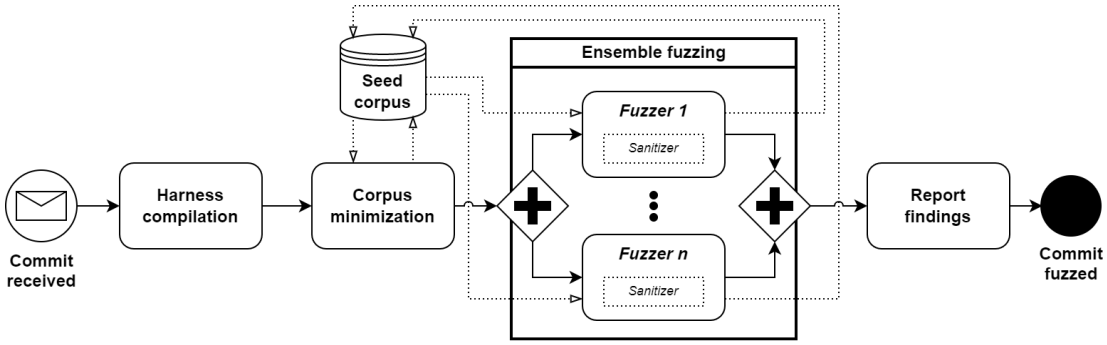


Fig. 1: General design of ensemble fuzzing in the context of continuous integration.

in a CI/CD pipeline is oftentimes already taken care of in the building stage, and so this compilation will not result in any additional overhead during the testing/fuzzing stage. After compilation, checksums will be calculated for all of the targets, and they will be compared to the checksums from the preceding commit. For targets that have identical checksums, we can decide to not start their corresponding fuzzing sessions, saving computational resources.

Another option is to check whether there is a change in code coverage. For instance, Google's `CIFuzz` [14] also provides the option to employ target selection. Yet, this option is available only to projects that support `OSS-Fuzz`'s code coverage, which makes it more complex to set up properly. Using checksums directly on fuzz targets enables software projects for which `OSS-Fuzz`'s code coverage is not available to take advantage of the fuzz target selection.

Table II lists the number of fuzzing harnesses that were processed during the target selection experiment, the number of commits that were processed for each library, and the proportion of the processed commits over the total amount of commits in the respective repositories. Instead of using `Magma`'s version of the libraries that contain the bugs, this experiment focuses on the respective live repositories, and so as many of the most recent commits as possible are processed. The experiment can no longer process commits when either the repository changed in a way that automated fuzz target compilation was no longer possible, or the fuzzing harnesses themselves were no longer part of the repository (the commit that introduced the fuzzing harnesses was reached). This is why both the number of commits processed and the percentages of commits processed differ among the nine libraries.

When the experiment starts, the most recent version of the software is taken, and the processing will continue backwards in time from there. For every commit, checksums for all of the available fuzzing harnesses will be obtained by first compiling the harnesses into targets, and consecutively calculating their checksum. The same will be done for the preceding commit, after which the checksums will be compared for each target, resulting in a proportion of unchanged fuzz targets for a single commit. These steps will be repeated until the experiment can no longer process commits due to the reasons mentioned before. When this process terminates, we would have obtained the overall proportion of fuzz targets that remained unaffected by commits on average.

TABLE II: Target selection experiment details: the number of harnesses and commits which were processed (along with the proportion of all commits in the repository), and the proportion of identical fuzz targets that was found.

Library	Harnesses processed	Commits processed	Identical targets
`libsndfile`	1	241 (8%)	64%
`libtiff`	2	801 (21%)	53%
`libpng`	1	1158 (28%)	41%
`lua`	1	2285 (42%)	20%
`poppler`	2	1919 (27%)	44%
`openssl`	12	7847 (26%)	63%
`php`	9	7821 (6%)	64%
			Weighted mean: 55%
`libxml2*`	1	625 (12%)	00%
`sqlite3*`	1	483 (2%)	00%

Results Our experiment achieved varying proportions of processed commits (Table II) among the nine libraries, ranging between 6% and 42% of all commits. During the experiment, we noticed that two of the nine libraries (`libxml2` and `sqlite3`) never produced identical fuzz targets, since they interfered with our checksum approach. Further investigation showed that this was due to the inclusion of versioning information (compilation timestamp or `Git` revision) in the actual executable fuzz targets. In order to alleviate this problem, either a more sophisticated form of target selection has to be applied, or the build process of these fuzz targets should be adapted in such a way that no versioning information makes it into the compiled targets, which proved to be laborsome. Since the proportion of identical fuzz targets could not be measured properly for these two libraries, they were not taken into consideration (i.e., marked with * in Table II) towards the final results of this experiment.

For the remaining seven libraries, the proportion of identical fuzz targets ranges from 20% to 64%, where the weighted arithmetic mean lies at 55% (as shown in Table II). Another observation is that the scope of the fuzzing harnesses greatly impacts the proportion of identical targets. Harnesses that are aimed specifically at one single function of the target library do not receive many changes, while harnesses that cover a library

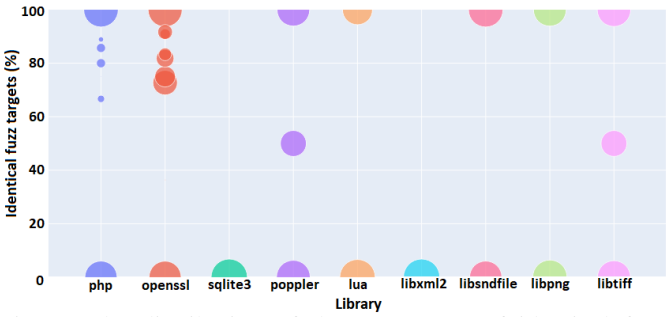


Fig. 2: The distribution of the percentage of identical fuzz targets per library for all processed commits. The marker size represents the logarithmically scaled distribution per library. `libxml2` and `sqlite3` have no identical fuzz targets.

more broadly (like `lua`) receive relatively more changes.

These results suggest that even employing a *simple* technique for selecting the subset of fuzz targets which should be fuzzed, can result in saving *more than half* of the computational resources needed for the continuous fuzzing effort on average.

Figure 2 shows more details across fuzz targets and commits per project. Exploring the results more in-depth, the experiment reveals that the distribution of identical fuzz targets per commit varies among the seven libraries greatly. We find that for most of the commits, either none of the fuzz targets were affected (the complete fuzzing stage of the pipeline can be skipped), or all of the targets were affected (all fuzz targets need to be fuzzed simultaneously). Three of the libraries (`libsndfile`, `libpng`, and `lua`) do not show any commits with a proportion of identical commits somewhere between 0% and 100%, since only a single fuzz target was processed for these libraries. Two other libraries (`libtiff` and `poppler`) do have commits where only one of the two fuzz targets was affected. The libraries with the largest number of fuzz targets (`openssl` and `php`) show that most commits fall into the categories of either no targets were affected or all of them were. However, we also observe a significant number of commits where the code changes affected only a small portion of the fuzzing harnesses. Target selection in a CI/CD pipeline would be beneficial in those cases as well, since the number of needed CPU cores scales with the number of affected fuzz targets.

In order to address **RQ.1**, we found that it suffices to *only fuzz the targets for which we know the code has changed* by the latest commit. Compared to fuzzing all of the targets every time, this would *save up to 64%* and *55% on average* in terms of fuzzing campaigns, and therefore in the computational resources required for fuzzing.

B. Fuzzing Campaign Duration

In an effort to settle the dispute between CI/CD pipeline durations and their effectiveness, the second experiment aims to find the optimal campaign duration that strikes a balance between faster builds and effective fuzzing. For this experiment, we chose to include three of the most popular fuzzers that are being used in modern fuzzing platforms like `OSS-Fuzz` [30]: `AFL++` [5], `libFuzzer` [25], and `Honggfuzz` [12]. In order to deal with the stochastic nature of fuzzers, we repeat every fuzzing campaign 10 times to reduce the effect of outliers due to randomness. We also use each of these repetitions to simulate a single commit, resulting in 10 simulated commits per fuzz duration. For every such duration, the successive runs will build a corpus along the way, as would be the case with real commits. Even though there are no actual code changes in these simulated commits, this does not affect the results and the conclusions we can draw from them. This is because the effectiveness of the fuzzing campaigns is not linked to the actual code changes when using state-of-the-art *coverage-guided* fuzzers. However, when we would have used directed fuzzers (e.g. `AFLGo` [2] and `AFLChurn` [33]) instead, our technique of simulating commits would have interfered with the results. This is due to the nature of these *distance-guided* fuzzers; they are specifically directed towards e.g. the code changes of commits.

With this experiment, we want to investigate the effectiveness of (continuous) fuzzing when the campaign duration is significantly reduced (with respect to the recommended *24 hours* [21]). Therefore, we do not want to exceed the available time a fuzzing campaign would have when running nightly (from midnight until eight in the morning). Although *10 minutes* is a commonly accepted CI/CD build time (which also includes testing) [6], [19], [31], we still wanted to investigate the effectiveness of an even shorter fuzzing session. In the end, we arrived at a set of campaign durations ranging from 5 minutes to 8 hours. The experiment was conducted on a Linux virtual machine with eight vCPUs running at 2.1 Ghz, eight gigabytes of RAM, and 80 gigabytes of disk space. During this experiment, we conducted *over one CPU-year worth* of continuous fuzzing campaigns.

The experimental setup uses corpus sharing and minimization between successive repetitions of the fuzzing campaigns (as per Figure 1), in order to simulate 10 commits being processed in a continuous fuzzing setting that would use the same techniques. Logically, this is a realistic assumption since fuzzing campaigns that do not continue from where the earlier ones left off (as per previous commits), are not expected to be very effective. Simulating commits instead of using ones from the live repositories is done because (by using `Magma`) we know exactly where the bugs are and when they are triggered by the fuzzers. This is not the case for live commits, where the ground truth knowledge of which bugs exist but not the knowledge of where they are. For the experiment, we also start off with a given set of initial seed inputs for the fuzz targets as it would take quite a few commits to obtain basic coverage of the fuzz targets. We think this is realistic as such corpora will be generated through longer-running fuzzing campaigns anyway (e.g. while initializing a CI/CD pipeline to include continuous fuzzing). For every fuzz target in the experiment,

the set of initial seed inputs is identical to the corresponding starting corpus used in the `Magma` fuzzer benchmark. Last but not least, `AFL++`, `libFuzzer`, and `Honggfuzz` will be working together in the experiment (through corpus sharing between commits, as depicted by Figure 1). This way, they can build upon the progress of their peers, which will be the case in a real continuous fuzzing setup as well.

Results While running the experiment, one observation was that for `lua`, no bugs were found within any of the time budgets. This is due to the fact that there are only four bugs included in `Magma`'s version of `lua`, where other libraries can have up to 20 different bugs available (see Table I). Apparently, these four bugs are quite complex ones, in the sense that finding them through fuzzing requires more time than we allocated to the campaigns. Another observation was that, looking at the overall number of bugs, for some libraries there are no large differences between the 10 minutes and 8 hour campaigns, while for other libraries there is.

Looking at the number of bugs *reached*, Figure 3a shows no clear increase of bugs when the fuzzing campaign is allowed to last longer. For some libraries like `sqlite3` and `libsndfile` the number of bugs does rise, while for others like `poppler` and `php` it declines. While a decline in the number of bugs seems strange, this might be attributed to the stochastic nature of the fuzzing process.

Looking at the number of bugs *triggered*, Figure 3b shows that four out of nine libraries exhibit an increase in the number of bugs while increasing the fuzzing campaign duration. For the library that exhibits the most significant change (`libsndfile`), the number of bugs has only doubled when the duration of the fuzzing campaign is thirty-two times its original duration. For the other five out of nine libraries, there is no significant change to the number of bugs.

Looking at the number of bugs *detected*, Figure 3c shows that five out of nine libraries show a positive change in the number of bugs when increasing the duration of the fuzzing campaigns. Note however, that the difference in the scale of the y-axes of Figures 3a, 3b, and 3c reflects the large differences in the number of bugs *reached*, *triggered* and *detected*. Nonetheless, the difference of a fuzzer *detecting* only one bug or three bugs seems quite significant.

We use *Vargha-Delaney* tests of effect size ($\hat{A}_{12}$) to assess the magnitude of differences in fuzzing effectiveness for the fuzz durations, and *Mann-Whitney U* tests to assess statistical significance of the results. Table III summarizes the effect sizes for the number of bugs *triggered* and *detected*, where statistically significant effect sizes are marked in bold. The *reached* category is left out, as there are no significant differences between any campaign durations in terms of number of bugs found. In both the *triggered* and *detected* category, we observe that *10m* significantly outperforms *5m*, but the increase in effect size for *15m* and *20m* is relatively small. For the *30m* duration, we observe a relatively larger increase in effect size with respect to both the *5m* and *10m* base durations. The next jump in effect size lies at the *4h* mark for *triggered* bugs, and at the *2h* mark for *detected* bugs.

TABLE III: Effect size $\hat{A}_{12}$ for *triggered* and *detected* number of bugs according to *Vargha-Delaney* tests. Statistically significant effect sizes (with p-value < 0.05) according to *Mann-Whitney U* tests are marked in bold. Coloured cells mark the smallest increase in duration where there still are significant differences in the number of bugs found.

triggered		**base duration**							
		5m	10m	15m	20m	30m	1h	2h	4h
test duration	10m	**0.6**							
	15m	**0.61**	0.51						
	20m	**0.6**	0.51	0.49					
	30m	**0.66**	**0.57**	0.56	0.57				
	1h	**0.67**	**0.59**	**0.58**	**0.59**	0.53			
	2h	**0.68**	**0.6**	**0.59**	**0.6**	0.54	0.51		
	4h	**0.72**	**0.65**	**0.64**	**0.64**	**0.59**	0.55	0.55	
	8h	**0.73**	**0.67**	**0.65**	**0.66**	**0.61**	**0.58**	0.57	0.52

detected		**base duration**							
		5m	10m	15m	20m	30m	1h	2h	4h
test duration	10m	**0.57**							
	15m	**0.61**	0.54						
	20m	**0.59**	0.53	0.49					
	30m	**0.67**	**0.61**	**0.57**	**0.59**				
	1h	**0.71**	**0.66**	**0.63**	**0.64**	0.55			
	2h	**0.71**	**0.66**	**0.63**	**0.64**	**0.57**	0.52		
	4h	**0.74**	**0.69**	**0.66**	**0.67**	**0.6**	0.56	0.54	
	8h	**0.74**	**0.7**	**0.68**	**0.69**	**0.63**	**0.59**	0.57	0.53

In order to address **RQ.2**, we found that the gain in fuzzing effectiveness when increasing the time budget for *per-commit* fuzzing campaigns is overall *relatively poor*. Campaigns of 10 minutes (or even better, 30 minutes) can still be *almost as effective* as ones that take multiple hours, especially if such lengthier campaigns are still *regularly* used to fuzz snapshots of the repository.

Employing *per-commit* fuzzing campaigns of *10 minutes* strikes a balance between the desired build time and the effectiveness of fuzzing. We do however advise to adopt a duration of *30 minutes* for pipelines that are not strictly time-constrained. We also advise to employ longer-running fuzzing campaigns during periods where no new commits are made (e.g., at night or during the weekend).

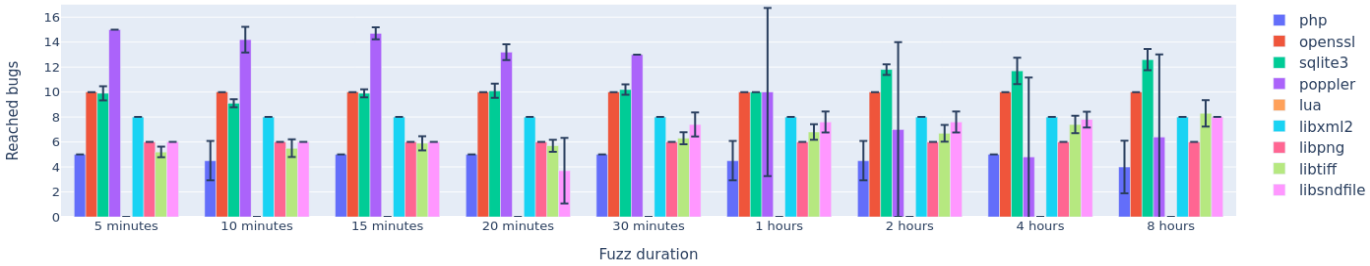
V. Related Work

The tension between testing effectiveness and the compute cycles required in continuous testing practices has been of recent research interest [23], [26], [29]. For instance, Memon et al. [26] showed that testing each commit (coming every second on average) in CI is not sustainable and found out that only a small fraction of the test cases in Google's code base (only 63K among 5.5 Million test cases) had actually ever failed for a given period of commits.

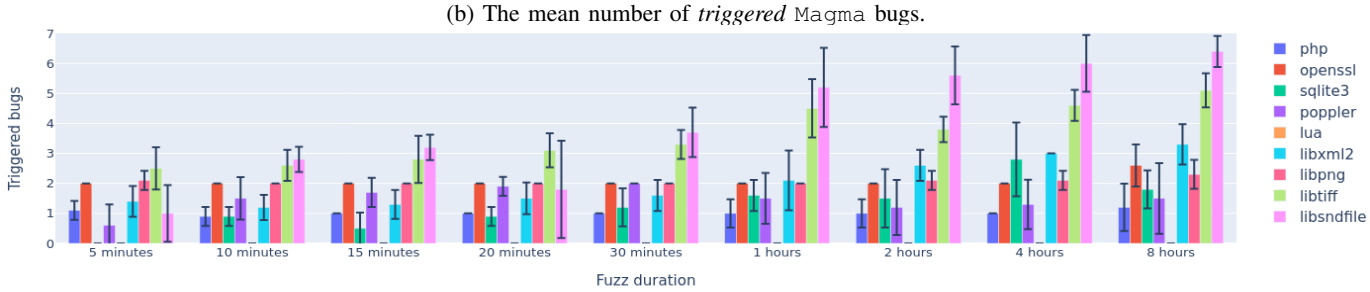
Böhme and Falk [1] found that finding known bugs can be done in half the time with twice the computational resources. However, they also showed that finding new bugs within the same time frame requires exponentially more resources. This means that re-discovering vulnerabilities is cheap but finding new ones is expensive.

Looking at the bug reports from OSS-Fuzz, Zhu and Böhme [33] observed that the recent code changes (i.e. regressions) are responsible for 77 percent of bugs. This means that

(a) The mean number of *reached* Magma bugs.



(b) The mean number of *triggered* Magma bugs.



(c) The mean number of *detected* Magma bugs.

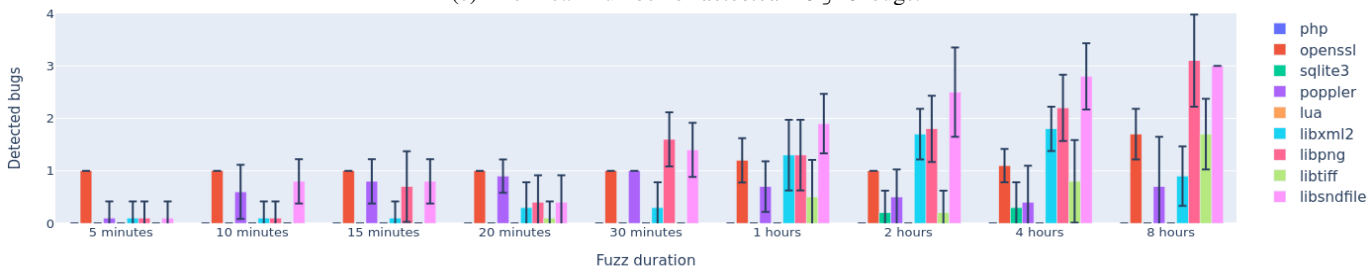


Fig. 3: Fuzzing campaign duration versus the effectiveness of the fuzzing campaign. Note that the difference in the scale of the y-axes of Figures 3a, 3b, and 3c reflects the large differences in the number of bugs *reached*, *triggered* and *detected*.

the majority of bugs will be the result of code that is currently under development. The authors then introduce regression greybox fuzzing in the form of a fuzzer called AFLChurn as a way of focusing the fuzzing effort on code that was changed more recently or more often. They propose to fuzz all commits at once, but to give higher priority to the code that is contained in commits more often or more recently. Regression bugs, which are introduced by recent code changes, can still go undiscovered while using automatic reporting for 68 days on average, and 5 days on the median, as found by [33]. This suggests that bugs of this type are quite hard to detect, and only after a software project is well-fuzzed, additional fuzzing efforts need to be focused on recently changed code.

Campos et al. [3] introduced *Continuous Test Generation* (CTG), which integrates automated unit test generation into CI/CD pipelines. The amount of time to spend on CTG is determined per class, as well as the order in which the classes should be processed. Classes that are not expected to benefit from CTG will not be processed to save resources. From their experiments, the authors report an increase in branch coverage up to 58%, an increase in thrown undeclared exceptions up to 69%, and a reduction in time spent up to 83%. While fuzzing and search-based software testing (e.g. CTG) are both automated software testing techniques and thus have commonalities, there are a number of differences as well, as nicely summarized by Guizzo and Panichella [17]. Regarding the experiments, the work presented in this paper is based on C/C++ programs, while the CTG experiments focus on Java. Additionally, the CTG experiments measure the effectiveness in terms of branch coverage and thrown undeclared exceptions, while the Continuous Fuzzing (CF) experiments measure the actual number of bugs found based on ground-truth knowledge of bugs that have existed in the software repositories in the past. Higher coverage does not necessarily indicate higher effectiveness with regards to finding software bugs [18], [22]. Furthermore, it is an important question whether a thrown exception represents a real fault [3]. CTG skips unchanged classes based on source code changes, while CF skips unchanged fuzz targets based on changes in the machine code. With CF, we report the weighted mean of the reduction in time spent (55%), while CTG reports the maximum reduction in time spent (83%).

VI. Conclusion

In this work, we have investigated the effectiveness of short but frequent fuzzing sessions in CI/CD pipelines. Towards this end, we have conducted two distinct experiments on the open source software libraries from the `Magma` benchmark. For the target selection experiment, we processed as many commits as possible from the live repositories of those libraries, and investigated whether there is a subset of the fuzz targets (per commit) for which the fuzzing sessions can be skipped. For the campaign duration experiment, we extended the `Magma` benchmark in order to make different fuzzers work in an ensemble style for continuous fuzzing with the aim of finding whether shorter fuzzing sessions can still be effective in discovering security vulnerabilities.

The results of the first experiment suggest that even employing a simple technique for selecting the subset of fuzz targets which should be fuzzed, can result in saving more than half (55% on average) of the computational resources needed for the continuous fuzzing effort.

The second experiment shows that while having longer (e.g., 8 hours) fuzzing campaigns can boost the fuzzing effectiveness, we found that the gain in fuzzing effectiveness when increasing the time budget for per-commit fuzzing campaigns is overall relatively poor. More specifically, our experiments show that employing per-commit fuzzing campaigns of *10 minutes* strikes an optimal balance between the recommended testing time and the effectiveness of continuous fuzzing. We do however advise to adopt a duration of *30 minutes* for pipelines that are not that strictly time-constrained.

References

[1] Böhme, M., and Falk, B. Fuzzing: On the exponential cost of vulnerability discovery. In *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (11 2020), Association for Computing Machinery, Inc, pp. 713–724.

[2] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2329–2344.

[3] Campos, J., Arcuri, A., Fraser, G., and Abreu, R. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 55–66.

[4] Code-Intelligence. Ci fuzz - automatically run powerful security tests at each pull request. https://www.code-intelligence.com/product-tour. Accessed on 19.04.2022.

[5] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (2020).

[6] Fowler, M. Continuous integration, 2006. https://martinfowler.com/articles/continuousIntegration.html. Accessed on 19.04.2022.

[7] GitHub. About billing for github actions. https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions#per-minute-rates. Accessed on 29.04.2022.

[8] GitHub. Github actions: Usage limits, billing, and administration. https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration. Accessed on 29.04.2022.

[9] GitLab. GitLab Coverage-guided fuzz testing. https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/. Accessed on 24.04.2022.

[10] Glover, A., Duvall, P., and Matyas, S. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[11] Google. Clusterfuzzlite - simple continuous fuzzing that runs in ci. https://google.github.io/clusterfuzzlite/. Accessed on 19.04.2022.

[12] Google. Hongfuzz - security oriented software fuzzer. https://github.com/google/honggfuzz. Accessed on 19.04.2022.

[13] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://google.github.io/oss-fuzz/. Accessed on 19.04.2022.

[14] Google. OSS-Fuzz - Continuous Integration. https://google.github.io/oss-fuzz/getting-started/continuous-integration/. Accessed on 24.04.2022.

[15] Google. OSS-Fuzz - Issue Tracker. https://bugs.chromium.org/p/oss-fuzz/issues/list?q=-status%3AWontFix%2CDuplicate%20-component%3AInfra&can=1. Accessed on 04.05.2022.

[16] Google. Clusterfuzz. https://google.github.io/clusterfuzz/, 2017. Accessed on 19.04.2022.

[17] Guizzo, G., and Panichella, S. Fuzzing vs sbst: Intersections & differences. *ACM SIGSOFT Software Engineering Notes 48*, 1 (2023), 105–107.

[18] Hazimeh, A., Herrera, A., and Payer, M. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 4*, 3 (2020), 1–29.

[19] Hilton, M., Nelson, N., Tunnell, T., Marinov, D., and Dig, D. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (8 2017), vol. Part F130154, Association for Computing Machinery, pp. 197–207.

[20] Humble, J., and Farley, D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[21] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, pp. 2123–2138.

[22] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security* (10 2018), Association for Computing Machinery, pp. 2123–2138.

[23] Klooster, T. Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines. Master's thesis, University of Groningen, 2021.

[24] Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. Fuzzing: State of the art. *IEEE Transactions on Reliability 67*, 3 (2018), 1199–1218.

[25] LLVM. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed on 19.04.2022.

[26] Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (2017), IEEE, pp. 233–242.

[27] Microsoft. microsoft-onefuzz - A self-hosted Fuzzing-As-A-Service platform. https://github.com/microsoft/onefuzz. Accessed on 24.04.2022.

[28] Paule, C. Securing DevOps: detection of vulnerabilities in CD pipelines. Master's thesis, Stuttgart University, 2018.

[29] Rangnau, T., v. Buijtenen, R., Fransen, F., and Turkmen, F. Continuous security testing: A case study on integrating dynamic security testing tools in CI/CD pipelines. In *24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020, Eindhoven, The Netherlands, October 5-8, 2020* (2020), IEEE, pp. 145–154.

[30] Serebryany, K. OSS-Fuzz - Google's continuous fuzzing software for open-source software. In *USENIX Security* (2017), USENIX Security, Ed.

[31] Wijers, B. 10-minute build extreme programming practice. https://explainagile.com/agile/xp-extreme-programming/practices/10-minute-build/, 2018. Accessed on 19.05.2022.

[32] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., and Vasilescu, B. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017* (2017), G. Rosu, M. D. Penta, and T. N. Nguyen, Eds., IEEE Computer Society, pp. 60–71.

[33] Zhu, X., and Böhme, M. Regression Greybox Fuzzing. In *CCS'21, November 14–19, 2021, Seoul, South Korea* (2021), pp. 883–894.