SoK

Schloegel, Moritz; Bars, Nils; Schiller, Nico; Bernhard, Lukas; Scharnowski, Tobias; Crump, Addison; Ale-Ebrahim, Arash; Bissantz, Nicolai; Muench, Marius; Holz, Thorsten

DOI:
10.1109/SP54263.2024.00137

Citation for published version (Harvard):
Schloegel, M, Bars, N, Schiller, N, Bernhard, L, Scharnowski, T, Crump, A, Ale-Ebrahim, A, Bissantz, N, Muench, M & Holz, T 2024, SoK: Prudent Evaluation Practices for Fuzzing. in *2024 IEEE Symposium on Security and Privacy (SP).* Proceedings of the IEEE Symposium on Security and Privacy, IEEE, Los Alamitos, CA, USA, pp. 137-137, 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, California, United States, 19/05/24. https://doi.org/10.1109/SP54263.2024.00137

SoK: Prudent Evaluation Practices for Fuzzing

Moritz Schloegel[1], Nils Bars[1], Nico Schiller[1], Lukas Bernhard[1], Tobias Scharnowski[1]
Addison Crump[1], Arash Ale-Ebrahim[1], Nicolai Bissantz[2], Marius Muench[3], Thorsten Holz[1]

[1]*CISPA Helmholtz Center for Information Security, {first.lastname}@cispa.de*
[2]*Ruhr University Bochum, nicolai.bissantz@ruhr-uni-bochum.de*
[3]*University of Birmingham, m.muench@bham.ac.uk*

***Abstract*—Fuzzing has proven to be a highly effective approach to uncover software bugs over the past decade. After AFL popularized the groundbreaking concept of lightweight coverage feedback, the field of fuzzing has seen a vast amount of scientific work proposing new techniques, improving methodological aspects of existing strategies, or porting existing methods to new domains. All such work must demonstrate its merit by showing its applicability to a problem, measuring its performance, and often showing its superiority over existing works in a thorough, empirical evaluation. Yet, fuzzing is highly sensitive to its target, environment, and circumstances, e. g., randomness in the testing process. After all, relying on randomness is one of the core principles of fuzzing, governing many aspects of a fuzzer's behavior. Combined with the often highly difficult to control environment, the *reproducibility* of experiments is a crucial concern and requires a prudent evaluation setup. To address these threats to validity, several works, most notably *Evaluating Fuzz Testing* by Klees et al., have outlined how a carefully designed evaluation setup should be implemented, but it remains unknown to what extent their recommendations have been adopted in practice.**

In this work, we systematically analyze the evaluation of 150 fuzzing papers published at the top venues between 2018 and 2023. We study how existing guidelines are implemented and observe potential shortcomings and pitfalls. We find a surprising disregard of the existing guidelines regarding statistical tests and systematic errors in fuzzing evaluations. For example, when investigating reported bugs, we find that the search for vulnerabilities in real-world software leads to authors requesting and receiving CVEs of questionable quality. Extending our literature analysis to the practical domain, we attempt to reproduce claims of eight fuzzing papers. These case studies allow us to assess the practical reproducibility of fuzzing research and identify archetypal pitfalls in the evaluation design. Unfortunately, our reproduced results reveal several deficiencies in the studied papers, and we are unable to fully support and reproduce the respective claims. To help the field of fuzzing move toward a scientifically reproducible evaluation strategy, we propose updated guidelines for conducting a fuzzing evaluation that future work should follow.

1. Introduction

Fuzzing, a portmanteau of "fuzz testing", has gained much attention in recent years, and the method has proven to be highly successful in uncovering many types of faults in software systems. Companies such as Meta, Google, and Oracle have invested significant resources in this technology and use it to test their products. Large software projects such as web browsers or the Linux kernel incorporate fuzzing into their development cycle, and Google is running an extensive and continuous fuzzing campaign for more than $1,200$ open-source projects via OSS-Fuzz [62]. Beyond the wide acceptance in the industry, a large number of academic papers have proposed numerous improvements and novel techniques to enhance fuzzing further. More specifically, we found that, over the past six years, more than 280 papers on fuzzing have been published in the top computer security and software engineering venues.

A cornerstone of fuzzing research, and science in general, is that other researchers can critically assess the correctness of scientific results. To this end, the research results must be *reproducible*, meaning that another group should be able to obtain the same results using the same experimental setup, often by using a research artifact provided by the authors [8]. Reproducibility is paramount for other researchers to understand, trust, and build on the research results.

To enable high-quality research and provide a common foundation for evaluating fuzzing methods, several works describe how newly proposed fuzzing approaches should be evaluated. In 2018, the first and most influential paper describing a reproducible evaluation design was published by Klees et al. [88]. It describes guidelines to advise researchers on how fuzzing research should evaluate their respective contributions. For example, a crucial insight introduced by Klees et al. is the repetition of experiments to account for the inherent randomness of the fuzzing process. Although Klees et al. recommend "a sufficient number of trials" and use 30 trials in their own experiments, we found that in practice, this recommendation is interpreted as anything between three and 20 repetitions. Another guideline is to confirm the fuzzers' performance statistically; however, this makes little sense with few repetitions and is often skipped.

In this work, we systematically review how the recommendations for evaluating fuzzing methods are implemented in practice and critically evaluate the reproducibility of fuzzing research. We propose revised best practices for evaluating fuzzing methods and point out pitfalls that we have observed in practice. In other fields, such work has had a significant impact on improving research from a methodological point of view [1], [4], [46], [155].

We conduct a thorough literature review of 150 fuzzing papers published in prestigious A* venues—as ranked by CORE2023 [128]—between 2018 and 2023. While we primarily focus on computer security venues, namely IEEE Symposium on Security and Privacy (S&P), USENIX Security Symposium (USENIX), ACM Conference on Computer and Communications Security (CCS), and ISOC Network and Distributed System Security (NDSS) Symposium, we also examine three software engineering venues: IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), and International Conference on Software Engineering (ICSE). For all papers, we: (i) systematically analyze how evaluations are conducted (in terms of metrics, targets, baselines, reported bugs, etc.), (ii) check whether common fuzzing guidelines (as outlined by Klees et al. [88] or embodied in implicit community wisdom, e. g., "do not use artificial bug datasets") are followed, and (iii) investigate potential flaws threatening the validity of the respective evaluation.

Following our literature analysis, we present eight case studies of fuzzing papers across different fields and attempt to reproduce (parts of) their evaluation. For each case study, we discuss any shortcomings we have identified because they illustrate potential pitfalls of which researchers should be aware. Note that these case studies are *not* intended to point fingers or criticize any particular work. Instead, we aim to highlight potential challenges that can affect the outcome of a research paper and explore what aspects need to be considered when designing the evaluation of a fuzzing method. Based on the findings of our literature review and case studies, we propose best practices for evaluating future fuzzing methods to enable reproducible research.

In summary, we make the following key contributions:

- We conduct a systematic literature survey of 150 papers published in the past six years at top venues to assess how fuzzing methods are typically evaluated.
- We attempt to reproduce eight papers to assess the practical aspect of fuzzing evaluations. In doing so, we identify several obstacles that illustrate (sometimes subtle) shortcomings of evaluating fuzzing methods.
- Based on our lessons learned, we provide revised recommendations and best practices for future fuzzing evaluations.

Supplementary material for this work is available online at https://github.com/fuzz-evaluator/, including our reproduction artifacts and recommended best practices for future work (see https://github.com/fuzz-evaluator/guidelines).

2. Fuzzing Evaluation Guidelines

We first provide a brief overview of fuzzing before describing several generally accepted best practices that guide a typical fuzzing evaluation.

2.1. Background on Fuzzing

Fuzzing, also referred to as *fuzz testing*, is a dynamic testing technique with the goal of uncovering bugs in systems. This typically happens by mutating some initial input(s) to the system or by deriving inputs from input specifications such as grammars. While processing the provided input, the system under test is monitored for *interesting* behavior. Beyond easily observable faults, such as program crashes, fuzzers can use more sophisticated bug oracles, such as sanitizers or differential testing. Moreover, modern fuzzers often use lightweight instrumentation to receive coverage feedback, allowing them to track inputs that executed previously unseen edges. A comprehensive overview of various fuzzing techniques can be found in the *Fuzzing Book* [178], and several surveys present a comprehensive overview of this topic [112], [193] or open challenges in this domain [14]. Most fuzzing research proposes an improvement by way of new techniques, new components, or entirely new fuzzers—few works focus on the theory behind fuzzing [20], [21], [23], [107].

A fundamental principle of all fuzzers is the inherent inclusion of randomness into the testing process. Starting from the scheduling order of the process, through the input and the mutations applied to it, to the fuzzing environment (including functions such as `getpid`, `time`, or `rand`, or shared resources such as the filesystem), there are numerous sources of randomness that make deterministic and reproducible execution challenging.

2.2. Guidelines of *Evaluating Fuzz Testing*

The randomized nature of fuzzing needs to be taken into account during the evaluation, which leads to challenges with reproducibility of research results in practice. Hence, the seminal paper by Klees et al. [88] outlined several guidelines on how a proper fuzzing evaluation should be conducted. For a reproducible and fair evaluation, they propose the following recommendations:

Recommendation 1 – Baseline: A comparison with a relevant and reasonable baseline is imperative to show what improvement a particular fuzzer provides.

Recommendation 2 – Targets: A relevant sample of targets to compare against is necessary. This includes benchmark programs with known bugs that can be used as a ground truth to measure bug detection capabilities.

Recommendation 3 – Setup & Parameters: Due to the inherent randomness of fuzzing, individual runs with the same configuration can yield significantly different outcomes. To address this problem, Klees et al. propose repeating the experiment multiple times. Similarly, fuzzing performance may vary within a single run, so short runtimes are not appropriate for extrapolating the behavior of a fuzzer over longer times. They propose 24 hours as a reasonable fuzzer runtime and recommend plotting the performance over time. Seed sets must be well documented and carefully selected; ideally, various sets, including the empty or uninformed seed, are tested.

Recommendation 4 – Evaluation Metrics: Ideally, fuzzing evaluations should not be based on proxy metrics such as code coverage alone, but on a fuzzer's ability to find bugs, i. e., the goal for which it was designed. In particular, an evaluation should not rely on heuristics such as AFL's coverage profiles or stack hashing. Complementing the evaluation on bug detection, Klees et al. recommend code coverage in terms of basic blocks or edges as secondary metric.

Recommendation 5 – Statistical Evaluation: Finally, the fuzzing evaluation should undergo statistical evaluation to rule out that the observed behavior is by mere chance. This requires a *sufficient* number of trials (Klees et al. themselves use 30); then, a statistical test such as the Mann-Whitney U-test or bootstrap-based methods should be used to test the null hypothesis that the new method exhibits no difference compared to a reasonable baseline.

2.3. Guidelines of *FuzzBench*

FuzzBench [118], a benchmarking suite for general-purpose fuzzer evaluation developed by Google, provides several target programs and aims to provide a standardized setup for fair comparison of fuzzers. FuzzBench is the successor to the Google Fuzzer Test Suite (FTS) [63]. During their extensive evaluation, the authors made two key observations that can serve as a guideline for future fuzzing research. First, the performance of a fuzzer varies significantly depending on the number of initial seeds; running without seeds allows for studying the difference when only a particular fuzzer can solve some comparisons/branches. Second, using a saturated corpus for fuzzing is *not* recommended, as fuzzers are barely capable of augmenting it. Even though this is common in practice, it is not well suited to discern or measure the performance of fuzzers.

2.4. Guidelines of *On the Reliability of Coverage*

More recently, Böhme et al. [23] made a number of recommendations based on their evaluation of the reliability of coverage. In particular, they recommend to use at least ten representative programs, each tested at least ten times for at least 12 hours (preferably, each value should be doubled). The selected programs should be real-world programs, and a bug evaluation should be done on real-world bugs. In addition to bugs, code coverage should also be evaluated—both using established metrics. In particular, fuzzer-specific measures such as AFL's unique paths should be avoided. For comparison, authors should choose a suitable baseline, such as the fuzzer on top of which the new technique is implemented. Authors should consider splitting benchmarks into a *training* and *validation* set to avoid overfitting. To confirm evaluation results, authors must measure significance and effect size using established techniques. They should discuss threats to the validity of their evaluation and how they mitigated them. Finally, authors should carefully document their setup and publish evaluation artifacts on long-term stable platforms such as Zenodo.

2.5. Fuzzing Benchmarks

Over the years, several *standardized benchmarks* and *platforms* to conduct fair and comparable fuzzing evaluations have been proposed, e. g., Google's Fuzzer-Test-Suite [63] (2016; superseded by FuzzBench), LAVA-M [51] (2016), CGC [45] (2018), Magma [70] (2020), FuzzBench [118] (2020), Unibench [99] (2021), ProFuzzBench [123] (2021), and RevBugBench [183] (2022).

These benchmark platforms aim to measure the performance of general-purpose fuzzing, except for ProFuzzBench, which focuses on stateful protocol fuzzing. Overall, we can distinguish between benchmarks focusing on the comparison of achieved coverage (Google's Fuzzer-Test-Suite, Unibench, FuzzBench, and ProFuzzBench) and those focusing on the bug-finding capabilities of the fuzzing technique (LAVA-M, CGC, Magma, and RevBugBench). In the latter category, some utilize artificial bug injection (LAVA-M and CGC), make efforts to port actual vulnerabilities to the latest version of a program (Magma), or to revert fixes (RevBugBench). Artificial bug injection methods often introduce shallow bugs that are amenable to fuzzers, and are generally no longer recommended for an evaluation [18], [118], [162], [183].

3. Literature Analysis

With these guidelines and benchmarks in mind, we now study their adoption to better understand what best practices are used in fuzzing research. To this end, we perform a comprehensive literature survey of recent fuzzing papers.

3.1. Method

We examine all fuzzing papers published at the top computer security and software engineering conferences between 2018 and 2023[1]. We include a paper in our analysis if its focus is on fuzzing, e. g., it proposes a new method or extensively evaluates existing ones. In contrast, we exclude papers using fuzzers as a means to support their primary focus, e. g., solely to generate some diverse inputs. We identify 289 candidate papers for which we collect metadata about the underlying evaluation method, including whether the paper successfully participated in an artifact evaluation process. We then randomly select 52% (150) from these 289 papers and manually review them, i. e., study the design and evaluation of the work in detail. Table 1 shows an overview of analyzed papers.

[1] For 2023, ASE and FSE have not published the papers at the time of writing. We therefore work with available preprints.

TABLE 1. OVERVIEW OF ANALYZED PAPERS.

Year	Venue	Papers	Studied
2023	ASE*	[159], [76], [105]	3/7
	FSE*	[166]	1/6
	ICSE	[82], [80], [165], [67], [92]	5/11
	CCS	[182], [47], [32], [116]	4/9
	NDSS	[78], [65], [17]	3/4
	S&P	[108], [19], [104]	3/9
	USENIX	[149], [109], [186], [143], [41], [111], [153], [2], [10], [161], [134], [96]	12/29
2022	ASE	[58], [174]	2/4
	FSE	[66], [189]	2/6
	ICSE	[97], [124], [89], [163], [64], [115], [151], [52]	8/17
	CCS	[83], [12], [57], [144], [37], [29], [191]	7/8
	NDSS	[84], [169], [180]	3/6
	S&P	[74], [147], [102], [28], [100]	5/9
	USENIX	[148], [190], [183], [140], [120], [43], [185], [9], [27]	9/19
2021	ASE	[106], [81]	2/6
	FSE	[118], [181]	2/4
	ICSE	[16], [157], [131]	3/6
	CCS	[61], [192], [122], [54], [72], [33]	6/13
	NDSS	[49], [136], [86]	3/6
	S&P	[117], [40]	2/7
	USENIX	[91], [142], [55], [139]	4/13
2020	ASE	[125], [188]	2/4
	FSE	[22], [152], [145]	3/7
	ICSE	[113], [164], [167], [158]	4/6
	CCS	[171]	1/2
	NDSS	[87], [141], [162]	3/4
	S&P	[132], [6], [168], [75], [48], [34]	6/7
	USENIX	[150], [175], [137], [77], [59], [42], [24], [93], [53], [194], [135]	11/19
2019	ASE	–	0/0
	FSE	[98]	1/4
	ICSE	[36], [126], [39], [172], [160]	5/7
	CCS	[38], [31]	2/3
	NDSS	[69], [5], [7]	3/4
	S&P	[121], [170], [146], [173], [79]	5/5
	USENIX	[68], [35], [187], [85], [13], [110]	6/6
2018	ASE	[95]	1/2
	FSE	–	0/0
	ICSE	–	0/0
	CCS	[25]	1/2
	NDSS	[26], [119]	2/2
	S&P	[60], [133]	2/3
	USENIX	[154], [129], [176]	3/3
total #papers analyzed			150/289

* *limited to available preprints*

We investigate whether the fuzzing evaluation guidelines outlined in Section 2 are followed or whether an evaluation deviates from them. We want to stress that there may be good reasons to deviate from these guidelines, making a manual review and judgment on a case-by-case basis mandatory. We also study whether the evaluations performed expose flaws that future fuzzing papers could avoid.

3.2. Results

We study the papers regarding their reproducibility, targets, fuzzers, evaluation setup in terms of resources, common metrics, and statistical evaluation.

3.2.1. Reproducibility. A crucial aspect of verifying and advancing science is the ability to reproduce existing research results. When examining the metadata we collected for all 289 fuzzing papers, we find that 74% (214) publish the code of their technique, while 23% (66) do not share their code. Some do not contribute new code, upstreamed their code, or have not yet released the code (applies to FSE, which will take place after time of writing). Regarding other data (excluding code), we find that 11% (31) share data, 20 of which publish data as a substitute because they do not share their code or have no code to share. All software engineering conferences (ASE, FSE, and ICSE), USENIX Security, and CCS (since 2023) offer an artifact evaluation process where independent reviewers assess the published research artifact (for 2023, ASE and FSE have not yet published this data). Our analysis found that 36% (103) of the papers did not have access to such an artifact evaluation; 37% (107) had access but opted to not participate or failed to receive any badge. Only 23% (66) of the papers have one or more badges. Of these, 64 are considered *available* and 63 *functional* or *reusable*, a crucial requirement for reproduction. USENIX Security and CCS offer to reproduce the results of a paper, which only 16 out of 57 eligible papers achieved. We emphasize that artifact evaluation has been introduced only in recent years, but participation is rising. CCS offered artifact evaluation for the first time in 2023, further supporting this trend.

> With 74%, a majority of works releases their code. Despite being relatively new, 60% of the papers already had access to artifact evaluation, with adoption lagging behind at 23% of papers that obtained a badge.

3.2.2. Targets under Test. To showcase the strengths of an approach, a suitable set of targets is required. Looking at the distribution of used targets (excluding datasets) in Table 2, we find that they are strongly biased towards byte-oriented file formats, especially binutils. On average, fuzzing papers evaluate on 8.9 targets. In summary, we found 753 different targets used across all studied papers; of these, 76% (576) were evaluated in only one paper. In addition to real-world targets, a common way of reproducibly measuring fuzzer performance is using benchmarks. Figure 1 shows how benchmarks have been adopted in the past years. In total, 61% (91) of the papers use no benchmark, 17% (26) use LAVA-M [51], 10% (15) use FuzzBench [118], 8% (12) use Google's Fuzzer Test Suite (FTS) [63], 5% (8) DARPA's CGC binaries (CGC) [45], 4% (6) rely on Magma [70], and 1% (2) build on Unibench [99] for benchmarking purposes. Despite its success, LAVA-M is nowadays considered flawed because it artificially injects vulnerabilities into a given target program that are easy for a fuzzer to find but do not correspond to real bugs [18], [118], [162], [183]. More recent works using LAVA-M often do so only for comparability reasons [78], [82]. Similar to LAVA-M, CGC is widely considered outdated and inadequate.

> Real-world targets are often limited to binary input-affine programs, while benchmarks are not used by the majority of papers. Benchmarks with artificial vulnerabilities are still used.

TABLE 2. TARGETS FUZZED IN FIVE OR MORE ANALYZED PAPERS (EXCLUDING BENCHMARKS). SOME PAPERS REPORT GENERICALLY TO EVALUATE ON BINUTILS, WHILE OTHERS SPECIFY EXACT TARGETS, SUCH THAT NUMBERS IN PRACTICE MAY DIFFER SLIGHTLY.

#Uses	Target
25	objdump, readelf
20	nm, tcpdump
19	libpng
17	libtiff
13	cxxfilt, jhead, libjpeg
12	libxml2
11	nasm
10	jasper, libming, openssl, size
9	file, ImageMagick, mjs, tiff2pdf
8	djpeg, exiv2, JavaScriptCore, libarchive, SQLite, v8, xmllint
7	ChakraCore, ffmpeg, harfbuzz
6	binutils, lcms, lrzip, mupdf, OpenJPEG, SpiderMonkey
5	bento, bsdtar, catdoc, cflow, curl, freetype2, GraphicMagick, json, pcre2, proj4, strip, tiff2ps, yara, zlib

3.2.3. Evaluation against State of the Art. Comparison with a strong set of existing work helps to demonstrate that a new method is particularly suited to solve a specific problem. Yet, only a few techniques published in the past few years have been broadly incorporated in follow-up work. Instead, the most famous fuzzers extended with new techniques are AFL [177] with 30% (45), AFL++ [56] with 6% (9), libFuzzer [101] with 5% (7), and syzkaller [50] with 4% (6). Interestingly, all of these tools are non-academic works; only for AFL++ a peer-reviewed paper has been published [56]. Contrasting this number, 33% (49) of the proposed tools are not based on any existing tool.

When looking at the fuzzers chosen as baselines for comparison, we find that AFL is compared against by 35% (53) of studies, followed by QSym [176] with 15% (23), AFLFast [15] with 14% (21), Angora [30] with 13% (20), FairFuzz [95] with 8% (12), and AFL++ with 9% (14). From the 150 papers we analyzed, only QSym (2018), FairFuzz (2018), and MOpt [110] (2019) have been chosen by more than five follow-up works for comparison. More recently, only Fuzzilli [65] (published 2023, open-sourced early 2019) was used by multiple works for their evaluation, even before the paper was published. This does not account for techniques replicated in AFL++ or LibAFL [57], which reimplement many successful techniques proposed [7], [15], [90], [110]. On average, a fuzzing paper evaluates against 3.2 other fuzzers.

Analyzing whether papers omit comparing against a relevant fuzzer in their evaluation, we find that 20% (30) of the works ignore at least one relevant state-of-the-art method and 3% (4) even omit comparing against their baseline, i. e., the tool on which they base their own fuzzer.

> 45% of fuzzing research builds on top of non-academic fuzzers, 33% build a new tool. 23% percent of fuzzing evaluations fail to compare against relevant state-of-the-art fuzzers or their own baseline.

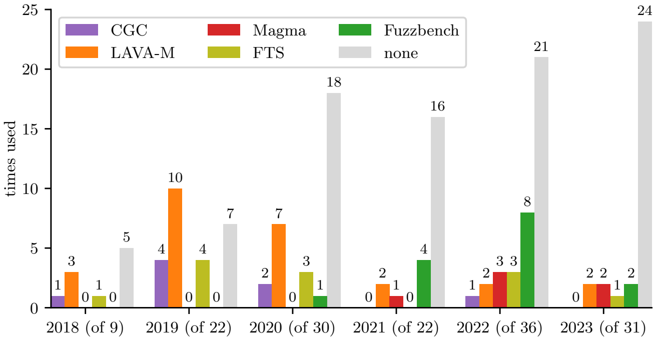

Figure 1. Benchmark usage over the years. The numbers in brackets represent the number of papers analyzed for the respective year. Note that some papers use multiple benchmarks, hence the numbers do not add up.

3.2.4. Evaluation Setup. With respect to the evaluation setup, we analyze the runtime, the number of CPU cores assigned, whether all resources were allocated fairly, and the seeds used for the experiments.

Runtime. Reviewing the experiment setup used across fuzzing evaluations, we find that the majority of papers uses a runtime of 24h, more precisely 56% (84) of the papers run at least one experiment for 24 hours. As Figure 2 outlines, only 27% (40) of the works use a runtime of less than 23 hours, while 29% (44) use an even higher runtime. 5% (8) do not specify their runtime or have no own experiments measuring time.

CPU cores. In terms of CPU cores assigned to fuzzers, we find an inconsistent picture, with a significantly varying number of CPU cores used. The most common result was that 25% (38) of the papers did not specify how many CPU cores they used, 27% (40) used one core, and 8% (12) used two cores.

Fair computing resources. When checking whether the available computing resources were allocated fairly (e. g., the same number of cores were allocated to each fuzzer and they were run for the same amount of time), we find that this is the case for 74% (111) of the works. For 15% (23), we could not infer this information from the description in the paper, and 5% (8) did not evaluate other fuzzers or did not conduct any experiments where this was an issue. Crucially, 5% (8) unfairly allocate resources, giving

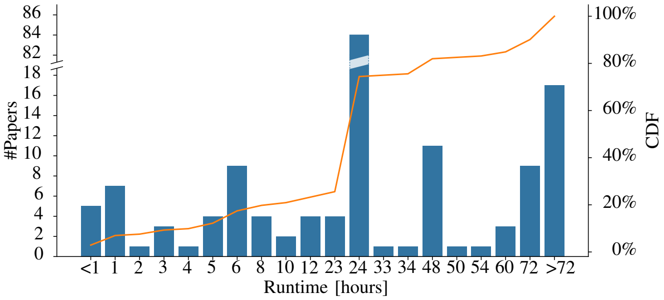

Figure 2. Distribution of runtimes used in practice and cumulative distribution function (CDF), which shows that 27% of papers use a runtime of less than 23 hours. 26 papers use multiple, different runtimes; we include all in these cases.

one fuzzer an advantage over another. For these 8, we found one benign case in which an existing method was given more resources, one case in which the number of executions was fairly distributed rather than the runtime (thereby giving slow fuzzers an advantage), two cases in which a different number of cores was used (in one case, giving the new fuzzer twice the cores than others), and four cases where the new approach was allowed some preprocessing time, e. g., for some static analysis pass or seed preprocessing, before it was then allotted the same time as all other tools, effectively giving it more computation time. Unfortunately, the authors rarely explain their motivation for doing so, nor do they consider consequences for the evaluation. Also, our analysis does not address manual work, which may be distributed unfairly between different fuzzers, for example, giving one fuzzer a fine-tuned configuration that performs better.

Initial seeds. Another crucial factor determining a fuzzer's performance is the set of initial seeds [73], [88]. We studied if the *type* of seeds is specified and if information on concrete seed files is available. Out of the 150 papers, 11% (16) require no seeds, 25% (38) use uninformed or empty seeds, 20% (30) use informed seeds, 16% (24) use seeds provided by the project as test cases or those that are shipped with a benchmark, and 3% (5) use multiple types of seed sets, while 25% (37) do not specify at all what type of seeds are used, making a reproduction challenging. Regarding concrete details, we find that 50% (75) of the papers fail to disclose what seeds they use, compared to 39% (59) that outline their seeds. A further pitfall potentially threatening an evaluation's validity is the fair distribution of the same seeds to all fuzzers. While this is the case in 46% (69) of the studied papers, in 30% (45) of the works this does not become clear, and 5% (8) even use diverging seed sets. Three of these cases arise due to the fuzzer design or other fuzzers lacking the capability to process a particular type of input. We stress that this may be valid, for example, when a fuzzer used for comparison needs a larger seed set than the proposed fuzzer, yet giving a fuzzer a different set of seeds requires special attention and documentation.

We find that 5% of the papers allocate computing resources unfairly, and 5% use different seed sets.

3.2.5. Evaluation Metrics. While many different metrics exist, often specific to the particular technique introduced, a small number of metrics has found widespread adoption: 77% (115) of the papers use some sort of *code coverage*, and 71% (107) use the (re-)discovery of bugs as a metric to compare fuzzers. The third most widespread metric, Time-To-Exposure (TTE), is used by 13% (20) of the papers, mainly from the directed fuzzing domain.

Code Coverage. Code coverage comes in different forms; the most popular are the following: 19% (29) of the papers use branch coverage, 17% (25) employ edge coverage, 13% (19) rely on basic block coverage, and 5% (8) use line coverage on the source code level. Furthermore, 11% (17) use some notion of paths to measure coverage. We stress this metric is unreliable *without* a definition of what the paper considers a path. Differences exist, for example, between actual program paths and AFL's path metric, requiring any paper to specify what they consider a path for their work. Beyond the type of coverage, the process of measuring coverage is also prone to errors, and the concrete choice of measurement is often not documented. In total, we find that 45% (67) of the works lack a clear definition or explanation of how they measure coverage, whereas 32% (48) document this (the remaining papers do not measure coverage). For example, measuring coverage using a binary with instrumentation that not all fuzzers had access to during the fuzzing campaign gives some fuzzers an advantage. Similarly, when measuring coverage on a bitmap with collisions, the reported coverage is up to 9% smaller [103] than the true one. This may cause problems when a different bitmap size was used during fuzzing, as the inputs saved by a fuzzer may no longer trigger the new coverage on the bitmap with collisions. A further pitfall affects emulation-based fuzzing, especially when using QEMU [11]. We observed that papers often provide no clear distinction between translated blocks as presented by the emulator and actual basic blocks for the target binary. We found that in at least one case this led to overcounting the reached coverage, as translated blocks were mistaken for basic blocks.

Known Bugs. As research from Klees et al. [88] as well as Böhme et al. [23] points out, coverage may not be an accurate proxy for bug finding, even though a strong correlation exists. Ultimately, a fuzzer's goal is finding bugs, making the evaluation of whether it can find known or unknown vulnerabilities an excellent experiment. Known bugs are a good way of measuring a fuzzer's performance, yet it is difficult to find suitable bugs outside well-designed benchmarks, such as Magma [70] or RevBugBench [183].

New Bugs / CVEs. Another commonly used approach is the capability of finding previously unknown bugs. Ethical handling requires researchers to responsibly disclose these bugs to the vendors or maintainers. Both sides can additionally request a CVE that serves as a unique identifier for the found vulnerability. In practice, CVEs have become a commonly used metric to assess whether a fuzzer can find bugs in real-world software, presumably showing its impact. Of the 150 analyzed papers, 59 claim one or more CVEs (9.7 on average, 662 in total). Given the implicit expectation of submissions to have a real-world impact, the authors often try to obtain as many CVEs as possible. We randomly selected 35 of these papers [9], [19], [33], [35], [36], [40], [41], [47], [49], [52], [72], [75], [77], [82], [93], [96], [97], [105], [106], [109], [110], [115], [120], [129], [130], [139], [146], [160], [164], [174], [175], [184]–[186], [189] and analyze the 339 CVEs they claim (51% of all CVEs claimed across the 59 papers).

As Figure 3 shows, surprisingly, only 43% (145) of the CVEs are valid (i. e., neither formally disputed, reserved, nor ignored or rejected by the project maintainers) and have been fixed (or at least acknowledged). 26% (88) of the CVEs were still marked as `RESERVED`, preventing us from viewing and analyzing them (all of them were assigned before 2023). For such CVEs and depending on the assigning

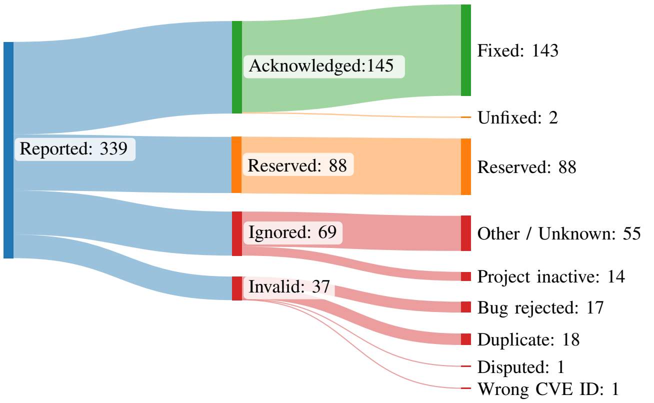

Figure 3. Outcome of 339 CVEs that were reported across 35 papers. Only 43% of the CVEs have been acknowledged by the developers. Pending public disclosure, information on CVEs in the *Reserved* state is withhold.

authority (called CNA), authors usually have to follow up with the CNA to unblind them once the vulnerabilities are publicly disclosed. Our analysis found 11% (37) of invalid CVEs, including both CVEs that were formally disputed or rejected as duplicates by the assigning CNA, such as MITRE, and such CVEs where the maintainer of the project considered the report to be invalid or not a bug. In one case, the CVE ID specified in the paper did not match the target, leading us to believe the authors mistakenly reported the wrong number. Three CVEs were claimed by more than one paper, raising questions about who identified and reported them initially. A larger number, 20% (69) of the CVEs, have been ignored by the maintainers of the respective projects. Investigating this, we found that in 14 cases, the projects were abandoned several years before the bug was found, or the projects had not found widespread adoption (with a single digit number of stars and forks on GitHub). In these cases, the perceived need to report many vulnerabilities in a paper appears to be the driving factor in requesting a CVE for such bugs.

Studying why some bug reports were ignored while other bugs were fixed, we found that maintainers tend to ignore issues such as memory leaks in client-side software, for example, an assembler. The reasoning appears to be that the program does not run continuously and is not exposed to external attackers. Many of the ignored CVEs were segmentation faults in `mjs` or `yasm`. The bug tracker of `mjs` appears to be flooded with similar fuzzer-generated bug reports, while the project has not received an update for two years. Similarly, the maintainer of `yasm` has moved to other projects, only occasionally merging pull requests. As security researchers usually only drop the bug details without proposing a patch, these issues remain unfixed. While studying papers, we noticed that several papers claim a specific number of CVEs credited to their work but do not specify any identifier, making it difficult to track them. Interestingly, 18 of the 35 papers report only CVEs that all have been fixed, accounting for 67 of the CVEs.

In summary, the need to show a fuzzer's real-world impact results in a large number of unwarranted CVEs, leading to a situation where only 42% (143) of the 339 assigned CVEs are valid and have been fixed, while many are what one maintainer referred to as "fuzzer fake CVEs" [114]. Creating such invalid vulnerabilities causes multiple problems: It unnecessarily alerts people, reduces maintainer acceptance of fuzzer findings, and raises the expectations for subsequent papers to find a similar number of vulnerabilities.

> 20% of the CVEs have been ignored and remain unfixed, 11% are invalid. 26% are reserved, eluding analysis.

3.2.6. Statistical Evaluation. To confirm the results obtained in the evaluation, a statistical evaluation is highly recommended [88], [127] to detect whether the observed difference is significant or by chance. In practice, the most common approach is to compare the final coverage values achieved by different fuzzers across multiple runs.

In general, a frequently used test for the comparison of the means of two sample sets—such as the coverage values of two fuzzers operating on the same target—is the t-test. While powerful for the detection of differences, it requires strong assumptions. In particular, the samples have to be approximately normally distributed. This is particularly true for small sample sizes, such as $n \approx 10$. To avoid these strong assumptions, the Mann-Whitney or the similar U-test (called Mann-Whitney U-test to emphasize their equivalence subsequently [138]) is often used. Here, the two samples are assumed to have the same unknown distribution except for a potential shift. The test statistics for the Mann-Whitney U-test is mainly based on the sum of ranks of the two samples in the joint sample. This results in a test for the difference of distribution medians, which is rather robust w.r.t. assumptions that do not hold. For a more detailed discussion of such tests, we refer to Sachs' work [138].

However, the Mann-Whitney U test can have low power, especially for small sample sizes. Suppose, for example, that we have two samples of three runs that achieved the following coverage:

$$x = (1000, 1002, 1001), \quad y = (1208, 1207, 1205)$$

As is easy to see, these samples are strongly separated, and it is hard to explain these results assuming the similarity of the samples' distributions. Yet, the Mann-Whitney U test will not reject the hypothesis of no difference for a significance level $\alpha = 5\%$. Even worse, it will never reject samples of this size, since it only uses the ordering of the observations, and the probability of two samples of size 3 generated from the same distribution to show this pattern of full separation on the real line has a probability $> 5\%$. In other words, we cannot use the Mann-Whitney U test to statistically confirm that the difference between two fuzzers is significant if only three trials have been conducted. Such situations frequently arise if sample sizes are small or observations cannot be approximately described by a

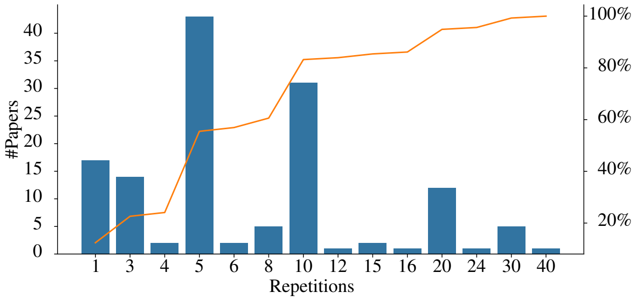

Figure 4. Distribution of trials used in practice and cumulative distribution function (CDF). 8 papers use a different number of trials for different experiments; we include all numbers in this case. Further 21 papers fail to specify the number of trials.

parametric distribution that depends only on few parameters, such as a normal distribution.

In summary, a statistical evaluation should use a sufficient number of trials, ideally 10 or more, and use a robust test. Studying the trials used in the 150 analyzed papers, we find that 1, 3, 5, 10, or 20 trials are the most common repetitions chosen. Figure 4 provides a detailed distribution. Overall, 55% (83) of the papers use fewer than 10 trials in at least one experiment (8 papers use a different number of trials throughout their paper). Even worse, 63% (94) conduct no statistical test at all. Only 37% (55) of the papers run a Mann-Whitney U test to measure statistical significance, which—paired with few trials—risks that it may never reject the hypothesis. We find that 15% (22) of the analyzed papers conduct a Mann-Whitney U-test while having five or less trials. One work reports p-values without specifying how they have been derived. Interestingly, we found no other tests, such as bootstrap-based ones, being used, despite being recommended by Klees et al. [88]. Beyond measuring statistical significance, it is recommended to quantify the *effect size*, for example, using Vargha and Delaney's $\hat{A}_{12}$ test [156]. Yet, we find that only 10% (15) of studies conduct this test; 2% (3) rely on other means to specify the effect size, leaving us with 88% (132) not using any test to measure the effect size.

Beyond the use of statistical tests, we find that 73% (109) of the papers provide no measure of *uncertainty*, for example, intervals in coverage plots or the standard deviation. This makes it difficult to assess the robustness of reported results, especially considering the inherent randomness in fuzzing runs.

> 63% of the works use no statistical test to assess their results, and 15% use too few trials to achieve robust outcomes. 73% provide no measure of uncertainty.

3.2.7. Threats to Validity. Scientific works often use a dedicated section on *threats to validity* to enumerate, reflect, and address any issue that could potentially render their evaluation invalid. However, when studying how many of the 150 analyzed papers provide such a section, we find that only a minority of 20% (30) of the papers does so.

4. Artifact Evaluation

Beyond studying the evaluation outlined and described in the papers, we select eight papers and study their artifacts. This allows us to assess the practical reproducibility of fuzzing research and provide recommendations grounded in practice. As selection criteria, we pick four recent papers from 2023 and focus on security venues featuring an artifact evaluation. In our experience, papers undergoing an artifact evaluation process provide enhanced documentation and significantly ease the process of setting up a particular tool. However, we test papers that have not undergone artifact evaluation as well to gain a more complete picture. Note that all papers we chose as case studies had attracted our attention during the initial reading for the literature survey in terms of evaluation setup or execution.

In the following, we discuss our lessons learned, pitfalls, and how fuzzing artifacts can be further improved to enhance their reproducibility. Again, we emphasize that it is not our intention to point fingers at specific works but rather to highlight potential pitfalls that researchers in this area should be aware of. More information on all case studies is available in dedicated reproduction repositories on GitHub: https://github.com/fuzz-evaluator/. Despite our best efforts, our reproduction may contain errors. If we become aware of any, we will update the respective reproduction repositories on GitHub.

Author Contact. We have anonymously contacted the authors of all case studies and brought up our findings for discussion with them, asking for their help, confirmation, or clarification. Five groups have responded to our mails. Where desired by the authors, we publish a statement of them alongside our reproduction artifact.

Setup. All our experiments were performed on two servers running Ubuntu 22.04 with 196 GB RAM, one with an Intel Xeon Gold 6230R CPU with 52 cores at 2.10GHz, and the other with an Intel Xeon Gold 6230 CPU with 40 cores at 2.10GHz (for consistency, a case study was fully run on one type of server or the other). We use the settings provided by the original papers where sensible, otherwise we run 10 trials for 24 hours each, restricting each fuzzer to a single core.

4.1. Case Study: Artificial Runtime Environment and Unique Crashes

Our first case study is MemLock [164], published at ICSE'20, which proposes to use memory usage as additional feedback. This way, the paper aims to identify resource exhaustion bugs, such as stack exhaustion.

Artifact status. MemLock has undergone artifact evaluation and received the *available* and *reusable* badges. Our additional experiments can be found at https://github.com/fuzz-evaluator/MemLock-Fuzz-eval.

Observations. After studying the paper and artifact, we observe the following:

1) According to the artifact but not documented in the paper, the authors artificially alter the runtime environment of one target and lower the maximum stack size. Manually limiting the stack size makes it easier to trigger stack overflow bugs, one of the declared goals of the presented technique.
2) MemLock, similar to many other fuzzing papers, relies on unique crashes as reported by AFL to draw conclusions on the fuzzer's performance. This metric is generally unreliable since a unique crash depends on the set of exercised edges; it does not reflect the number of actual bugs. Here, MemLock's use of the call stack depth as additional feedback may lead to an inflated number of "unique" crashes per root cause.
3) To demonstrate practical impact, MemLock reports 26 CVEs. We found multiple cases among them where up to five CVEs were requested and assigned for a single bug report, to which none of the maintainers responded.
4) MemLock's artifact is based on PerfFuzz [94] (itself an AFL-derivative), but the paper suggests it is based on AFL.

We design three experiments to analyze and reproduce MemLock's performance. For full details, we refer the interested reader to our reproduction artifact.

Experiment 1: Artificial Runtime Limits. We first study the impact of artificially lowering the stack size for the target `flex`, which was not documented in the paper. After recreating the setup and running the fuzzing campaign with and without the artificial limit, we observe that MemLock finds the claimed crashes only with the artificially lowered limit. While memory corruption bugs may warrant discussing artificial scenarios, we believe memory exhaustion created through artificial limits cannot be considered realistic. In any case, we recommend documenting such limits in the paper.

Experiment 2: Unique Crashes. We investigate whether superiority claimed due to *unique crashes* persists when examining the underlying bugs and root causes. Using a developer patch and manual triaging, we identify the underlying bugs for three evaluation targets and find that AFL finds four bugs, while MemLock locates only three, even though it finds significantly more unique crashes.

Experiment 3: Reported CVEs. When studying the reported vulnerabilities, we noticed that six CVEs, CVE-2020-36370 to CVE-2020-36375, refer to the same bug in `mjs`. This bug was never acknowledged by the maintainers of `mjs`. This pattern repeats for other groups of CVEs.

Lessons learned: Unique crashes are not a reliable metric; instead, we suggest using (known) bugs. We recommend not using artificial runtime environments without good reason and, if done, documenting such limits. We strongly recommend against the practice of obtaining as many CVEs as possible. Real-world impact should not be measured based on the number of assigned CVEs.

4.2. Case Study: Exaggerated Vulnerabilities

For the next case study, we selected SoFi [72], published at ACM CCS'21. This work aims to use a reflection-based analysis to create a syntactically and semantically valid but diverse set of seeds for fuzzing JavaScript engines.

Artifact status. Artifact evaluation was not available for SoFi, but the authors released the source code via an independent web page [71]. While trying to set up the artifact, we noticed that crucial parts of the source code were missing. The authors stated they would release the missing pieces once the code is polished [71], but did not react to our e-mails asking for access to the code. Without a chance to reproduce the artifact, we solely studied the paper, in particular the reported vulnerabilities summarized in Table 2 of their paper, entitled "Summary of discovered vulnerabilities" [72].

Observations. We find that all seven vulnerabilities claimed in the actively used modern browser engines (i. e., v8, SpiderMonkey, and JavaScriptCore) are invalid and have been rejected by the respective developers, six out of seven even *before* the conference submission deadline. While SoFi manages to find confirmed vulnerabilities in other programs, we believe it is important to not oversell results by claiming to have found vulnerabilities in browser engines, when in fact they were not a bug at all. We assume that the bug report IDs were blinded, as is common practice for submission, such that the reviewers could not verify the validity of the presumed vulnerabilities.

Lessons learned: We highly discourage marketing invalid bug reports as a vulnerability. Feedback from the developers must be taken into account (especially if bug reports are rejected by the developers). Pledges to release the source code should be kept.

4.3. Case Study: Missing Baselines

DARWIN [78] was published at NDSS'23 and honored with a *Distinguished Paper Award*. The paper focuses on improving mutation scheduling. More specifically, the authors propose to use an evolution strategy and dynamically adapt the mutation selection to the target under test.

Artifact status. Artifact evaluation was not available to DARWIN, but the authors publicly released an artifact. Our reproduction artifact is available at https://github.com/fuzz-evaluator/DARWIN-eval.

Observations. Analyzing the paper and artifact, we found a number of issues:

1) Coverage differences between DARWIN and tested baselines on FuzzBench are not statistically significant nor consistent with the paper's FuzzBench results.
2) The results on MOpt [110] listed in the DARWIN paper indicate that the port implemented for MOpt may have erroneously restricted the number of usable mutations. We find that this strongly influences the results.
3) The artifact appears to be based on Git tag 2.55b of Google's AFL fork and not 2.54b, as listed in the paper.

4) The artifact does not provide the AFL 2.55b port for MOpt or their baseline AFL-S, preventing reproduction or analysis.

We design three experiments to analyze DARWIN. More experiments and details are available in our artifact.

Experiment 1: Coverage. We use FuzzBench to reproduce DARWIN's coverage measurements (in particular, Table III of their paper). Running all targets for 24 hours, we compare it against AFL 2.55b and MOpt, which is based on AFL 2.52b. Notably, we do not use DARWIN as configured in FuzzBench but follow the author's recommended configuration (see Experiment 3). In our FuzzBench results, MOpt does not show the major performance degradation shown in the paper results. Overall, FuzzBench ranks DARWIN above MOpt and AFL, both by score and rank. In individual targets, DARWIN is the best performer in nine of the targets, but only with statistical significance in four. Our results show the difference between DARWIN and its baselines to be less than reported in Table III of their paper. Where they find DARWIN's median relative coverage to be the highest for 15 out of 19 targets, we find this to be the case for 4 out of 18 targets[2] (DARWIN is worse than at least one baseline in two cases and tied with at least one baseline in the other cases). Note that the original paper evaluates over a six hour period instead of the 24 hours recommended by Klees et al. [88]. While we provide the statistical data for the 24 hour data here, we emphasize that the results reported in the paper for the six hour mark are similarly not reproducible and invite the reader to view our full evaluation report data available on GitHub.

In summary, our results show a similar tendency to their paper, but the difference observed between DARWIN and its baselines is smaller. Notably, DARWIN reports a coverage improvement of only 1.73% over AFL, making it difficult to judge the difference between these fuzzers meaningfully.

Experiment 2: New Baseline. We propose a second baseline to test DARWIN's contribution of a dynamically adapting mutation selection: we replaced its proposed weighting with a random selection (that is reweighted at a constant interval). This implementation, DARWIN$_{RAND}$, provides a new baseline that allows to better judge DARWIN's contribution, as any improvement can be directly attributed to DARWIN's evolutionary algorithm rather than other fuzzer implementation details, such as dynamically adapting mutation selection. We find in our FuzzBench results no statistical significant difference between DARWIN and DARWIN$_{RAND}$, meaning we were unable to demonstrate that the evolutionary aspects of DARWIN's approach significantly contributed to the improvement compared to randomly changing mutation selection over time.

Experiment 3: Per-Seed Mutation Scheduling. After contacting the authors, they noted that the per-seed mutation scheduling (`-p` flag) set by FuzzBench should be disabled for the evaluation because it worsens performance and was not intended as part of the paper. To confirm this, we separately evaluated DARWIN with and without per-seed mutation scheduling on seven targets: we found that disabling the per-seed mutations slightly improved performance overall, leading to higher median coverage in some targets, but not statistically significantly so for any target by Mann-Whitney U. We have used the author-recommended configuration (no `-p` flag) for Experiments 1 and 2.

Lessons learned: A baseline suited to test the proposed technique is necessary to detect differences that can be attributed to the proposed technique rather than the new fuzzer implementation as a whole. We further recommend publishing all evaluation artifacts, also including benchmarking reports and raw data.

4.4. Case Study: Non-reproducible Measurements

A recent paper published at USENIX'23, FuzzJIT [161], aims to detect bugs in JIT compilers, including those used in modern browsers.

Artifact Status. FuzzJIT underwent artifact evaluation and was awarded the *available* and *functional* badges. Our reproduction artifact can be found at: https://github.com/fuzz-evaluator/fuzzjit-eval.

Observations. After studying the paper and testing the artifact, we observe several shortcomings:

1) Coverage does not reproduce as outlined in the paper; in our experiments, FuzzJIT performed worse than Fuzzilli on all targets.
2) Reported improvements of the semantic correctness rate did not materialize in our experiments.
3) It is not possible to study the bugs found because the time frame, engine versions, and resources spent were not specified in the paper, hindering fair reproduction.

We design two experiments to analyze the claims of FuzzJIT in more detail.

Experiment 1: Code Coverage. When trying to reproduce code coverage, we find significantly different results. As shown in Table 3, FuzzJIT reports a code coverage improvement of up to 33% over Fuzzilli. In stark contrast,

TABLE 3. COMPARING THE CODE COVERAGE REPORTED BY FUZZJIT TO OUR MEASUREMENTS.

	Reported			Measured
Engine	**Fuzzilli**	**FuzzJIT**	**Rel. Increase**	**Rel. Increase**
JSC	16.47%	21.90%	33%	-2%
V8	13.82%	16.67%	21%	-3%
SM	15.53%	17.97%	16%	-12%

TABLE 4. COMPARING THE SEMANTIC CORRECTNESS RATE REPORTED BY FUZZJIT TO OUR MEASUREMENTS.

	FuzzJIT		Fuzzilli	
Engine	**Reported**	**Measured**	**Reported**	**Measured**
JSC	90.33%	65.88%	62.80%	66.56%
V8	97.04%	63.67%	64.34%	66.74%
SM	93.28%	63.93%	64.13%	67.47%

[2] FuzzBench has meanwhile removed the target `php`.

our experiments show a code coverage decrease of -2% to -12%. Despite searching for the cause, we find none explaining this difference. We speculate that the negative outcome of the comparison experiment is a consequence of benchmarking with different versions of Fuzzilli. This is based on the observation that the state-of-the-art fuzzers compared to in the evaluation are taken from UniFuzz [99], which uses an outdated version of Fuzzilli; FuzzJIT itself is based on a more recent version of Fuzzilli. Unfortunately, the authors have not responded to our request for help.

Experiment 2: Semantic Correctness Rate. Besides code coverage, FuzzJIT also evaluates the semantic correctness rate of generated samples, i. e., the number of samples that do not raise an uncaught exception during execution in the JS engine. As shown in Table 4, we could *not* measure any improvement of the semantic correctness rate, contrasting the paper's claim of a significant improvement.

Lessons learned: Relying on outdated baseline versions can create a distorted picture of a fuzzer's performance. Authors should ensure that they use the latest version of all tools for comparison.

4.5. Case Study: Uncommon Metrics

Published at USENIX'20, EcoFuzz [175] proposes to replace AFL's seed scheduling algorithm with a version relying on the adversarial multi-armed bandit model. This way, EcoFuzz finds more paths while generating less seeds.

Artifact status. EcoFuzz has undergone artifact evaluation and was awarded the *passed* badge, indicating that the artifact is available and ready to be reproduced. Our independent reproduction repository is located online at https://github.com/fuzz-evaluator/EcoFuzz-eval.

Observations. When studying the paper and artifact, we noticed that the evaluation deviates from typical fuzzing evaluations: The work does not report achieved code coverage over time. Instead, the paper visualizes the total number of paths discovered over executions. This aligns with the paper's goal of finding more path (bandits in EcoFuzz's multi-armed bandit model) with fewer executions (trials in the model). The presented results may lead readers to infer that a higher number of total paths equates to higher code coverage, which is not necessarily true.

Experiment: Code Coverage. We design an experiment in FuzzBench where we compare EcoFuzz against its best-performing competitor, AFLFast, and its baseline, AFL. We test these fuzzers on three targets, `nm`, `libpng`, and `objdump`, where the original evaluation[3] found EcoFuzz to be the fuzzer to find the *most* paths. Our results, shown in Figure 5, demonstrate that EcoFuzz achieves *less* code coverage than the other fuzzers in all scenarios, except for a statistically insignificant one, where it performs similar to AFLFast on `libpng`. This underlines that finding more paths does not necessarily translate to achieving a higher coverage. The full results and the generated FuzzBench reports can be found in our reproduction repository.

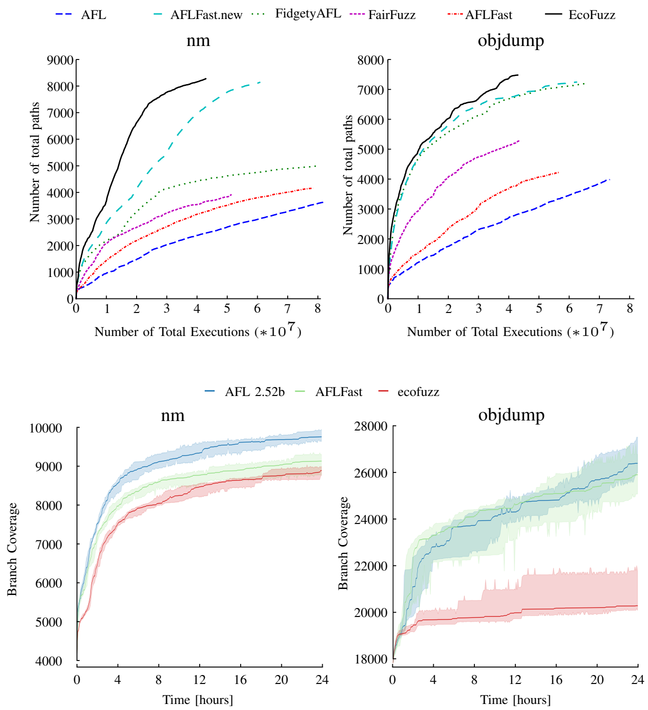

Figure 5. The upper two graphs published in the EcoFuzz paper [175] show a strong advantage over all competitors on the non-standard metric *number of totals paths over the number of total executions*. The two plots at the bottom compare EcoFuzz on the standard metric *branch coverage over time*. On the standard metric, EcoFuzz performs significantly worse.

Corresponding with the authors, they state they have been following fuzzing evaluations at the time that focused on path coverage, and they have confirmed that EcoFuzz may cover fewer branches on some binaries, stating that its goal is to optimize for paths over executions rather than branches over time.

Lessons learned: A fuzzer may excel at one metric but not on another; hence, selecting a suitable set of evaluation metrics is crucial to provide a reader with the full picture. Evaluating on established metrics is required, as new metrics may imply a completely different picture.

4.6. Case Study: Unclear Documentation

Another paper published at USENIX'23, Polyfuzz [96], targets programs containing code in different languages, such as interpreter languages calling into native bindings.

Artifact status. PolyFuzz has been awarded the *available* badge. Our reproduction artifact is available at https://github.com/fuzz-evaluator/PolyFuzz-eval.

Observations. While studying the artifact, we noticed irregularities regarding the seed sets used by PolyFuzz compared to the other fuzzers. An example of such a case is the `image_load` harness for the Python image processing

[3] The evaluation used `readpng`, which internally uses `libpng`, while we use `libpng_read_fuzzer` as bundled with FuzzBench.

library Pillow. In this particular case, the fuzzer Atheris gets 39 seed files, while PolyFuzz's seed directory has 58 files.

Experiment: Fair seed allocation. We intended to run both fuzzers with their respective seed sets to measure the impact of these different seed sets on the coverage. Unfortunately, the authors' extension of Atheris (called Atheris-Cext in the PolyFuzz paper), which would allow to compute combined coverage for both Python and the native code, was not released alongside their artifact. Hence, as proxy measurement, we compute the initial coverage achieved by PolyFuzz on both seed sets. For the seed set given to Atheris, PolyFuzz covers 218 edges, while for its own seed set, it covers 814 edges. Evidently, one seed set provides more than three times as much coverage as the other, giving PolyFuzz a headstart during the evaluation.

When contacted, the authors clarified that they did not keep the seed sets from their evaluation, but they assured us that they used the seeds from the corresponding benchmarks for all fuzzers.

Lessons learned: Seeds have an impact on fuzzer performance. We recommend to give all fuzzers the same set of seeds and to publish the seeds used.

4.7. Case Study: Incomplete Artifact

Firm-AFL [187], published at USENIX Security'19, aims to fuzz Linux-based IoT firmware via augmented process emulation. To do so, the core fuzzing loop targets a single binary under user-mode emulation, while selectively forwarding system calls to a full-system emulator.

Artifact status. Artifact evaluation was not available to Firm-AFL, but different versions of its source code are publicly available across multiple repositories. Our reproduction artifact is available at https://github.com/fuzz-evaluator/firmafl-eval/.

Observations. During our analysis of the artifact, we noticed that the repository lacks documentation. Crucial steps are missing, like correct build instructions for different configurations, making it hard for researchers to reuse the artifact and set up the fuzzer and its environment correctly. Furthermore, when setting up the experiments, we noticed that some of the experiment configuration files were missing and target harnessing is heavily inlined with core emulation logic. Not only do these issues hinder extensibility, but they also prevented us from getting all targets working to reproduce the Firm-AFL experiments. The fuzzer binaries are shipped in a pre-compiled binary version and fail to build from the provided source code. Moreover, the provided baseline uses an older version of AFL (2.06b), while the augmented mode uses AFL v2.52b.

Experiment: Crash Triggers. Being the only experiment with enough documentation to reproduce, we aim to measure the number of crashes produced by both the augmented and full-system emulator versions. We were able to run fuzzing campaigns for 9 out of 11 targets, where one of them only ran for the baseline and not Firm-AFL. The remaining two targets lack the required target-specific configurations. Unfortunately, we could only partially reproduce the claims as presented in the Firm-AFL paper and observed one case where the baseline performed better than Firm-AFL. The full results of our experiments can be found in our reproduction repository.

Lessons learned: While it is unreasonable to expect each academic artifact to be of production quality, we recommend to strive for a reasonable level of readability and documentation that allows others to understand and use the code, thus promoting reproducibility.

4.8. Case Study: Unfair Coverage Measurements

The final case study analyzes FishFuzz [186], published at USENIX'23. The paper proposes an input prioritization strategy based on a multi-distance metric that allows for optimizing the fuzzing efforts towards thousands of targets (e. g., sanitizer labels) in the sense of direct fuzzing.

Artifact status. FishFuzz has received the *available* and *functional* badges. Our additional experiments are available at https://github.com/fuzz-evaluator/FishFuzz-eval.

Observations. When studying the artifact in detail, we notice that FishFuzz's way of measuring coverage may erroneously give FishFuzz an unfair edge. From all evaluated fuzzers, FishFuzz was the only fuzzer to place coverage instrumentation not only within the actual target but also in the added ASAN instrumentation. Consequently, FishFuzz also stored inputs that exercised new coverage in the instrumentation; other fuzzers discarded these inputs, as no new coverage was observed. This became a problem when the binary instrumented by FishFuzz was used for coverage measurements for all fuzzers during evaluation since—by design—only FishFuzz would keep inputs exercising coverage in the ASAN instrumentation.

Experiment: Fair coverage measurement. To demonstrate the impact of measuring coverage in instrumentation code, we measure the coverage for a binary both with and without FishFuzz instrumentation. The result is depicted in Figure 6. If the FishFuzz coverage binary is used for coverage computation, FishFuzz covers 8.44% more edges on average over all runs. When using a binary with standard AFL instrumentation (i. e., where coverage is not measured in the additional instrumentation), the observed coverage increase is reduced to 1.69%. Furthermore, the total number of edges is considerably smaller, showing that edge counts between different binaries do not translate. Note that both coverage binaries rely on colliding bitmaps since the artifact tooling of FishFuzz expects standard AFL bitmaps to be used. We recommend to not use colliding bitmaps for coverage measurements.

Lessons learned: Unintended side effects may skew coverage measurements; we recommend using standardized methods of measuring coverage.

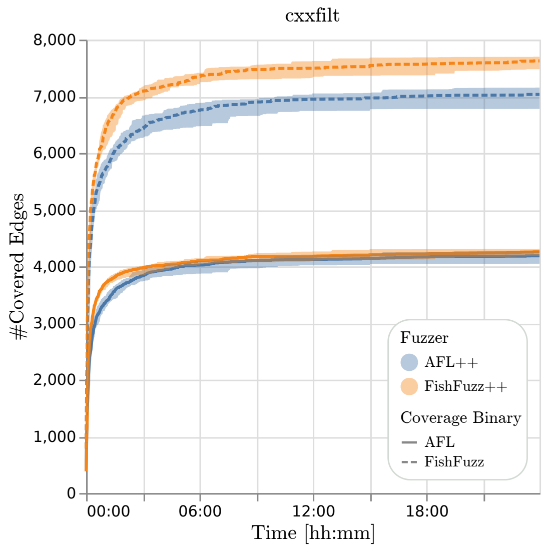

Figure 6. Median coverage over time for `cxxfilt`: In one case, we measure coverage via a standard AFL binary and, in the other we use FishFuzz's binary that contains additional coverage instrumentation. For each fuzzer, the target was run 10 times for 24h each. The displayed intervals enclose all ten runs of the respective fuzzer. If the coverage is measured on the biased binary with FishFuzz instrumentation (--), FishFuzz++ finds on average 8.44% more edges than AFL++. Measuring coverage on a standard AFL binary (—) (without additional instrumentation introduced by FishFuzz), the coverage delta is only 1.69%.

5. Revised Best Practices for Evaluation

Based on our literature analysis and the case studies, we now provide recommendations on ensuring a fair and reproducible fuzzing evaluation. A comprehensive checklist that summarizes these recommendations is available in our GitHub repository at https://github.com/fuzz-evaluator/guidelines. Overall, we recommend that authors thoroughly review the *threats to validity* for their respective works to reflect potential issues that could invalidate their evaluation.

5.1. Reproducible Artifact

For reproducibility, it is crucial to open-source the source code including documentation. We highly recommend participating in an artifact evaluation if available. Furthermore, it is essential to (i) specify the exact versions of targets (and harnesses) and fuzzers used for comparison, (ii) use runtime environment abstractions, such as Docker (where feasible), (iii) name the baseline on which the new technique is implemented upon (if any) as well as its version, and avoid squashing commits of this baseline. In the long term, a mandatory artifact evaluation as part of the submission process could improve the quality and reproducibility of research artifacts.

5.2. Targets under Test

Selected evaluation targets should form a representative set that shows strengths of the proposed approach and allows for comparability with previous work. It is therefore desirable to include targets that have been tested in other works. Actions such as patches applied to targets should be explained. If a fuzzer has certain restrictions (such as symbolic execution-based techniques not being able of modeling all syscalls), we recommend outlining those. We also highly recommend using well-established benchmarks, such as FuzzBench, to facilitate easy reproducibility.

5.3. Comparison to Other Fuzzers

It is crucial to compare against the state of the art in the respective field and the baseline (if any) on which the new technique is implemented. This also includes well-established and actively maintained fuzzers, such as AFL++. Including the new fuzzer in benchmarks such as FuzzBench allows for comparing against a wide range of fuzzers. If presenting a new technique with separable design choices, review them individually via ablation studies, for example, by designing baselines that successively enable or disable individual components.

5.4. Evaluation Setup

The chosen evaluation setup should be well documented. This entails details regarding the used hardware, experiment runtime, number of allocated cores, and processes per fuzzer. The conducted experiments and how to reproduce them should be explained.

For the runtime, we recommend choosing at least 24 hours. Longer runtimes may be appropriate if the evaluated fuzzers do not flatline at the end of the experiment. Regarding CPU cores, choosing a single core may not be representative of modern systems. Special care must be taken to avoid congestion in the kernel when running multiple fuzzers in parallel on one system; even if using Docker, the kernel may become a bottleneck in resolving certain syscalls, unfairly slowing down one fuzzing process. Individual fuzzer instances can be encapsulated in separate virtual machine instances to avoid such situations.

Regarding seeds, we recommend running with uninformed seeds or multiple seed sets. Seeds must be described and accessible (in the case of informed seeds) to allow for reproducibility. All fuzzers should have fair access to all seeds. If using informed seeds, we recommend plotting or analyzing the coverage achieved by the initial seed set. This avoids attributing a high coverage achieved to fuzzer performance instead of the initial seeds.

5.5. Evaluation Metrics

A fuzzer comparison should use standardized, well-established metrics (at least as a complementary metric if a technique requires the introduction of a new metric); this includes both coverage and found bugs. Optimally, both code coverage and bug-finding capability are evaluated, as both suffer from individual drawbacks [23], [88], [179]. We recommend using modern benchmarks that aid in setting up the experiment and ensure a fair, bias-free execution.

It is necessary to specify details such as how coverage is collected, for example, whether it is measured on a non-instrumented binary, translated blocks from an emulator, or using established means such as `lcov`. Ideally, coverage is not measured using bitmaps with collisions, but using a collision-free encoding or other means. Additionally, the evaluation must ensure that the same notion of coverage is used for each of the compared fuzzers.

When searching for bugs in new targets to show real-world impact, it is crucial to select reasonable targets, i. e., projects that are not insecure by design, have been inactive for years, or are unsuitable for other reasons. We also recommend running other state-of-the-art fuzzers to see whether they find the bugs as well, thereby addressing concerns regarding fuzzing previously untested software. Crashes identified by the fuzzer should be deduplicated before opening a report, and the triaging process should be clearly described. When testing crashes, we recommend reproducing them on a binary without fuzzer or coverage instrumentation to avoid reproducibility issues.

Ideally, only maintainers should request CVEs. If they do not request one, researchers can still link to the bug report instead. Requesting multiple CVEs for a single bug or requesting CVEs without coordinating or informing the maintainers must be avoided. If possible, reporting bugs or CVEs anonymously allows for providing the reviewers with access during submission, such that they can inspect the CVEs or bug reports and assess their validity (as opposed to the current practice of blinding CVEs and bug reports during submission, preventing any analysis by reviewers). That said, we do not believe that having CVEs should be required to show the practical impact of a fuzzer.

5.6. Statistical Evaluation

Any evaluation should be backed by statistical tests. To enable these tests, we recommend running at least ten trials. Alternatively, the number of trials can be calculated via an a-priori power analysis to ensure a sufficient sample size leading to statistically significant results [44]. This is particularly important if the fuzzer under consideration only slightly outperforms the state of the art, where $n \gg 10$ may be required. To avoid the problems mentioned in Section 3.2.6, we recommend an alternative to the widely used Mann-Whitney-U test; permutation tests or resampling tests such as bootstrap methods. These methods avoid strong assumptions regarding a normal distribution.

If more than two fuzzers have been compared for a target, the (bootstrap-based) two-sample t-test is not a good choice, since we would have to perform more than one pairwise comparison to test the null hypotheses of no difference between any of the expected means for the fuzzing methods. This results in the *multiple testing problem*, which is the observation that the probability of at least one false positive result in the set of comparisons performed for a target exceeds the single test level α substantially. The same argument holds for other strategies based on two-sample comparisons such as the Mann-Whitney-U test [3].

A solution to this problem is the bootstrap version of the *ANOVA method*. If the ANOVA rejects the null hypothesis, it shows at level α that there is at least one pair of fuzzing methods that perform significantly different for the target considered. In a second step, a so-called *Posthoc*-test is performed to determine which pairwise comparisons are significant, *given that the ANOVA has already shown that there are significant differences at all*. Possible *Posthoc*-tests are, for example, the Tukey-Kramer method if all pairwise comparisons among all samples are of interest or the Dunnett method if only the comparisons to a reference method, such as the newly developed fuzzer, are of interest [138]. For a bootstrap version of these algorithms, we propose as a simple solution two-sample t-tests with critical values for rejection based on a bootstrap resampling with replacement of the test statistics. Here, for each simulation, the maximum value of the test statistics is used for all pairwise comparisons of interest. We provide more details, algorithms, and scripts implementing examples for these tests in our artifact at https://github.com/fuzz-evaluator/statistics. Additionally, evaluations should measure *effect size*, e. g., using Vargha and Delaney's $\hat{A}_{12}$ test [156], and quantify *uncertainty*, for example, by using intervals in plots.

6. Conclusion

Reproducibility is a cornerstone of science and the basis for research. In this work, we have systematically studied how 150 fuzzing papers published in the past six years at leading conferences design their evaluation. Furthermore, we have performed an in-depth analysis of the artifacts of eight papers and attempted to reproduce their results. Based on the insights gained, we outlined several potential pitfalls and shortcomings threatening the validity of fuzzing evaluations. Ultimately, we provided revised recommendations and best practices to improve future evaluation of fuzzing research. We published a concise set of guidelines at https://github.com/fuzz-evaluator/guidelines and welcome community contributions.

Acknowledgment

We thank our anonymous shepherd and reviewers for their valuable feedback. Further, we thank Dominik Maier, Johannes Willbold, Daniel Klischies, Merlin Chlosta, and Marcel Böhme (in no particular order) for their helpful comments on a draft of this work. We also thank the countless researchers with whom we have discussed fuzzing research and how to evaluate it, ultimately paving the way for this work. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research under the grants KMU-Fuzz (16KIS1898) and CPSec (16KIS1899). Additionally, this research was partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/V000454/1. The results feed into DsbDtech.

References

[1] M. Abadi and R. Needham, “Prudent Engineering Practice for Cryptographic Protocols,” *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 6–15, 1996.

[2] I. Angelakopoulos, G. Stringhini, and M. Egele, “FirmSolo: Enabling Dynamic Analysis of Binary Linux-based IoT Kernel Modules,” in *USENIX Security Symposium*, 2023.

[3] A. Arcuri and L. Briand, “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering,” in *International Conference on Software Engineering (ICSE)*, 2011.

[4] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, “Dos and don’ts of machine learning in computer security,” in *USENIX Security Symposium*, 2022.

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “NAUTILUS: Fishing for Deep Bugs with Grammars,” in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[6] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, “Ijon: Exploring Deep State Spaces via Fuzzing,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[7] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, “REDQUEEN: Fuzzing with Input-to-State Correspondence,” in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[8] Association for Computing Machinery, “Artifact Review and Badging Version 1.1,” 2020. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[9] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, “Stateful Greybox Fuzzing,” in *USENIX Security Symposium*, 2022.

[10] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller, and T. Holz, “Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge,” in *USENIX Security Symposium*, 2023.

[11] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *USENIX Annual Technical Conference (ATC)*, 2005.

[12] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, “JIT-Picking: Differential Fuzzing of JavaScript Engines,” in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[13] T. Blazytko, C. Aschermann, M. Schloegel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, “GRIMOIRE: Synthesizing Structure while Fuzzing,” in *USENIX Security Symposium*, 2019.

[14] M. Böhme, C. Cadar, and A. Roychoudhury, “Fuzzing: Challenges and Reflections,” *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based Greybox Fuzzing as Markov Chain,” *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[16] L. Borzacchiello, E. Coppa, and C. Demetrescu, “Fuzzing Symbolic Expressions,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[17] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, “No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions,” in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[18] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, “Evaluating Synthetic Bugs,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[19] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, “TEEzz: Fuzzing Trusted Applications on COTS Android Devices,” in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[20] M. Böhme and B. Falk, “Fuzzing: On the Exponential Cost of Vulnerability Discovery,” in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[21] M. Böhme, D. Liyanage, and V. Wüstholz, “Estimating Residual Risk in Greybox Fuzzing,” in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[22] M. Böhme, V. J. M. Manès, and S. K. Cha, “Boosting Fuzzer Efficiency: An Information Theoretic Perspective,” in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[23] M. Böhme, L. Szekeres, and J. Metzman, “On the Reliability of Coverage-Based Fuzzer Benchmarking,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[24] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, “MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs,” in *USENIX Security Symposium*, 2020.

[25] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a Desired Directed Grey-box Fuzzer,” in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[26] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing,” in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[27] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, “SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis,” in *USENIX Security Symposium*, 2022.

[28] J. Chen, J. Wang, C. Song, and H. Yin, “JIGSAW: Efficient and Scalable Path Constraints Fuzzing,” in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[29] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang, and Z. Xue, “SFuzz: Slice-based Fuzzing for Real-Time Operating Systems,” in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[30] P. Chen and H. Chen, “Angora: Efficient Fuzzing by Principled Search,” in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[31] P. Chen, J. Liu, and H. Chen, “Matryoshka: Fuzzing Deeply Nested Branches,” in *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[32] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, “HOPPER: Interpretative Fuzzing for Libraries,” in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[33] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, “SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers,” in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[34] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, “SAVIOR: Towards Bug-Driven Hybrid Testing,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[35] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers,” in *USENIX Security Symposium*, 2019.

[36] Y. Chen, T. Su, and Z. Su, “Deep Differential Testing of JVM Implementations,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[37] Z. Chen, S. L. Thomas, and F. D. Garcia, “MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware,” in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[38] M. Cho, S. Kim, and T. Kwon, “Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing,” in *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[39] J. Choi, J. Jang, C. Han, and S. K. Cha, “Grey-box Concolic Testing on Binary Code,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[40] J. Choi, K. Kim, D. Lee, and S. K. Cha, “NtFuzz: Enabling Type-

Aware Kernel Fuzzing on Windows with Static Binary Analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[41] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks," in *USENIX Security Symposium*, 2023.

[42] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *USENIX Security Symposium*, 2020.

[43] T. Cloosters, J. Willbold, T. Holz, and L. Davi, "SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing," in *USENIX Security Symposium*, 2022.

[44] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Academic press, 2013.

[45] DARPA, "DARPA Cyber Grand Challenge," 2018. [Online]. Available: https://github.com/CyberGrandChallenge

[46] N. Demir, M. Große-Kampmann, T. Urban, C. Wressnegger, T. Holz, and N. Pohlmann, "Reproducibility and Replicability of Web Measurement Studies," in *ACM Web Conference 2022*, 2022.

[47] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[48] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[49] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[50] Dmitry Vyukov and Google, "Syzkaller – Kernel Fuzzer," 2015. [Online]. Available: https://github.com/google/syzkaller

[51] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale Automated Vulnerability Addition," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[52] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[53] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *USENIX Security Symposium*, 2020.

[54] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[55] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The Use of Likely Invariants as Feedback for Fuzzers," in *USENIX Security Symposium*, 2021.

[56] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[57] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[58] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-Free DBMS Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[59] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing," in *USENIX Security Symposium*, 2020.

[60] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[61] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[62] Google, "OSS-Fuzz: Continuous Fuzzing for Open Source Software." [Online]. Available: https://github.com/google/oss-fuzz

[63] ——, "Fuzzer-Test-Suite," 2016. [Online]. Available: https://github.com/google/fuzzer-test-suite

[64] H. Green and T. Avgerinos, "GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[65] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities," in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[66] T. Gu, X. Li, S. Lu, J. Tian, Y. Nie, X. Kuang, Z. Lin, C. Liu, J. Liang, and Y. Jiang, "Group-based Corpus Scheduling for Parallel Fuzzing," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[67] S. Guo, X. Wan, W. You, B. Liang, W. Shi, Y. Zhang, J. Huang, and J. Zhang, "Operand-Variation-Oriented Differential Analysis for Fuzzing Binding Calls in PDF Readers," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[68] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "AntiFuzz: Impeding Fuzzing Audits of Binary Executables," in *USENIX Security Symposium*, 2019.

[69] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[70] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 4, no. 3, pp. 49:1–49:29, 2020.

[71] X. He, X. Xie, Y. Li, J. Sun, F. Li, W. Zou, Y. Liu, L. Yu, J. Zhou, W. Shi, and W. Huo, "SoFi Artifact," 2021. [Online]. Available: https://sites.google.com/view/sofi4js/souce-and-data

[72] ——, "SoFi: Reflection-Augmented Fuzzing for JavaScript Engines," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[73] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed Selection for Successful Fuzzing," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[74] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[75] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[76] A. Humayun, Y. Wu, M. Kim, and M. A. Gulzar, "NaturalFuzz: Natural Input Generation for Big Data Analytics," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[77] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic Fuzzer Generation," in *USENIX Security Symposium*, 2020.

[78] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the Fittest Fuzzing Mutators," in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[79] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[80] H. Jia, M. Wen, Z. Xie, X. Guo, R. Wu, M. Sun, K. Chen, and H. Jin, "Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[81] J. Jiang, H. Xu, and Y. Zhou, "RULF: Rust Library Fuzzing via API Dependency Graph Traversal," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[82] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and Improving Hybrid Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[83] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring Bug Capabilities from a Single PoC," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[84] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[85] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-Fuzzing Techniques," in *USENIX Security Symposium*, 2019.

[86] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[87] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[88] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[89] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "CONFETTI: Amplifying Concolic Guidance for Fuzzers," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[90] lafintel, "laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations." [Online]. Available: https://lafintel.wordpress.com

[91] G. Lee, W. Shim, and B. Lee, "Constraint-guided Directed Greybox Fuzzing," in *USENIX Security Symposium*, 2021.

[92] M. Lee, S. Cha, and H. Oh, "Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[93] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer," in *USENIX Security Symposium*, 2020.

[94] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[95] C. Lemieux and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[96] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems," in *USENIX Security Symposium*, 2023.

[97] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[98] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware Adaptive Fuzzing for Effective Vulnerability Detection," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[99] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers," in *USENIX Security Symposium*, 2021.

[100] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "PATA: Fuzzing with Path Aware Taint Analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[101] "LibFuzzer - A Library for Coverage-guided Wuzz Testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[102] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[103] S. Lipp, D. Elsner, T. Hutzelmann, S. Banescu, A. Pretschner, and M. Böhme, "FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer," in *International Conference on Software Engineering (ICSE)*, 2022.

[104] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "VIDEZZO: Dependency-aware Virtual Device Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[105] Y. Liu, S. Chen, Y. Xie, Y. Wang, L. Chen, B. Wang, Y. Zeng, Z. Xue, and P. Su, "VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[106] Y. Liu, Y. Wang, P. Su, Y. Yu, and X. Jia, "InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[107] D. Liyanage, M. Böhme, C. Tantithamthavorn, and S. Lipp, "Reachable Coverage: Estimating Saturation in Fuzzing," in *International Conference on Software Engineering (ICSE)*, 2023.

[108] C. Luo, W. Meng, and P. Li, "SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[109] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations," in *USENIX Security Symposium*, 2023.

[110] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *USENIX Security Symposium*, 2019.

[111] C. Lyu, J. Xu, S. Ji, X. Zhang, Q. Wang, B. Zhao, G. Pan, W. Cao, P. Chen, and R. Beyah, "MINER: A Hybrid Data-Driven Approach for REST API Fuzzing," in *USENIX Security Symposium*, 2023.

[112] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[113] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[114] M. Matz, "Comment 1," 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87675#c1

[115] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time Temporal Logic guided Greybox Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[116] R. Meng, G. Pirlea, A. Roychoudhury, and I. Sergey, "Greybox Fuzzing of Distributed Systems," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[117] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[118] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[119] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[120] C. Myung, G. Lee, and B. Lee, "MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference," in *USENIX Security Symposium*, 2022.

[121] S. Nagy and M. Hicks, "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[122] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[123] R. Natella and V.-T. Pham, "ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[124] H. L. Nguyen and L. Grunske, "BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[125] H. L. Nguyen, N. Nassar, T. Kehrer, and L. Grunske, "MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[126] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: Differential Fuzzing for Side-channel Analysis," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[127] D. Paaßen, S. Surminski, M. Rodler, and L. Davi, "My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers," in *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[128] L. Padgham, Y. Lee, S. Sadiq, M. Winikoff, A. Fekete, S. MacDonell, D. Kaafar, and S. Zollmann, "CORE Rankings." [Online]. Available: https://www.core.edu.au/conference-portal

[129] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *USENIX Security Symposium*, 2018.

[130] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[131] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[132] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[133] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[134] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer, "GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation," in *USENIX Security Symposium*, 2023.

[135] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *USENIX Security Symposium*, 2020.

[136] ——, "SymQEMU: Compilation-based Symbolic Execution for Binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[137] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *USENIX Security Symposium*, 2020.

[138] L. Sachs, *Applied Statistics: A Handbook of Techniques*, 2nd ed., ser. Springer Series in Statistics. New York, NY: Springer New York, 1984.

[139] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level Fuzzing," in *USENIX Security Symposium*, 2021.

[140] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *USENIX Security Symposium*, 2022.

[141] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[142] ——, "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types," in *USENIX Security Symposium*, 2021.

[143] L. Seidel, D. Maier, and M. Muench, "Forming Faster Firmware Fuzzers," in *USENIX Security Symposium*, 2023.

[144] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, and S. Jana, "MC2: Rigorous and Efficient Directed Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[145] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: Fuzzing with a Multi-task Neural Network," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[146] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient Fuzzing with Neural Program Smoothing," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[147] D. She, A. Shah, and S. Jana, "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[148] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds," in *USENIX Security Symposium*, 2022.

[149] J. Shi, Z. Wang, Z. Feng, Y. Lan, S. Qin, W. You, W. Zou, M. Payer, and C. Zhang, "AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering," in *USENIX Security Symposium*, 2023.

[150] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints," in *USENIX Security Symposium*, 2020.

[151] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[152] S. Song, C. Song, Y. Jang, and B. Lee, "CrFuzz: Fuzzing Multipurpose Programs through Input Validation," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[153] L. Stone, R. Ranjan, S. Nagy, and M. Hicks, "No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting," in *USENIX Security Symposium*, 2023.

[154] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems," in *USENIX Security Symposium*, 2018.

[155] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "SoK: Benchmarking Flaws in Systems Security," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[156] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[157] V. Vikram, R. Padhye, and K. Sen, "Growing A Test Corpus with Bonsai Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[158] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided Fuzzer for Discovering Use-after-free Vulnerabilities," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[159] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, "MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[160] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware Greybox Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[161] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler," in *USENIX Security Symposium*, 2023.

[162] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[163] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[164] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "MemLock: Memory Usage Guided Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[165] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[166] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, G. Yang, and Y. Zhang, "SJFuzz: Seed & Mutator Scheduling for JVM Fuzzing," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[167] V. Wüstholz and M. Christakis, "Targeted Greybox Fuzzing with Static Lookahead Analysis," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[168] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data Race Fuzzing for Kernel File Systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[169] P. Xu, Y. Wang, H. Hu, and P. Su, "COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[170] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[171] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[172] W. You, X. Liu, S. Ma, D. M. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without Valid Seed Inputs," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[173] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[174] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang, and P. Su, "HTFuzz: Heap Operation Sequence Sensitive Fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[175] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit," in *USENIX Security Symposium*, 2020.

[176] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security Symposium*, 2018.

[177] M. Zalewski, "American Fuzzy Lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/

[178] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," 2019. [Online]. Available: https://www.fuzzingbook.org/

[179] A. Zeller, S. Just, and K. Greshake, "When Results Are All That Matters: Consequences," 2019. [Online]. Available: https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters.html

[180] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[181] Q. Zhang, J. Wang, and M. Kim, "HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[182] Y. Zhang, C. Pang, S. Nagy, X. Chen, and J. Xu, "Profile-guided System Optimizations for Accelerated Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[183] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing," in *USENIX Security Symposium*, 2022.

[184] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[185] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing," in *USENIX Security Symposium*, 2022.

[186] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets," in *USENIX Security Symposium*, 2023.

[187] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *USENIX Security Symposium*, 2019.

[188] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[189] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: Browser API Fuzzing with Dynamic mod-ref Analysis," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[190] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths," in *USENIX Security Symposium*, 2022.

[191] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[192] X. Zhu and M. Böhme, "Regression Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[193] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[194] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *USENIX Security Symposium*, 2020.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This SoK submission selects 150 papers from 2018-2023 published in top-tier security and software engineering venues for fuzzing research. It then performs a meta-evaluation of each paper's evaluation in terms of experimental design and adherence to generally accepted fuzzing guidelines using Klees et al. as a baseline. In addition, eight papers are subject to artifact evaluation. The conclusions are stark: fuzzing papers continue to fall short of known best practices in conducting rigorous fuzzing research. An updated set of guidelines is then presented.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Other (Reproducibility Study)

A.3. Reasons for Acceptance

1) Fuzzing is an important research area, and understanding whether fuzzing papers hew to best practices intended to maximize the validity and reproducibility of the results is important
2) The paper uses an overall strong review methodology
3) The paper examines a wide range of fuzzing papers over time and across conferences
4) The paper includes an artifact evaluation on a subset of the reviewed fuzzing papers
5) The paper's observations are significant