My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers

David Paaßen(✉), Sebastian Surminski, Michael Rodler, and Lucas Davi

University of Duisburg-Essen, Duisburg, Germany
{david.paassen,sebastian.surminski,michael.rodler,lucas.davi}@uni-due.de

Abstract. Fuzzing has become one of the most popular techniques to identify bugs in software. To improve the fuzzing process, a plethora of techniques have recently appeared in academic literature. However, evaluating and comparing these techniques is challenging as fuzzers depend on randomness when generating test inputs. Commonly, existing evaluations only partially follow best practices for fuzzing evaluations. We argue that the reason for this are twofold. First, it is unclear if the proposed guidelines are necessary due to the lack of comprehensive empirical data in the case of fuzz testing. Second, there does not yet exist a framework that integrates statistical evaluation techniques to enable fair comparison of fuzzers.

To address these limitations, we introduce a novel fuzzing evaluation framework called SENF (Statistical EvaluatioN of Fuzzers). We demonstrate the practical applicability of our framework by utilizing the most wide-spread fuzzer AFL as our baseline fuzzer and exploring the impact of different evaluation parameters (e.g., the number of repetitions or run-time), compilers, seeds, and fuzzing strategies. Using our evaluation framework, we show that supposedly small changes of the parameters can have a major influence on the measured performance of a fuzzer.

1 Introduction

Fuzzing approaches aim at automatically generating program input to assess the robustness of a program to arbitrary input. The goal of a fuzzer is to trigger some form of unwanted behavior, e.g., a crash or exception. Once a program fault occurs during the fuzzing process, a developer or analyst investigates the fault to identify its root cause. Subsequently, this allows the software vendor to improve the quality and security of the software. One of the most prominent fuzzers, called American Fuzzy Lop (AFL) [41], has discovered hundreds of security-critical bugs in a wide variety of libraries and programs.

Following the success of AFL, various other fuzzers have been proposed which aim to outperform AFL by implementing new and improved fuzzing techniques (e.g., [8,19,27,29]). However, it remains largely unclear whether the claim of

© Springer Nature Switzerland AG 2021
E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12972, pp. 173–193, 2021.
https://doi.org/10.1007/978-3-030-88418-5_9

improving the overall fuzzing performance is indeed true. This is because accurately evaluating a fuzzer is challenging as the fuzzing process itself is non-deterministic. Hence, comparing single runs or multiple runs using simple statistical measurements such as average values can lead to false conclusions about the performance of the evaluated fuzzer. Similarly, deriving the number of potentially discovered bugs based solely on coverage measurements and the number of program crashes does not necessarily map to the effectiveness of a fuzzer. For instance, Inozemtseva et al. [25] show that there is no strong correlation between the coverage of a test suite and its ability to detect bugs. Additionally, there are fuzzing approaches that prioritize certain program paths instead of maximizing the overall coverage [7,10,39]. Such approaches cannot be evaluated using overall code coverage as a measurement.

A study by Klees et al. [26] shows that existing evaluation strategies do not consider state-of-the-art best practices for testing randomized algorithms such as significance tests or standardized effect sizes. They also provide a list of recommendations. However, these recommendations are mainly derived from known best practices from the field of software testing or from a small set of experiments on a small test set. Nevertheless, as we will show in Sect. 4, recent fuzzing proposals still do not consistently follow recommendations regarding the employed statistical methods and evaluation parameters (e.g., run-time or number of trials). Since the goal of the recommendations is to ensure that the reported findings are not the results of randomness, it remains unclear whether we can trust existing fuzzing experiments and conclusions drawn from those experiments.

Another important aspect of any fuzzer evaluation concerns the employed test set. Several research works introduced test sets such as LAVA-M [14], Magma [22], or the Google Fuzzer Suite [20]. Ideally, a test set should contain a wide variety of different programs as well as a set of known bugs covering various bug types including a proof-of-vulnerability (PoV). This is crucial to enable accurate assessment on the effectiveness and efficiency of a fuzzer as a missing ground truth may lead to overestimating or underestimating the real performance of a fuzzer. We analyze these test sets in detail in Sect. 5.2 as the test set selection is crucial for evaluating and comparing fuzzers.

Lastly, existing evaluation strategies lack uniformity for evaluation parameters such as the number of trials, run-time, and size of the employed test set and the included bugs. As it is still unknown how these parameters affect the fuzzer evaluation in practice, fuzzing experiments are commonly executed using a wide variety of different parameters and evaluation methods. This may not only affect the soundness (e.g., due to biases caused by the choice of parameter) of the results but also makes it even harder to compare results across multiple studies.

Our Contributions. In this study, we address the existing shortcomings of fuzzing evaluations. To do so, we review current fuzzing evaluation strategies and introduce the design and implementation of a novel fuzzing evaluation framework, called SENF (Statistical EvaluatioN of Fuzzers), which unifies state-of-the-

art statistical methods and combines them to calculate a ranking to compare an arbitrary number of fuzzers on a large test set. The goal of our framework is twofold. First, we aim to provide a platform that allows us to easily compare a large number of fuzzers (and configurations) on a test set utilizing statistical significance tests and standardized effect sizes. Contrary to existing frameworks, such as UNIFUZZ [28], SENF provides a single ranking which allows for an easy comparison of the overall performance of fuzzers. Second, due to the lack of comprehensive empirical data we test if following the recommended best practices is necessary or if we can loosen the strict guidelines to reduce the computational effort needed to compare different fuzzing algorithms without impairing the quality of the evaluation which was not possible with the data provided by Klees et al. [26].

To show the applicability of SENF, we build our evaluation based on the most prominent fuzzer, namely AFL [41] and its optimizations, as well as the popular forks AFLFast [8], Fairfuzz [27], and AFL++ [15]. This allows us to argue about the usefulness and impact of the proposed methods and techniques as AFL is commonly used as the baseline fuzzer in existing fuzzing evaluations. We ensure that all tested fuzzers share the same code base which allows us to precisely attribute performance differences to the changes made by the respective fuzzer or optimization technique.

We provide an extensive empirical evaluation of the impact of fuzzing parameters. In total, we ran over 600 experiments which took over 280k CPU hours to complete. To the best of our knowledge, this is currently the largest study of fuzzers published in academic research.

In summary, we provide the following contributions:

- We implement a fuzzing evaluation framework, called SENF, which utilizes state-of-the-art statistical evaluation methods including p-values and standardized effect sizes to compare fuzzers on large test sets.
- We conduct a large-scale fuzzer evaluation based on a test set of 42 different targets with bugs from various bug classes and a known ground truth to quantify the influence of various evaluation parameters to further improve future fuzzer evaluations.
- We open-source SENF [32], containing the statistical evaluation scripts, the result data of our experiments, and seed files to aid researchers to conduct fuzzer evaluations and allowing reproducibility of our study.

2 Background

In this section, we provide background information on the most relevant fuzzing concepts and discuss how these are implemented in case of the popular fuzzer AFL [41].

Fuzzers are programs that need to decide on a strategy to generate inputs for test programs. The inputs should be chosen in such a way that they achieve as much coverage of the program's state space as possible to be able to detect abnormal behavior that indicates an error. Fuzzers are commonly categorized

into black-box, white-box, and grey-box fuzzers. Where black-bock fuzzers (e.g., zzuf [23]) try to maximizes the number of executions while white-box fuzzers (e.g., KLEE [9]) make heavy use of instrumentation and code analysis to generate more significant input. Grey-box fuzzers (e.g., AFL [41]) try to find a balance between the executions per second and time spend on analysis.

One of the most well-known fuzzers is called American fuzzy lop (AFL) and is a mutation-based coverage-guided grey-box fuzzer. It retrieves coverage feedback about the executed program path for a corresponding test input. Since its creation, AFL discovered bugs in more than 100 different programs and libraries [41] confirming its high practical relevance to improve software quality and security.

Given the influence of AFL in the fuzzing area, we take a closer look at its architecture. AFL includes a set of tools that act as drop-in replacements for known compilers, e.g., as a replacement for `gcc` AFL features `afl-gcc` which is used to add code instrumentation. The instrumentation provides crucial information such as the branch coverage and coarse-grained information about how often a specific branch has been taken.

The fuzzing process can be divided into four different core components which can also be found in many existing mutations-based grey-box fuzzers, including forks of AFL: ① *Search strategy*: The search strategy selects an input (e.g., one of the initial seeds) that is used in the mutation stage to generate more test inputs. ② *Power schedule*: The power schedule assigns an energy value which limits the number of new inputs generated in the mutation stage. The idea is to spend more time mutating input that is more likely to increase the code coverage. ③ *Mutations*: The mutation stage changes (part of) the selected input to produce new inputs which are then executed by the program under test. AFL has two different mutation stages. The *deterministic stage* does simple bit flips or inserts specific values such as `INT_MAX`. In the *havoc stage*, AFL executes a loop that applies different mutations on the selected input, e.g., inserting random data or trimming the input. ④ *Select interesting input*: After executing a new input, the fuzzer collects the feedback data and decides if the newly generated input is interesting, i.e., whether or not the input should be mutated to generate new inputs.

Successors of AFL commonly implement their improvements as part of one or more of the discussed core components. In Sect. 6.1 we describe the changes implemented by the different fuzzers we test in our evaluation in more detail.

To address the problem of inputs being rejected due to rigorous input checks, fuzzer leverage seed files which provide initial coverage and useful inputs for the mutation stage. Thus, a fuzzer does not need to learn the input format from scratch. To generate a set of seed files, one can either collect sample files from the public sources or manually construct them. AFL prefers seeds with high code coverage, a small file size, and low execution time. To minimize the seed set, AFL provides a tool called `afl-cmin` which one can use to remove useless seed files. However, if it is not possible to collect a sophisticated seed set one can always employ an empty file as the only initial seed file.

3 Statistical Evaluations

As the main purpose of fuzzers is to find bugs, the naive approach to compare two or more fuzzers, is to fuzz a target program for a fixed amount of time and then either compare the time it took to find bugs or compare the number of bugs a fuzzer discovered. However, the fuzzing process itself is non-deterministic. For instance, in AFL, the non-deterministic component is the havoc stage which is part of the mutation module. Thus, executing multiple trials with the same fuzzer may yield different results. As a consequence, using only a single execution might lead to a false conclusion. Other utilized evaluation metrics such as the average and median can be affected by similar issues. The common problem of these simple techniques is that they ignore randomness, i.e., they do not consider the non-deterministic nature of fuzzing. The most common method to address this problem is to calculate the statistical significance, i.e., the p-value which was popularized by Fisher [17]. If the p-value is below a predefined threshold we assume that the observed difference between to fuzzers is genuine and consider the results statistically significant.

When comparing two fuzzers, it is not only relevant to know whether the performance differences are statistically significant but also to properly quantify the difference, namely, we have to calculate the effect size. However, when comparing fuzzers on multiple targets non-standardized effect sizes are affected by the unit of measurement which may result in unwanted biases. To address this issue a standardized effect size should be used [2].

In general, we can differentiate between statistical tests for dichotomous and interval-scale results which require a different set of statistical evaluation methods. In the following, we describe both result types and the recommended approach to calculate statistical significance and the corresponding effect size as discussed by Arcuri et al. [2]. For more details about the employed statistical methods, we refer the interested reader to the relevant literature [16,30,38].

An interval-scale result in the context of fuzzing is the time a fuzzer needs to detect a specific bug. In such a case it is recommended to use the Mann-Whitney-U test to calculate the p-value to test for statistical significance. Contrary to the popular *t-test* the Mann-Whitney-U test does not assume that the underlying data follows the normal distribution. To quantify the effect size for interval-scale results, one can utilize the Vargha and Delaneys $\hat{A}_{12}$ statistic.

A dichotomous result can only have two outcomes, usually *success* or *failure*. In the context of a fuzzer evaluation, a dichotomous result simply states whether a specific bug has been discovered in the given time limit. To calculate the statistical significance, Arcuri et al. [2] recommend using the Fisher exact test. As the name suggests, this statistical test is exact which means that it is precise and not just an estimation for the actual p-value. To calculate the effect size for dichotomous results, it is recommended to calculate the odds ratio.

4 Problem Description and Related Work

The evaluation of fuzzers was first analyzed by Klees et al. [26] who demonstrate that simply comparing the number of crashes found using a single trial on a small set of targets is misleading as it gives no insight into whether the fuzzer finding more crashes discovers more bugs in practice. Thus, it is preferred to use a test set with a ground truth, i.e., a set of inputs that trigger a known bug or vulnerability inside the test program. To improve fuzzer evaluations, Klees et al. [26] provided a set of recommendations for evaluating fuzzers based on best practices from the field of software engineering. They recommend 30 trials, a run-time of 24 h and use of the Mann-Whitney-U test for statistical significance, and the $\hat{A}_{12}$ statistic as an effect size. However, as we show in Table 1, these recommendations are only partially followed by current fuzzing evaluations. As it is unknown how much influence each evaluation parameter has on the results, it is unclear whether or not these results are reproducible in practice. Contrary to Klees et al. [26], we conduct comprehensive experiments to be able to argue about the influence of different evaluation parameters based on empirical data.

To discuss the state of current fuzzer evaluations we analyze the evaluations from previous work published in reputable security conferences. The experiments gathered from the evaluation sections of different studies based on the following criteria: ① the experiment is used to compare the overall performance of the respective approach to at least one different fuzzers ② we exclude experiments that are used to either motivate the work or certain design choices as well as case studies. The results are summarized in Table 1. Note that we use the term *Crashes* as an evaluation metric for all evaluations that do not utilize a ground truth and rely on a de-duplication method which tries to correlate crashes to a root cause. However, de-duplication methods are prone to errors and cannot sufficiently estimate the correct number of bugs [26]. We use the term *Bugs* when the authors evaluate fuzzers with a set of targets that contain known vulnerabilities, i.e., it is possible to determine which inputs trigger which bug without utilizing a de-duplication technique.

We observe that none of the fuzzing proposals strictly follows all best practices in their evaluations. For instance, none of the listed studies uses 30 trials per experiment and only a single study employs a standardized effect size. Another problem is the lack of uniformity. This is especially prevalent when real-world programs are used to evaluate fuzzers which regularly use different sets of programs or program versions which may introduce unwanted biases and also makes it hard to compare these results. Furthermore, most studies either do not provide any statistical significance results or only for some of the conducted experiments.

A work that partially addresses similar issues has been introduced by Metzman et al. [31] from Google who published FuzzBench, an evaluation framework for fuzzers. FuzzBench generates a report based on coverage as an evaluation metric including a statistical evaluation. However, as the main purpose of a fuzzer is to find bugs, the coverage is only a proxy metric and therefore less meaningful than comparing the number of bugs found on a ground truth test set and thus not recommended [26,34,37].

Table 1. Analysis of current fuzzer evaluations. Entries with a question mark mean that we were unable to find the respective information in the related study. Test set: *RW* = real-world programs, *Google* = Google fuzzing suite. The number following the test sets corresponds to the number of targets used. Effect Size: *Avg.* = average, *Max.* = maximum value of all trials. Statistical significance: *CI* = confidence intervals, *MWU* = Mann-Whitney-U test.

Fuzzer	No	Test set	Run-time	Trials	Eval. metric	Effect size	Stat. significance
Hawkeye [10]	1	RW (19)	8 h	20	Bugs	Average	–
	2	RW (1)	4 h	8	Bugs	Average, $\hat{A}_{12}$	–
	3	RW (1)	4 h	8	Bugs	Average, $\hat{A}_{12}$	–
	4	Google (3)	4 h	8	Coverage	Average, $\hat{A}_{12}$	–
Intriguer [13]	1	LAVA-M (3)	5 h	20	Bugs	Median, Max	
	2	LAVA-M (1)	24 h	20	Bugs	Median	CI, MWU
	3	RW (7)	24 h	20	Coverage	Median	CI[b], MWU
DigFuzz [42]	1	CGC (126)	12 h	3	Coverage	Norm. Bitmap[a]	-
	2	CGC (126)	12 h	3	Bugs	–	–
	3	LAVA-M (4)	5 h	3	Bugs	?	–
	4	LAVA-M (4)	5 h	3	Coverage	Norm. Bitmap[a]	–
REDQUEEN [3]	1	LAVA-M (4)	5 h	5	Bugs	Median	CI[c]
	2	CGC (54)	6 h	?	Bugs	–	–
	3	RW (8)	10 h	5	Coverage	Median	CI, MWU
	4	RW (8)	10 h	5	Bugs	–	–
GRIMOIRE [5]	1	RW (8)	48 h	12	Coverage	Median	CI, MWU[d]
	2	RW (4)	48 h	12	Coverage	Median	CI, MWU
	3	RW (3)	48 h	12	Coverage	Median	CI, MWU
	4	RW (5)	?	?	Bugs	–	–
EcoFuzz [40]	1	RW (14)	24 h	5	Coverage	Average	p-value[e]
	2	RW (2)	24 h	5	Coverage	Average	–
	3	RW (2)	24 h	?	Crashes	–	–
	4	LAVA-M (4)	5 h	5	Bugs	–	–
GREYONE [18]	1	RW (19)	60 h	5	Crashes[f]	–	–
	2	RW (19)	60 h	5	Coverage	–	–
	3	LAVA-M (4)	24 h	5	Bugs	Average	-
	4	LAVA-M (4)	24 h	5	Crashes	Average	–
	5	RW (10)	60 h	5	Coverage	Average	-
Pangolin [24]	1	LAVA-M (4)	24 h	10	Bug	Average	MWU
	2	RW (9)	24 h	10	Crashes	–	–
	2	RW (9)	24 h	10	Coverage	Average	MWU

[a]Normalized Bitmap size describes the relative size of the bitmap compared to the bitmap found by all tested fuzzers.
[b]Confidence intervals only given for five of the seven targets.
[c]Confidence intervals are only provided for Redqueen.
[d]The Appendix further provides: mean, standard deviation, skeweness, and kurtosis.
[e]We were unable to determine the exact statistical test which has been used to obtain the p-value.
[f]The evaluation compares de-duplicated crashes as well as unique crashes as reported by AFL-style fuzzers.

UNIFUZZ is a platform to compare different fuzzers based on 20 real-world programs [28]. The evaluation metrics are based on crashes which are de-duplicated using the last three stack frames which is known to be unreliable because stack frames might be identical even though they trigger different bugs or stack frame may be different while triggering the same bug [26]. UNIFUZZ provides an overview of the fuzzer performance on each test program which

makes it hard to assess the overall performance. SENF goes one step further and summarizes the results in a single ranking which allows us to easily compare all tested fuzzers. Therefore, it is not required to manually analyze the results on each target separately. However, if needed one can still get the target specific data from the results database of SENF.

5 Our Methodology

In this section, we provide an overview of the most important aspects of a fuzzer evaluation. We describe our choice of fuzzers, seeds, test set, and test machine setup which we use to test our framework to quantify the influence of various evaluation parameters.

5.1 Comparing Fuzzers

Comparing fuzzers with each other is not straightforward due to the various fuzzer designs and the wide variety of available testing methods. A fuzzer design is usually highly complex and given that a fuzzer executes millions of test runs, even small differences can have a huge impact on the performance. Some fuzzers are based on completely novel designs which makes it hard to attribute performance improvements to a specific change. For example, Chen and Chen proposed Angora [11] a mutation-based fuzzer that is written in Rust instead of C/C++ like AFL. Angora implements various methods to improve the fuzzing process: byte-level taint tracking, a numeric approximation based gradient descent, input length exploration, and integration of call stacks to improve coverage mapping. Due to the considerable differences to other fuzzers, it is impossible to accurately quantify the respective contribution of each technique when comparing it with AFL or other fuzzer which do not share the same code base. As a consequence, it is important to evaluate fuzzers based on common ground. Given the high popularity of AFL, we opted to focus on fuzzers that are based on the AFL code base. Note however that our evaluation framework is not specifically tailored to AFL in any way. Thus, it can be used to evaluate an arbitrary selection of fuzzers.

5.2 Test Set Selection

A crucial aspect of any fuzzer evaluation is the underlying test set, i.e., the target programs for which the fuzzer aims to discover bugs. In what follows, we study four different test sets available at the time of testing and argue why we decided to use the CGC test set for our evaluation. Note that we focus on test sets that provide a form of ground truth as there is currently no way to reliably match crashes to the same root cause as proper crash de-duplication is still an open problem (see Sect. 4).

LAVA-M. In 2016, Brendan et al. presented LAVA [14], a method to inject artificial bugs into arbitrary programs. The corresponding *LAVA-M* test set was

the first ground truth test set to evaluate fuzzers that has been published in academia. It consists of four different programs with hundreds of injected bugs. Each bug has a specific bug-id that is printed before deliberately crashing the program. Due to its rather small size, the *LAVA-M* test set lacks the diversity found in real-world programs. Further, recent fuzzers such as Redqueen [3] and Angora [11] solve the test set by finding all injected bugs. This is possible because LAVA-M only features a single bug type which requires that the fuzzer solves magic byte comparisons, missing the bug diversity found in real-world software.

Google Fuzzing Suite. The Google Fuzzer Suite [20] consists of 22 different real-world programs with 25 different challenges that fuzzers are expected to solve. All challenges are documented and may include seed files. However, the test suite is not suitable for our use case as the majority of the bugs can be discovered by existing fuzzers in a very short time span (seconds or minutes). Furthermore, some of the challenges do not contain any bugs. Instead, the goal of these challenges is for the fuzzer to reach a certain path or line of code (i.e., a coverage benchmark) which is not compatible with our evaluation metric as we are interested in the number of bugs found. Additionally, the included bugs are partially collected from other fuzzing campaigns which might introduce biases.

Magma. The Magma fuzzing benchmark is a ground truth test set [22] that is based on a set of real-world programs. At the time of testing, the test set contains six different targets each containing a set of known bugs. Similar to LAVA-M, Magma uses additional instrumentation in the form of bug oracles to signal whether a bug condition has been triggered by a specific input. However, we do not use the Magma test set because at the time of testing it did not provide a sufficiently large test set. Further, not all bugs include a proof-of-vulnerability (PoV) which makes it impossible to know if a fault can be triggered by any means.

CGC. The DARPA Cyber Grand Challenge[1] (CGC) was a capture-the-flag style event where different teams competed by writing tools that are able to detect and subsequently fix bugs in a test corpus of close to 300 different programs with a prize pool of nearly 4 million USD. Each challenge has been designed carefully and consists of one or more binary which mirror functionality known from real-world software. CGC challenges contain at least one bug of one of two types: Type 1 bugs allow an attacker to control the instruction pointer and at least one register. Type 2 bugs allow reading sensitive data such as passwords. The challenges are written by different teams of programmers and do not rely on automatically injected bugs. As a result, the CGC test set offers great bug diversity which are similar to bugs found in real-world software and is therefore not susceptible to the same limitations as the LAVA-M test set.

Instead of the original binaries which were written for *DECREE*, we use the multi OS variant published by Trail of Bits [21] which allows us to execute the challenge binaries on Linux. All challenges and bugs are very-well documented and contain a PoV and a patched version of the respective challenge program(s).

[1] https://github.com/CyberGrandChallenge/.

Each bug is categorized into their respective CWE classes[2]. Further, challenges include test suits that we can use to ensure that the compiled program works as intended which can be especially helpful when using code instrumentation.

Given the greater bug and program diversity of CGC in combination with its great documentation and comprehensive test suites, we select a subset of the ported version of the CGC test set based on the following criteria: ① All tests (including the PoV) are successfully executed on our test servers. ② The target only contains one vulnerability. ③ The vulnerability is of type 1 as type 2 bugs do not lead to a crash. ④ The challenge consists of only one binary as fuzzers usually do not support to fuzz multiple binaries.

We are using targets with only one vulnerability as this allows us to verify the discovered crashing inputs using differential testing (see Sect. 5.6). This process does not require any additional instrumentation (e.g., bug oracles) which may significantly change the program behavior and lead to non-reproducible bugs [28]. Furthermore, we do not need to correlate crashes to their root cause using de-duplication methods. Our complete test set is composed of 42 targets including bugs of 21 different CWE types. We provide a complete list of all targets including their bug types in our public repository [32]. Note that it is not required to use CGC to be able to use SENF because the framework is not specifically tailored to the test set but can used with any test set. Furthermore, SENF can also be used to evaluate fuzzers based on the code coverage, e.g., when testing how long it takes a fuzzer to reach a certain basic block. We provide further details in the public repository [32].

5.3 Seed Sets

To evaluate fuzzers, we opted to use two sets of seed files. The first set of seed files contains sample input which we extract from the test inputs that are shipped with each CGC challenge. We minimize each seed set using `afl-cmin`. As it might not always be possible for users to create a comprehensive seed set for their target, we use an empty file as a second seed set.

5.4 Statistical Evaluation

To evaluate the results of our experiments, we employ the statistical methods described in Sect. 3. SENF supports both, dichotomous and interval-scale statistics as their usage depends on the use case. Dichotomous results provide an answer to the question which fuzzer finds the most bugs in a certain time frame, but ignores the time it took to find a bug. These types of evaluations are relevant for use cases such as OSS-Fuzz [1] where fuzzing campaigns are continuously run for months without a fixed time frame. Statistical tests on interval-scale results are useful in practical deployments when the amount of time to fuzz a target is

[2] Common Weakness Enumeration (CWE) is a list of software and hardware problem types (https://cwe.mitre.org/).

limited, e.g., when running tests before releasing a new software version. We use R [35] to calculate statistical significance tests as well as effect sizes.

When comparing multiple fuzzers or fuzzer configurations on a large set of targets, we encounter two problems. First, due to the large number of comparisons, it is not practical to publish all p-values and effect sizes as part of a study. Secondly, even if one publishes all values, it is not trivial to assess if a fuzzer actually outperforms another. Therefore, we implement a scoring system, which is inspired by Arcuri et al. [2], to summarize the results in a single score. The scoring system follows the intuition that the fuzzer which finds the most bugs the fastest, on the most of the targets is overall the best fuzzer. To determine the best fuzzer, the algorithm compares all fuzzers using the statistical significance tests and standardized effect sizes. For each target, it generates a ranking based on the time it took each fuzzer to find a specific bug. The final score is the average ranking of each fuzzer over the whole test set. For a more detailed description of the scoring algorithm we refer the interested reader to the respective publication [2].

5.5 Fuzzing Evaluation Setup

In Fig. 1 we provide an overview of our fuzzing evaluation setup. At its core, our design features a management server that runs a controller which provides the target program and seed set to one of the available experiment servers. On each experiment server, a dedicated executor starts the fuzzer and monitors the fuzzing process. The monitoring includes logging the CPU utilization and number of executions of the fuzzer. Thus, we can detect hangs and crashes of the fuzzer itself and restart a run if necessary.

After the pre-defined run-time, the executor stops the fuzzer and sends a success message to the controller program. Using the data from all successful fuzzing runs, SENF evaluates the reported results using evaluation methods which compare all executed runs of an arbitrary number of fuzzers and calculates statistical significance, effect size, the ranking based on dichotomous and interval-scale statistical tests.

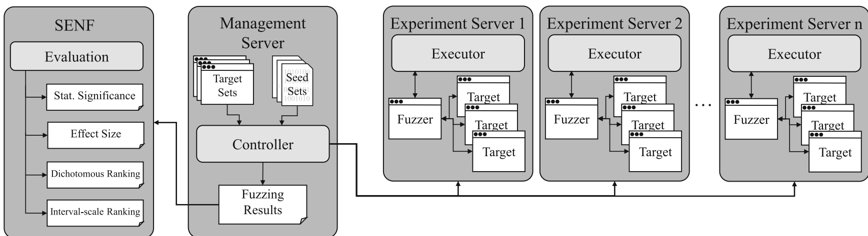


Fig. 1. Conceptual overview of the fuzzing evaluation setup.

5.6 Test Runs

Note that we conduct each experiment (i.e., a combination of fuzzers, targets, and seeds) for a maximum of 24 h. As each target only contains a single bug, we stop fuzzing when an input has been found by a fuzzer that triggers the bug of the respective target. To avoid false positives [3], we verify each crash using differential analysis, i.e., we execute a potential crashing input on the vulnerable and patched version of the respective binary and check whether the input crashes the binary.

SENF itself only requires a database which contains the result data, i.e., the time it took until a specific bus has been triggered, and a list of targets/seeds that should be used in the evaluation. Therefore, our framework can be used to evaluate other fuzzers or test sets. With minimal modifications one can also use other evaluation metrics (e.g., block coverage) to compare fuzzers with SENF.

6 Experiments

We run extensive fuzzing campaigns to systematically quantify the influence of various parameters used in fuzzing evaluations while following state-of-the-art statistical evaluation methodology. We test the influence of the following parameters: the seed set, number of trials, run-time, and number of targets as well as bugs. In total we run 616 fuzzing experiments with an accumulated run-time of over 284k CPU hours.

If not stated otherwise we use the following parameters as a default configuration for the statistical evaluation: a p threshold of 0.05, a non-empty seed set, interval-scale statistical tests, with 30 trials per experiment and a run-time of 24 h. Further, experiments for a specific target are always run on the same hardware configuration to ensure uniform test conditions. Note that when testing with an empty seed we have to exclude seven targets of our test set of 42 programs as these targets do not properly process an empty file thus fail initial tests done in AFLs initialization routine.

We execute all experiments on a cluster of 13 servers. To ensure equal conditions for all fuzzers, we use Docker containers and assign them one CPU core each and a ramdisk to minimize the overhead caused by I/O operations. We utilize Ubuntu 18.04 LTS as an operating system. If not stated otherwise we use fuzzers AFL/Fairfuzz in version 2.52b and AFLFast in version 2.51b as well as AFL++ in version 2.65c. The CGC test set was built using the code from commit `e50a030` from the respective repository from Trail of Bits.

Due to the extensive amount of data in our evaluation and the inherent space limitations, we cannot publish all results (e.g., comparing different p-value thresholds) as part of this publications. We provide an extended version of our work in our public repository [32].

6.1 Fuzzers

We test a total of four fuzzers (❶ AFL [41], ❷ AFLFast [8], ❸ Fairfuzz [27], ❹ AFL++ [15]), two AFL-based compiler optimizations (❺ AFL-CF, ❻ AFL-

LAF), and two modes of AFL (❼ `-d` and ❽ `-q`) which provide a wide range of different performances. In the following, we explain the different fuzzers and modes of AFL we tested including the different compiler optimizations.

❶ **AFL.** The general purpose fuzzer AFL supports various different optimizations and parameters which change one or more its core components: ❼ **AFL (-d).** If the `-d` flag is enabled, AFL skips the deterministic part of the mutation stage and directly proceeds with the havoc stage. ❽ **AFL (-q)** The `-q` flag enables the *qemu mode*. Using this mode, AFL can fuzz a target without access to its source code. The necessary coverage information is collected using QEMU. According to the AFL documentation[3], the performance may decrease substantially due to the overhead introduced by the binary instrumentation.

❺ **AFL-CF.** As described in Sect. 2, AFL ships with various compilers that add the coverage feedback instrumentation when compiling a target program from source code. Using the alternative compiler `afl-clang-fast`, the instrumentation is added on the compiler level, instead of the assembly level, using a LLVM pass which improves the performance.

❻ **AFL-LF.** Based on `afl-clang-fast`, one can try to improve the code coverage by using the LAF LLVM passes[4]. For instance, these passes split multi-byte comparisons into smaller ones which AFL can solve consecutively.

❷ **AFLFast.** AFLFast [8] is a fork of AFL that investigates fuzzing *low-frequency paths*. These are paths that are reached by only a few inputs following the intuition that these inputs solve a path constraint that may lead to a bug. The implementation is part of the power schedule with an optimized search strategy. Note that AFL incorporated improvements from AFLFast starting with version 2.31b.

❸ **Fairfuzz.** Fairfuzz [27] is also based on AFL. Similar to AFLFast, it aims at triggering branches that are rarely reached by other testing inputs. However, it does not utilize a Markov chain model but rather relies on a dynamic cutoff value (i.e., a threshold for the number of hits) to decide which branches are considered *rare*. Further, Fairfuzz uses a heuristic that checks if certain bytes can be modified while still executing the same respective rare branch. Fairfuzz implements these changes as part of the search strategy and the mutation stage of AFL.

❹ **AFL++.** The AFL++ fuzzer [15] is a novel variation of AFL that improves usability and enables easy customization. It implements various improvements from academia as well as the fuzzing community (e.g., the AFLFast power schedules and the LAF LLVM passes). The goal is to introduce a new baseline fuzzer that is used for future fuzzing evaluations.

6.2 Seed Set

First, we evaluate the influence of the seed set by comparing an empty with a non-empty seed set (see Sect. 5.3). The results are depicted in Table 2 which

[3] https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu.
[4] https://gitlab.com/laf-intel/laf-llvm-pass/tree/master.

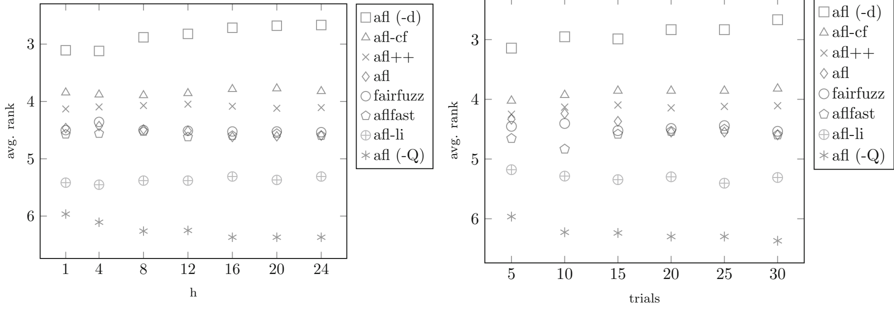
lists the number of times that a fuzzer performed statistically better with the empty and non-empty seed set. We find that with the majority of targets the non-empty seed set either outperforms the empty seed or performs equally well on both statistical tests. We find that AFL is able to detect five bugs using the empty seed set that AFL is unable to detect when using the non-empty seed set. We believe that the main reason for this is that AFL spends less time in the deterministic stage when using an empty seed as the file is only a single byte. Note that even though the performance with a proper seed set is significantly better, testing an empty seed is still useful in cases where it is favorable to minimize the number of variables which may influence the fuzzing process [26] as well as scenarios where one cannot compile a comprehensive sets of inputs for the tested programs.

Table 2. Comparison of the non-empty and empty seed sets using interval-scale and dichotomous tests. Listed are the number of times the performance of the non-empty seed set was statistically better than the empty seed set and vice versa.

	Interval-scaled		Dichotomous	
Fuzzer	Non-empty	Empty	Non-empty	Empty
afl	12	6	6	2
afl (-Q)	8	4	7	1
afl (-d)	18	2	8	1
fairfuzz	13	4	7	1
afl-li	13	6	5	3
afl-cf	12	6	5	2
aflfast	12	5	6	0
afl++	12	5	5	2

6.3 Run-Time

To show the impact of differences in run-time, we calculate the ranking for maximum run-times between 1 h and 24 h. For each ranking, we only consider crashes that have been found in the respective time frame. We present the results in Fig. 2a. We observe that the run-time has a significant influence on the results. Interestingly, we find that even though all fuzzers are based on the same code base there is no uniform trend when increasing the run-time. For example, AFL without its deterministic stage consistently improves, in total by 0.45 in the average ranking from 1 h to 24 h. In the same time the performance of Fairfuzz, AFLFast, and AFL may increase or decrease slightly which also changes the relative ranking of these fuzzers depending on the maximum run-time. *We observe that on our test set, the run-time should be at least* 8 h *as lower run-times may lead to false conclusions of the fuzzer performance.*



(a) Run-times varying between 1h and 24h.

(b) Trials varying between 5 and 30.

Fig. 2. Average ranking when using different run-times and number of trials.

6.4 Number of Trials

To calculate a p-value, one has to repeat every experiment multiple times. The number of trials also influences the minimum p-value that can be achieved. We compare the average ranking of each fuzzer and configuration considering between the first 5 and all 30 trials. In Fig. 2b, we can see that the performance may vary significantly depending on the number of trials used. For example, using 10 trials AFL++ has a slightly better performance than AFL and Fairfuzz, both of which clearly outperform AFLFast. Analyzing the results after 30 trials we find that AFL++ now outperforms AFL and Fairfuzz which both perform as well as AFLFast. *We conclude that the number of trials has significant impact on the results and if under serious resource constraints one should prioritize a higher number of trials over longer run-times.*

6.5 Number of Bugs/Targets

Another parameter that one can adjust is the number of targets a fuzzer is evaluated on. As we use targets with a single bug, the number of targets is equal to the number of bugs in our test set. We evaluate all fuzzers on test sets between five and 35 different targets. For each test set size, we randomly sample 1000 different target combinations and calculate the ranking including maximum and minimum. Note that given larger test sets, the spread will naturally decrease as we sample from a maximum of 42 different targets. In Fig. 3, we can see that the performance may vary substantially depending on the used test set. We further analyze the results and find randomly sampled test sets with 15 targets where AFL-CF outperforms AFL without the deterministic stage or test sets where the performance of Fairfuzz is second to last. Even when we use 35 targets, we find randomly sampled test sets that result in a substantially different rankings compared to the 42 target test set. For example, we observe test sets where AFL++ outperforms AFL-CF or test sets where Fairfuzz performs better than AFL++. *Our results show that target and bug selection should not be taken lightly as it can introduce significant biases when testing fuzzers.*

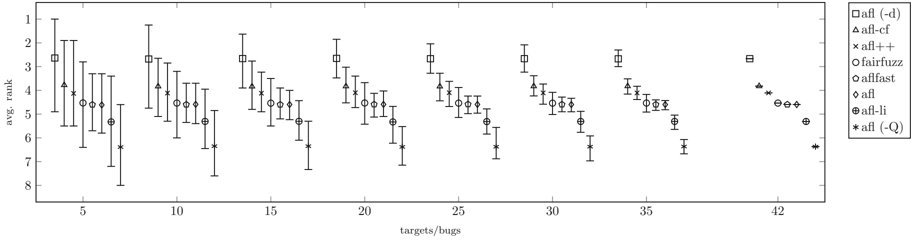


Fig. 3. Average ranking when using varying numbers of targets/bugs. Whiskers correlate to the minimum and maximum rank.

6.6 Further Insights

Next, we compare the SENF-ranking with a ranking that utilizes the *average* as commonly found in fuzzer evaluations (see Sect. 4). The results are shown in Table 3. Notably, when we only consider the average, the overall ranking changes drastically with the exception of the best and worst performing fuzzers. *This shows the influence of statistically insignificant results on the overall performance results which further confirms the choice of using righteous statistical methods as employed in SENF.*

Table 3. Comparison of the SENF-ranking and avg. number of bugs found over all targets.

SENF ranking		Ranking based on Avg.	
Fuzzer	Avg. ranking	Fuzzer	Avg. #bugs found
afl (-d)	2.67	afl (-d)	16.86
afl-cf	3.82	fairfuzz	14.17
afl++	4.11	afl-cf	13.55
fairfuzz	4.54	aflfast	13.26
aflfast	4.60	afl	12.60
afl	4.60	afl-li	12.60
afl-li	5.31	afl++	12.48
afl (-Q)	6.37	afl (-Q)	9.21

To test the *consistency* of each fuzzer, we take a closer look at the time it takes a fuzzer to detect a bug in Fig. 4. To improve the readability of the figure we plot the difference between the shortest and longest time a fuzzer needs to find a bug over all trials for each target. If a fuzzer is not able to find a bug, we set the execution time to 24h. When a fuzzer was not able to find a bug in a target over all trials, we omitted the result to increase readability. For all fuzzers and configurations, randomness plays a significant role when searching for bugs

with differences between minimum and maximum time close to our run-time of 24 h. *No fuzzer in our evaluation is able to consistently find bugs over all trials.*

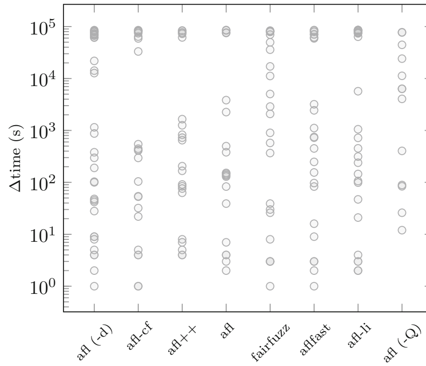


Fig. 4. Difference between min. and max. exec. time for each fuzzer and target over all trials.

7 Discussion

Test Set Selection. Our framework SENF in combination with a ground truth test set significantly increases the probability that the reported results are reproducible. Even though our test set of 42 different programs and 21 different bug types ensures a certain level of diversity in our evaluation, the resulting ranking could potentially differ if a larger, representative test set of real-world programs with a ground truth is used because programs from the CGC test set do not provide the same level complexity. Note that other test sets can easily be evaluated with SENF as we only require a database containing the experiment results as input.

Resource Limitations. Due to unavoidable limitations of resources, we cannot analyze the full range of parameters used in existing fuzzing evaluations, e.g., run-times of 60 h (see Sect. 4). Therefore, we limit our experiments to values recommended in fuzzing literature [26]. For the same reason, we do not conduct experiments with multiple concurrent fuzzer instances testing the same target. The experiments of Chen et al. [12] as well as Böhme and Falk [6] suggest that the performance of fuzzers varies significantly when fuzzing with multiple instances simultaneously.

Fuzzer Selection. Due to the aforementioned resource constraints, we have to limit the selection of fuzzers as the experiments in Sect. 6 already required over 280k CPU hours. We opted to focus on AFL, AFL-based fuzzers, and various optimizations as this allows us to easily attribute performance differences. Furthermore, AFL is the most popular baseline fuzzer, e.g., it is

recommended by Klees et al. [26] and used in all evaluations we studied in Sect. 4. Additionally, AFL is commonly used as a code base to implement new fuzzers [8,19,27,29,33,36]. For these reasons, we argue that focusing on AFL-style fuzzers is more significant for common fuzzer evaluations compared to other fuzzers. However, since our implementation is open-source one can easily use SENF to evaluate any set of fuzzers. We provide detailed guidelines in our public repository [32].

Scoring Algorithm. The scoring algorithm we use in our evaluation adopts the commonly used intuition that the fuzzer which outperforms the other fuzzers (i.e., finds more bugs) on the most targets has the best overall performance. However, other evaluation metrics may be useful for other use cases, e.g., when testing a target with a section of different fuzzers one may not only be interested in the fuzzer that finds the most bugs but also fuzzers that find a unique set bugs which all other fuzzers are unable to detect. However, calculating the unique set of bugs for each fuzzer does not require complex statistical evaluations as provided by SENF.

Furthermore, our evaluation does not take into account by how much a fuzzer A improves over a different fuzzer B. SENF addresses this problem by supporting a variable effect size thresholds. Thus, interested parties can set a custom minimum effect size which SENF takes into account when calculating the score of a fuzzer. We provide more detailed information on the effect size and its influence in the extended version of this paper.

Threshold of the p-value. In our evaluation, we opted to use the widely established p threshold of 0.05 which is commonly used in software engineering evaluations [26]. However, this threshold is generally considered a trade-off between the probability of false positive and false negative results. Other scientific communities opted to use lower thresholds or other methods of statistical evaluation [4]. SENF addresses this and lets the user set an arbitrary threshold to calculate the average ranking of each fuzzer.

8 Conclusion

Our analysis of recent fuzzing studies shows that fuzzers are largely evaluated with various different evaluation parameters which are not in line with the recommendations found in academic literature. To address these issues, we presented SENF, which implements dichotomous and interval-scale statistical methods to calculate the p-value and effect sizes to compute a ranking to asses the overall performance of all tested fuzzers.

Based on extensive empirical data, we quantified the influence of different evaluation parameters on fuzzing evaluations for the first time. We demonstrate that even when we utilize the recommended statistical tests, using insufficient evaluation parameters—such as a low number of trials or a small test set—may still lead to misleading results that in turn may lead to false conclusions about the performance of a fuzzer. Thus, the choice of parameters for fuzzing evaluations

should not be taken lightly and existing recommendations should be followed to lower the chance of non-reproducible results. We described and open-sourced a practical evaluation setup that can be used to test the performance of fuzzers.

Acknowledgements. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297.

References

1. Aizatsky, M., Serebryany, K., Chang, O., Arya, A., Whittaker, M.: Announcing OSS-Fuzz: continuous fuzzing for open source software (2016)
2. Arcuri, A., Briand, L.: A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**, 219–250 (2014)
3. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: REDQUEEN: fuzzing with input-to-state correspondence. In: Symposium on Network and Distributed System Security (NDSS) (2019)
4. Benjamin, D.J., Berger, J.O., Johannesson, M., Nosek, B.A., Wagenmakers, E.J., et al.: Redefine statistical significance. Hum. Nat. Behav. **2**, 6–10 (2017)
5. Blazytko, T., et al.: GRIMOIRE: synthesizing structure while fuzzing. In: USENIX Security Symposium (2019)
6. Böhme, M., Falk, B.: Fuzzing: on the exponential cost of vulnerability discovery. In: Symposium on the Foundations of Software Engineering (FSE) (2020)
7. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: ACM Conference on Computer and Communications Security (CCS) (2017)
8. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: ACM Conference on Computer and Communications Security (CCS) (2016)
9. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Conference on Operating Systems Design and Implementation (2008)
10. Chen, H., et al.: Hawkeye: towards a desired directed grey-box fuzzer. In: ACM Conference on Computer and Communications Security (CCS) (2018)
11. Chen, P., Chen, H.: Angora: efficient fuzzing by principled search. In: IEEE Symposium on Security and Privacy (S&P) (2018)
12. Chen, Y., et al.: EnFuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. In: USENIX Security Symposium (2019)
13. Cho, M., Kim, S., Kwon, T.: Intriguer: field-level constraint solving for hybrid fuzzing. In: ACM Conference on Computer and Communications Security (CCS) (2019)
14. Dolan-Gavitt, B., et al.: LAVA: large-scale automated vulnerability addition. In: IEEE Symposium on Security and Privacy (S&P) (2016)
15. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++: combining incremental steps of fuzzing research. In: USENIX Workshop on Offensive Technologies (WOOT) (2020)

16. Fisher, R.: On the Interpretation of χ^2 from contingency tables, and the calculation of P. J. R. Stat. Soc. **85**, 87–94 (1922)
17. Fisher, R.: Statistical Methods for Research Workers. Oliver and Boyd, Edinburgh (1925)
18. Gan, S., et al.: GREYONE: data flow sensitive fuzzing. In: USENIX Security Symposium (2020)
19. Gan, S., et al.: CollAFL: path sensitive fuzzing. In: IEEE Symposium on Security and Privacy (S&P) (2018)
20. Google: fuzzer-test-suite (2016). https://github.com/google/fuzzer-test-suite/
21. Guido, D.: Your tool works better than mine? Prove it (2016). https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/
22. Hazimeh, A., Herrera, A., Payer, M.: Magma: a ground-truth fuzzing benchmark. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems 4 (2020)
23. Hocevar, S.: zzuf (2006). https://github.com/samhocevar/zzuf/
24. Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C.: Pangolin: incremental hybrid fuzzing with polyhedral path abstraction. In: IEEE Symposium on Security and Privacy (S&P) (2020)
25. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: International Conference on Software Engineering (ICSE) (2014)
26. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: ACM Conference on Computer and Communications Security (CCS) (2018)
27. Lemieux, C., Sen, K.: FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage (2018)
28. Li, Y., et al.: UNIFUZZ: a holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: USENIX Security Symposium (2021)
29. Lyu, C., et al.: MOPT: optimized mutation scheduling for fuzzers. In: USENIX Security Symposium (2019)
30. Mann, H., Whitney, D.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18** (1947)
31. Metzman, J., Arya, A., Szekeres, L.: FuzzBench: Fuzzer Benchmarking as a Service (2020). https://opensource.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html
32. Paaßen, D., Surminski, S., Rodler, M., Davi, L.: Public github respository of SENF. https://github.com/uni-due-syssec/SENF
33. Pham, V.T., Böhme, M., Santosa, A.E., Căciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. IEEE Trans. Softw. Eng. (2019)
34. Pham, V.-T., Khurana, S., Roy, S., Roychoudhury, A.: Bucketing failing tests via symbolic analysis. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 43–59. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_3
35. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2019).https://www.R-project.org/
36. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kAFL: hardware-assisted feedback fuzzing for OS kernels. In: USENIX Security Symposium (2017)
37. van Tonder, R., Kotheimer, J., Le Goues, C.: Semantic crash bucketing. In: ACM/IEEE International Conference on Automated Software Engineering (2018)
38. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. 25, 10—132 (2000)

39. Wang, Y., et al.: Not all coverage measurements are equal: fuzzing for input prioritization. In: Symposium on Network and Distributed System Security (NDSS) (2020)
40. Yue, T., et al.: EcoFuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: USENIX Security Symposium (2020)
41. Zalewski, M.: Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt
42. Zhao, L., Duan, Y., Yin, H., Xuan, J.: Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Symposium on Network and Distributed System Security (NDSS) (2019)