CollabFuzz: A Framework for Collaborative Fuzzing

Sebastian Österlund*
Vrije Universiteit Amsterdam
s.osterlund@vu.nl

Elia Geretto*
Vrije Universiteit Amsterdam
e.geretto@vu.nl

Andrea Jemmett*
Vrije Universiteit Amsterdam
a.jemmett@vu.nl

Emre Güler
Ruhr-Universität Bochum
emre.gueler@rub.de

Philipp Görz
Ruhr-Universität Bochum
philipp.goerz@rub.de

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

ABSTRACT

In the recent past, there has been lots of work on improving fuzz testing. In prior work, EnFuzz showed that by sharing progress among different fuzzers, they can perform better than the sum of their parts. In this paper, we continue this line of work and present CollabFuzz, a collaborative fuzzing framework allowing multiple different fuzzers to collaborate under an informed scheduling policy based on a number of central analyses. More specifically, CollabFuzz is a generic framework that allows a user to express different test case scheduling policies, such as the collaborative approach presented by EnFuzz. CollabFuzz can control which tests cases are handed out to what fuzzer and allows the orchestration of different fuzzers across the network. Furthermore, it allows the centralized analysis of the test cases generated by the various fuzzers under its control, allowing to implement scheduling policies based on the results of arbitrary program (e.g., data-flow) analysis.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, parallel fuzzing, collaborative fuzzing, ensemble fuzzing, automated bug finding

ACM Reference Format:
Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. CollabFuzz: A Framework for Collaborative Fuzzing. In *14th European Workshop on Systems Security (EuroSec '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447852.3458720

*Equal contribution joint first authors.

EuroSec '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8337-0/21/04.
https://doi.org/10.1145/3447852.3458720

1 INTRODUCTION

In recent years, fuzzing has become an essential tool for finding bugs and vulnerabilities in software. Fuzzers, such as AFL [26] and Honggfuzz [13], have successfully been applied to generate inputs and find bugs in a large number of applications [22]. Recent work in fuzzing [1, 5, 25] has focused on improving the fraction of the target application covered by the fuzzer by implementing new input mutation techniques and branch constraint solving strategies.

Since it is common to use automated bug finding tools to find new bugs in software development scenarios (on every new commit/release), in pentesting scenarios (to find evidence of vulnerabilities), or in server consolidation scenarios (where spare CPU cycles can be dedicated to fuzzing), producing results in bounded time is crucial. Consequently, we target practical use cases where the time budget available for fuzzing is limited and it may be difficult to saturate coverage within that budget. It is, thus, important to look at how existing tools can be utilized in a more efficient way. Large-scale fuzzing efforts, such as OSS-Fuzz [22], have shown that fuzzing scales well with additional computing resources in order to find security-relevant bugs in software. Moreover, researchers further improved the speed of fuzzing by parallelizing and distributing the fuzzing workload [17, 18, 26]. Typically, in these setups, multiple instances of the same fuzzer run in parallel and their results are periodically synchronized [22]. In contrast, EnFuzz [7] demonstrated that running combinations of *different* fuzzers in parallel leads to a noticeable variation in performance, paving the way for further improvement. Intuitively, this makes sense as fuzzers that have different properties and advantages in some areas often come with disadvantages in others. Hence, a *collaborative fuzzing* run using a combination of fuzzers with different abilities can outperform multiple instances of the same fuzzer.

Given a set of COTS (*commercial-off-the-shelf*) fuzzers and a number of cores, collaborative fuzzing has two possible ways to improve beyond simply running the fuzzers as independent tasks in a predetermined configuration. First, we can synchronize the fuzzers so that a good input found by one can benefit the others. Second, we can determine the right mix of fuzzers to run to combine their strengths—e.g., one fuzzer may be better at solving some constraints and another may be better at solving others. While the second direction has already been explored in the literature [11], to our knowledge, optimally sharing test cases represents a resource

allocation problem that has not yet received the attention it deserves. Yet, this problem is important, as each fuzzer has its own strengths and weaknesses that influence the time it takes to get past specific obstacles in the program. For example, a heavyweight symbolic execution-based approach might be better at solving certain constraints, while a lightweight greybox fuzzer might be better at rapidly exploring a program under test.

In this paper, we investigate whether *test case scheduling* on a fuzzer level (i.e., selectively handing out test cases to particular fuzzers) can improve the overall results of a collaborative fuzzing campaign. To this end, we introduce CollabFuzz, a collaborative fuzzing framework capable of orchestrating fuzzing campaigns of a diverse set of fuzzers and deciding how these fuzzers share their progress with each other. Using CollabFuzz, we implement a number of relatively simple test case scheduling policies and evaluate whether such policies can improve fuzzing performance.

Summarizing, we make the following contributions:

- We present CollabFuzz, a distributed collaborative fuzzing framework.
- We implement and evaluate a number of test case scheduling policies on top of CollabFuzz.
- We release CollabFuzz as open source software, available at https://github.com/vusec/collabfuzz.

2 BACKGROUND

2.1 Fuzzing

Fuzzing is the process of automatically finding bugs by generating randomly mutated inputs and observing the behavior of the application under test. Current fuzzers are mainly *coverage-guided*, meaning that they try to generate inputs to maximize code coverage. These fuzzers are generally classified into three categories: *blackbox*, where the fuzzer has no inherent knowledge of the target program (with the advantage of being fast and easily compatible, but with less opportunity for generating high-quality test cases); *whitebox*, with a focus on heavyweight and high-quality input generation (but suffering from scalability and compatibility issues); and *greybox*, which combines the strengths of the first two, trying to be compatible while still using some lightweight analysis to produce high-quality test cases.

Besides improving the fuzzing techniques themselves, the growing code size of projects like web browsers have required developers to scale performance by running fuzzers in parallel [18, 22, 26].

When automatically testing large applications like Chrome, with over 25 million lines of code [8], it becomes increasingly clear that even optimized fuzzing tools need access to multi-core and distributed systems to maximize code coverage and their likelihood of finding bugs, as shown e. g. by the ClusterFuzz project [22].

For this purpose, fuzzers like AFL ship with a parallel-mode [18, 26], where multiple AFL instances share a corpus and thus synchronize their efforts. Although this approach does indeed increase code coverage, it does not solve some of the limitations inherent to AFL. For instance, whenever AFL has difficulties solving magic bytes comparisons, multiple instances of AFL will still have a low probability of solving these conditions.

2.2 Collaborative fuzzing

To counter the limitations imposed by using one single type of fuzzer, EnFuzz [7] introduces *ensemble fuzzing*. The authors demonstrate that combining a *diverse* set of fuzzers leads to greater code coverage than running multiple instances of the *same* fuzzer. The boost in performance seems to stem from the symbiosis of the different fuzzing techniques, where the combination of fuzzers are more likely to cancel out individual disadvantages. Recently, Güler & al. [11] showed how it is possible to automatically select a *good* set of diverse fuzzers to use in such a scenario.

While state-of-the-art fuzzers typically focus on increasing code coverage, a recent area of research focuses on minimizing the latency of reaching specific or interesting parts of the program [21]. Within such a constrained budget, some combinations of fuzzers most likely provide a higher return on investment than others.

Besides looking at *which* fuzzers to run together, there is also the question of *how* they should collaborate. Is handing out all the generated test cases to all the fuzzers always the best choice? Certain fuzzers, such as QSYM [25], are good at finding new branches, but their performance can degrade significantly if they get too many (low-quality) test cases. We thus investigate how selectively handing out test cases—or, in other words, *test case scheduling*—can improve the performance in a collaborative setting.

In summary, CollabFuzz is a framework that allows multiple fuzzers to collaborate on a large scale, while a central scheduling component can optimize the fuzzing process by improving the exchange of information between fuzzers—in other words improve resource allocation.

3 DESIGN

Since fuzzing is a parallelizable task, it is reasonable to run several fuzzers collaboratively to improve code coverage and bug finding. Without test case scheduling, a large fraction of each fuzzer's execution is spent to just get to a point in the target program that another fuzzer may have already found. This is true not only for several instances of a fuzzer, despite the randomness involved in fuzzing, but even for different fuzzers. Indeed, different fuzzers have different strengths that influence the time it takes for them to get past specific obstacles in the program, but are inherently similar. With CollabFuzz, we want to implement a generic, flexible fuzzer orchestration framework that can be be used for large-scale fuzzing campaigns as well as fuzzer evaluation. In contrast to prior fuzzer orchestration efforts, such as OSS-fuzz [22] and FuzzBench [20], CollabFuzz allows multiple different fuzzers to collaborate while supporting the user in running fine-grained analysis during the fuzzing campaign.

In this paper, we showcase how we can use CollabFuzz to implement a number of *test case scheduling* mechanisms, allowing the manager to selectively hand out test cases according to an informed scheduling policy.

We identify three main critera for CollabFuzz's design:

(1) Flexibility. We want a framework that can easily be extended by future work. As such, we design the different components to also be reusable for other uses than presented here.

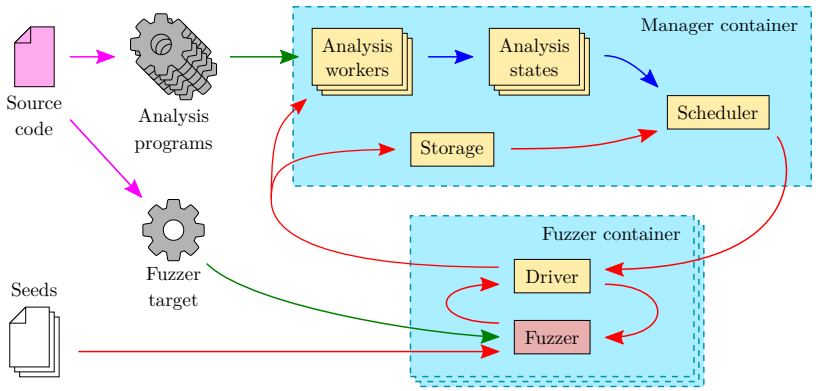

Figure 1: An overview of CollabFuzz and its components.

(2) Reproducibility. In fuzzing, being able to reliably repeat experiments is paramount. CollabFuzz uses Docker to achieve a reproducible environment for all the fuzzing targets.
(3) Scalability. We want to support large-scale fuzzing. As such, CollabFuzz allows the framework to run in a distributed setting, making it easy to scale fuzzing campaigns to large clusters.

At a high level, the central scheduling *manager* interacts with the fuzzer drivers to control the fuzzers. The manager hands out test cases to the different fuzzer *drivers* that interact with the fuzzer in question, in turn allowing the fuzzer to mutate the input trying to increase coverage. When a fuzzer finds a new test case, the driver sends this test case back to the manager. The subsequent action is determined by the scheduling policy and informed by various analyses. In a typical scenario, if the generated test case provides new coverage (as decided by an analysis pass), the scheduler will hand out the new test case to one or more fuzzers or cache it for scheduling at a later stage.

Specifically, for a single run, CollabFuzz has to be provided with the source of the program under test, a set of seeds for that program, a combination of fuzzers to run, and a policy to coordinate them. As shown in Figure 1, the source will be compiled into several instrumented binaries (which will be used to analyze test cases) and all the binaries each fuzzer requires. At this point, the framework can be started. While running, the following three components are of interest:

(1) Central *manager*. This component schedules test cases to the different fuzzers, depending on the scheduling policy set by the user. We discuss a number of scheduling policies we implemented in CollabFuzz in Section 5.
(2) *Fuzzer*. We can add any off-the-shelf fuzzer to the mix. The fuzzer fetches test cases and mutates them, typically optimizing for finding new coverage.
(3) Fuzzer *driver*. The driver interacts with the off-the-shelf fuzzer, handing it new test cases when the manager needs to schedule them and reporting new findings back to the manager.

Upon receiving a new test case from one of the drivers, the *manager* first places the incoming test case in *storage*, after which it starts up a number of analysis jobs that are defined by the *scheduler*. For example, a scheduler might require coverage analysis, in which case the manager would start up a coverage-gathering job on an *analysis worker* for the incoming test case. The results of these jobs are stored as *analysis states*, which can later be queried by the scheduler when making a scheduling decision.

The scheduler is invoked both periodically (with a user configurable interval) and in an event-driven fashion when a new test case arrives at the manager, allowing for maximum flexibility when implementing scheduling policies. When the scheduler is invoked, it can reason over the stored analysis states and then make an informed decision on whether to hand out zero or more test cases to any running fuzzers. When the scheduler makes its decision, the selected test cases are sent out to the corresponding fuzzer *drivers*. The new test cases are then inserted into the fuzzer queue. We further discuss the design choices and implementation details in the following sections.

4 IMPLEMENTATION

CollabFuzz consists of three components to facilitate the collaboration between fuzzers: the scheduling manager, to coordinate and schedule different fuzzers and inputs; the fuzzer drivers, to allow the fuzzers to interact with the scheduling manager; and the (off-the-shelf) fuzzers.

We implemented the scheduling manager in Rust (about 6k LOC) and C++ (about 1k LOC), while the fuzzer drivers are written in Python (about 2k LOC). Each fuzzer runs in its own Docker [19] container and communicates with the scheduling manager over ZeroMQ [12] sockets, allowing the whole setup to run in a large-scale distributed setting. The framework is designed in an extensible way, allowing developers of new fuzzers to easily add support by simply creating a new container image.

Scheduling Manager. The central scheduling manager listens for incoming new test cases from the fuzzers. When a test case arrives, a scheduler is invoked, which decides how to react to the event. It is also possible to let the manager invoke the scheduler periodically, allowing for a flexible way to implement different schedulers. When a scheduler is invoked, it typically selects one or more test cases to send out to a group of fuzzers.

The scheduler registers a number of *analyses* that are executed for incoming test cases. These analyses (such as coverage tracing) are performed by *analysis workers* (which can be distributed over the network) and stored globally in *analysis states* that the scheduler can query. This design allows for flexible and possibly heavyweight analysis without a significant performance penalty. For example, some schedulers might require data-flow analysis as part of their scheduling decision-making. In such cases, the scheduler would register a data-flow analysis pass, which is run on every incoming test case which is deemed interesting by one of the fuzzers.

Fuzzer Driver. We implemented a generic fuzzer driver (using Python), which listens to files created in a number of directories (e.g., queue, crashes, hangs for AFL). When a new inotify event is dispatched, the driver sends the new test case to the scheduling manager over a ZeroMQ socket. In a similar fashion, the driver also listens for incoming messages from the scheduling manager, placing these incoming test cases in a specified directory. This generic design allows this single driver to work with a variety of fuzzers. CollabFuzz currently supports *AFL, AFLFast, FairFuzz, QSYM, radamsa, Honggfuzz*, and *libFuzzer*. We extended libFuzzer and Honggfuzz with an AFL-style synchronization mechanism to allow all fuzzers to share test cases. Each target application for a particular fuzzer is based on a Docker image. Each fuzzing campaign is configured using a YAML file, allowing for repeatable runs of the campaign.

Analyses. As mentioned before, each scheduler bases its decisions on data produced by a series of static and dynamic analyses. These analyses are implemented as LLVM [15] passes and thus require source code. Despite the absence of technical limitations in implementing them at the binary level, we chose this approach to ease development.

As an example, we implemented the following analyses:

Global coverage Extracts the exact edge coverage of a test case and then aggregates it during a single campaign.

Test case benefit Implements the design described in Section 5 using DataFlowSanitizer and a static interprocedural control flow graph.

Instruction count Uses a modified version of DataFlowSanitizer to dynamically compute the length of the dynamic backward slice for each instruction, storing the minimum.

New analysis passes can easily build on top of the individual building blocks in our existing passes. The data generated by the analysis passes and all scheduler events are stored in a SQLite database, which can be queried for further analysis.

Patches for compatibility. CollabFuzz includes fuzzing targets as Docker containers. We include containers for every target binary in LAVA-M [9], Binutils, and Google fuzzer-test-suite [10] for every fuzzer that we used (AFL, AFLFast, FairFuzz, QSYM, Honggfuzz, libFuzzer, radamsa, lafIntel), allowing for a consistent environment when performing our benchmarks. To make the target programs compatible with our DFSan-based analysis passes, we had to patch a number of issues in the build systems (e.g., of the Google fuzzer-test-suite), as well as in DFSan. We compiled all C++ programs against LLVM's `libcxx` to be able to get adequate DFSan coverage. Furthermore, we had to patch a number of fuzzers to allow for external test case syncing at runtime.

5 CASE STUDY: TEST CASE SCHEDULING

As a case study of applying CollabFuzz, we evaluate whether coarse-grained fuzzer-level scheduling (i.e., the scheduler has no insight into the fuzzer's internal queue) of test cases can be utilized to improve the overall results of a collaborative fuzzing campaign. We consider EnFuzz as a baseline, and see whether other strategies of synchronizing (i.e., scheduling) the corpus yields a noticeable effect on the overall result of the fuzzing campaign.

To showcase how CollabFuzz allows for diverse scheduling policies, we implemented four relatively simple test case scheduling policies. We believe that more informed and effective policies are possible, but leave this as future work. Our goal is to demonstrate that CollabFuzz can be used as a platform for reasoning about such scheduling and resource allocation policies.

EnFuzz scheduler. This scheduler is a reimplementation of the approach described by Chen & al. [7]. The scheduler continuously receives new test cases from each single fuzzer and, every 2 minutes, it forwards them to all the other fuzzers participating in the collaboration.

Broadcast scheduler. This second scheduler is a simple optimization over the one employed by EnFuzz. It simply eliminates the synchronization delay by forwarding test cases as soon as they are received by the coordination server. The intuition behind this approach is that the 2 minutes delay may build up over the run and thus negatively influence the increase in global coverage over time. Since CollabFuzz is continuously analyzing new incoming test cases and caching the analysis results, there is no need for a 2-minute delay for coverage information analysis.

Benefit scheduler. In contrast to the previous scheduler, the benefit scheduler introduces a synchronization delay in order to focus the fuzzers on important test cases and delay less interesting ones. In detail, the benefit scheduler fills a priority queue with all the received test cases, but flushes only 1% of it every 5 seconds (these parameters ensure a continuous small stream of test cases).

The prioritization happens based on *benefit*, a novel metric that, given a test case, we define as the count of unseen basic blocks in the program that are reachable from the frontier of the test case. In turn, we define the *frontier* of a test case as the set of basic blocks in the trace for that test case which match the following criteria:

(1) They have at least one unseen basic block as neighbor in the interprocedural CFG.
(2) The terminator instruction of the basic block from which they can be reached is tainted by the input given by the fuzzer.

The intuition behind this scheduler is that focusing the fuzzing effort on test cases with a high benefit can potentially increase the global coverage more rapidly. This is particularly important for fuzzers that employ heavyweight analyses and thus have a lower execution count, like QSYM, since they can be focused on important test cases first, without wasting cycles on fuzzing, for example,

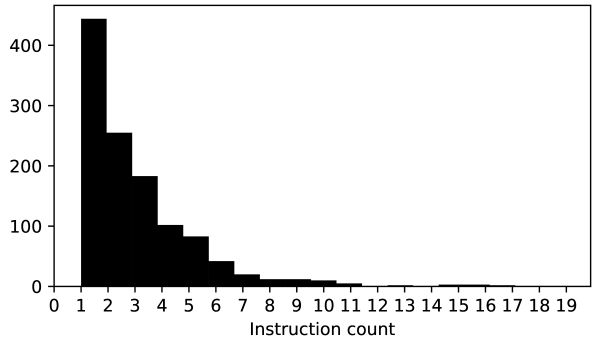

Figure 2: Distribution of the cost metric at the end of a run on OBJDUMP. While a lot of basic blocks have a cost of 1, there still exist plenty of harder blocks to solve.

error-handling code. An approach, similar in nature, which prioritizes certain test cases based on some heuristic has been previously presented by DigFuzz [27].

Cost-benefit scheduler. This last scheduler partially overlaps with the previous one because it uses the same priority queue system, but it changes the way in which the priority is calculated. Apart from the benefit metric defined before, it also relies on *cost*, another novel metric that, given a basic block, we define as the minimum number of instructions in the trace for that test case which match the following criteria:

(1) They manipulate their arguments in some way, e.g. arithmetic operations, and do not just move them around in memory, e.g. `store` instructions.
(2) They belong to the dynamic backward slice of the terminator instruction of the basic block for which the cost metric is being computed.

These two metrics are then aggregated in the following way:

$$\text{cost\_benefit}(t) = \frac{\sum_{b \in \text{frontier}(t)} \text{benefit}(t)/\text{cost}(b)}{|\text{frontier}(t)|} \quad (1)$$

The intuition behind this scheduler is that the benefit that can be produced when solving a specific branch constraint needs to be weighted against the difficulty to solve that constraint, which is approximated with the cost metric. For example, a very beneficial constraint that is almost impossible to solve is probably not worth focusing on. In turn, the intuition behind the cost metric is that the more instructions concur to the computation of a single value, the more complex the constraint will be. An example of the distribution of the cost metric at the end of a run is shown in Figure 2.

5.1 Evaluation of schedulers

We performed experiments on 32-core/64-thread AMD ThreadRipper 2990WX processors with 128GB of RAM. For each experiment, we allocated 4 hardware threads (running 4 fuzzers + framework/-drivers). We ran the fuzzers inside Docker containers and enabled core pinning for AFL. We ran each experiment for 10 hours with 10 repetitions. We show the median of the branch coverage count and the area-under-the-curve (AUC) of the branch coverage at the end of the campaign. We also indicate whether the results are statistically significant using the Mann-Whitney U-test as suggested by Klees et al. [14].

We evaluate our Cost-Benefit against the EnFuzz scheduling policy on the Google fuzzer-test-suite with the well-performing diverse selection of fuzzers suggested by Cupid [11] (AFL,FairFuzz, libFuzzer, QSYM). The results are presented in Table 1. We observed similar initial results for the other scheduling policies.

Not surprisingly, our results show that, with the given selection of fuzzers, the coverage at saturation is typically similar regardless of the scheduling policy, with no statistically significant difference for the overwhelming majority of our target programs. In the end, the achieved coverage that a set of fuzzers reaches is determined by the individual mutation techniques of the fuzzers. As such, the manner in which they share their progress (as long as it is shared somehow) has little influence on how much of the target program can possibly be explored, if coverage saturation is reached within the time limit.

Despite the achieved coverage being the same, we also investigate whether scheduling policies can improve *how quickly* said coverage is reached. In other words, can different scheduling policies affect the latency of reaching a certain amount of coverage?

The AUC metric shows the evolution of coverage over time. The more coverage is found earlier on in the campaign, the higher the AUC will be. On the other hand, reaching the same end coverage at a later time will result in a lower AUC metric. To make the AUC metric somewhat more tangible, we derive some samples from it. Namely, looking at the latency of achieving a partial amount of coverage is a useful indication of the real-world speedup a user can expect when fuzzing with limited time and resources. We show the difference in latency to achieve the 90, 95, 97, and 99th percentile of the total coverage. For example, as shown in Table 2, the EnFuzz scheduler reaches 90% of its end coverage 13% faster than Cost-Benefit.

At a first glance, the differences appear significant. Namely, the EnFuzz scheduler seems to outperform the Cost-Benefit scheduler in every case. However, after some more thorough analysis of the data obtained through CollabFuzz, we can conclude that there is *no statistical significant difference* in the AUC between the different schedulers (as can be seen in Table 1) and thus the aforementioned latency deviations can likely be attributed to randomness. Latency-wise there can be a large difference in when the different setups reach a particular milestone. However, in practice, this large difference in latency might simply be due to a very small skew (even due to a single branch) in the distribution of when coverage is found. For example, our broadcast scheduler performs similar to the EnFuzz scheduler, despite it cutting out the 2 minute synchronizationb time-window of EnFuzz.

Overall, our results show that the different schedulers we have presented do not significantly affect the overall achieved coverage of a fuzzing campaign, nor do they affect the AUC of the coverage. As such, we can conclude that a fuzzer-level coarse-grained scheduling of test cases is unlikely to yield any significant performance improvements. Nonetheless, we believe our analysis is an important first step to study scheduling policies in a collaborative fuzzing scenario and CollabFuzz can serve as a basis to quickly evaluate a variety of more sophisticated policies in future work.

Table 1: Median branch coverage for different scheduling policies. ↑ indicates that Cost-Benefit was significantly better than EnFuzz; ↓ indicates that EnFuzz performed better; ✗ means no statistically significant difference.

Binary	Cost-Benefit	EnFuzz	p-val	AUC p-value
c-ares	45	45	—	✗
guetzli	5047.5	4983.5	✗	✗
json	1544	1545	✗	✗
libarchive	5264	5424.5	↓	✗
libpng	1514	1516.5	✗	✗
libxml2	5407.5	5354.5	↑	✗
openssl-1.0.2d	1442	1442	—	✗
openssl-1.1.0c	1281	1281	✗	↓
openthread	1915.5	1912.5	✗	✗
proj4	5773	5882	✗	✗
sqlite	1733	1733	—	✗
woff2	2950.5	3021.5	↓	↓
Geomean final coverage	1847.23	1855.05		

Table 2: Speedup in achieving partial coverage compared to the EnFuzz scheduler.

Coverage	Cost-Benefit
90%	-0.32%
95%	-12.75%
97%	-18.78%
99%	-0.76%

6 RELATED WORK

Existing work on fuzzing has investigated how prioritizing certain test cases can improve the performance within one single fuzzer. FairFuzz [16] prioritizes input mutations, such that "rare" branches are given priority over commonly exercised branches. In AFLFast [3], the authors model fuzzing as a Markov model, and use it to steer fuzzing towards low-frequency paths. In contrast, CollabFuzz does not look at the individual fuzzers at the queue level, but rather implements scheduling policies on a global level over a variety of different fuzzers. CollabFuzz's scheduling policies can be applied to any off-the-shelf fuzzer, and requires little or no modification to the actual fuzzer. In [24], the authors evaluate a large number of scheduling algorithms for blackbox fuzzing. AFLGo [2], HawkEye [4], and ParmeSan [21] all use static analysis and instrumentation to allow for prioritization (i.e., scheduling) of test cases that lead to coverage of pre-specified locations in the target program.

Hybrid fuzzing [23, 25] shows that augmenting lightweight greybox fuzzing with more heavy-weight analysis (e.g., symbolic execution) can yield more bugs without significantly slowing down the whole process. This approach can be seen as a type of test case scheduling, where the hard-to-solve cases are offloaded to the heavyweight analysis. In fact, these schemes can be easily expressed as a scheduling policy in CollabFuzz. Recent work by Chen & al [6] and Zhao & al [27] show how adaptive scheduling policies can further improve hybrid fuzzing.

EnFuzz [7] introduces *ensemble fuzzing*, i.e., having a diverse set of fuzzers collaborate, showing how selecting an ensemble of diverse fuzzers can increase code coverage. In this paper, we generalize the intuition provided by EnFuzz and present CollabFuzz, a framework that can model such collaboration between different fuzzers in a more generic fashion.

7 DISCUSSION & FUTURE WORK

In this paper, we have limited ourselves to COTS fuzzers with the explicit goal of not making significant modifications to the fuzzers themselves. We are thus limited to the interface that the selected fuzzers provide. This means that we can select which test case to hand out to the fuzzer, but we cannot select which test case any particular fuzzer should work on at a given moment. With a more fine-grained interface, the scheduler could have more control over, for example, the kinds of branches to target. While the scheduling policies presented in this paper did not yield a statistically significant improvement, we see ample opportunity for improving test case scheduling by introducing such fine-grained control mechanisms.

Furthermore, in our current implementation, the set of selected fuzzers is static over the whole run. In some cases, changing the resource allocation among the fuzzers over the fuzzing campaign might yield better results. Our current CollabFuzz prototype has rudimentary support for this, but we have limited the scope of this study to a static set of fuzzers.

8 CONCLUSION

We have presented CollabFuzz, a collaborative fuzzing framework that allows multiple fuzzers to share their progress towards one end goal. By using CollabFuzz's orchestration of large-scale fuzzing campaigns on a cluster, we have shown that coarse-grained test case scheduling between fuzzers has a negligible effect on the result of the fuzzing campaign. Nevertheless, CollabFuzz enables developers to easily express different fuzzing techniques by means of scheduling policies and allows them to easily collect fuzzer statistics for further analysis. The source code for our CollabFuzz prototype is available at https://github.com/vusec/collabfuzz.

ACKNOWLEDGMENTS

We would like to thank the anonynous reviewers for their constructive feedback. This work was supported by Cisco Systems, Inc. through grant #1138109 and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CaSa – 390781972. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct). This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based Greybox Fuzzing As Markov Chain. In *IEEE Transactions on Software Engineering*.

[4] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: towards a desired directed grey-box fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*.

[5] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*.

[6] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92.

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*.

[8] Chrome25 [n.d.]. The Chromium (Google Chrome) Open Source Project on Open Hub: Languages Page. https://www.openhub.net/p/chrome/analyses/latest/languages_summary. Accessed: March 31, 2021.

[9] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (S&P)*.

[10] Google, Inc. 2018. fuzzer-test-suite. https://github.com/google/fuzzer-test-suite. Accessed: March 31, 2021.

[11] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*. https://doi.org/10.1145/3427228.3427266

[12] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

[13] Honggfuzz [n.d.]. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: March 31, 2021.

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Iinternational Symposium on Code generation and optimization*.

[16] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In *ACM International Conference on Automated Software Engineering (ASE)*.

[17] Yang Li, Chao Feng, and Chaojing Tang. 2018. A Large-scale Parallel Fuzzing System. In *International Conference on Advances in Image Processing*.

[18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.

[19] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[20] László Szekeres Jonathan Metzman, Abhishek Arya, and L Szekeres. 2020. FuzzBench: Fuzzer benchmarking as a service. *Google Security Blog* (2020).

[21] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*. Paper=https://download.vusec.net/papers/parmesan_sec20.pdfCode=https://github.com/vusec/parmesan

[22] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*.

[23] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Symposium on Network and Distributed System Security (NDSS)*.

[24] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *ACM Special Interest Group on Security, Audit and Control (SIGSAC)*.

[25] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*.

[26] Michał Zalewski. [n.d.]. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: March 31, 2021.

[27] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.