UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers

Yuwei Li[1], Shouling Ji[1,2], Yuan Chen[1], Sizhuang Liang[4], Wei-Han Lee[5], Yueyao Chen[1], Chenyang Lyu[1], Chunming Wu[1,3], Raheem Beyah[4], Peng Cheng[2,1], Kangjie Lu[6], and Ting Wang[7]

[1]Zhejiang University, [2]Zhejiang University NGICS Platform, [3]Zhejiang Lab, Hangzhou, China,
[4]Georgia Institute of Technology, [5]IBM Research, [6]University of Minnesota, [7]Pennsylvania State University

E-mails: liyuwei@zju.edu.cn, sji@zju.edu.cn, chenyuan@zju.edu.cn, liangsizhuang@gatech.edu, wei-han.lee1@ibm.com, coffee.ki.hy@gmail.com, puppet@zju.edu.cn, wuchunming@zju.edu.cn, rbeyah@ece.gatech.edu, saodiseng@zju.edu.cn, kjlu@umn.edu, inbox.ting@gmail.com.

Abstract

A flurry of fuzzing tools (fuzzers) have been proposed in the literature, aiming at detecting software vulnerabilities effectively and efficiently. To date, it is however still challenging to compare fuzzers due to the inconsistency of the benchmarks, performance metrics, and/or environments for evaluation, which buries the useful insights and thus impedes the discovery of promising fuzzing primitives. In this paper, we design and develop UNIFUZZ, an open-source and metrics-driven platform for assessing fuzzers in a comprehensive and quantitative manner. Specifically, UNIFUZZ to date has incorporated 35 usable fuzzers, a benchmark of 20 real-world programs, and six categories of performance metrics. We first systematically study the usability of existing fuzzers, find and fix a number of flaws, and integrate them into UNIFUZZ. Based on the study, we propose a collection of pragmatic performance metrics to evaluate fuzzers from six complementary perspectives. Using UNIFUZZ, we conduct in-depth evaluations of several prominent fuzzers including AFL [1], AFLFast [2], Angora [3], Honggfuzz [4], MOPT [5], QSYM [6], T-Fuzz [7] and VUzzer64 [8]. We find that none of them outperforms the others across all the target programs, and that using a single metric to assess the performance of a fuzzer may lead to unilateral conclusions, which demonstrates the significance of comprehensive metrics. Moreover, we identify and investigate previously overlooked factors that may significantly affect a fuzzer's performance, including *instrumentation methods* and *crash analysis tools*. Our empirical results show that they are critical to the evaluation of a fuzzer. We hope that our findings can shed light on reliable fuzzing evaluation, so that we can discover promising fuzzing primitives to effectively facilitate fuzzer designs in the future.

Yuwei Li and Shouling Ji are the co-first authors. Shouling Ji and Chunming Wu are the co-corresponding authors.

1 Introduction

Fuzzing is a software-testing technique that detects vulnerabilities by executing target programs with a large amount of abnormal or random test cases. In recent years, a plethora of fuzzing related works have emerged in both industry and academia. In industry, major software vendors such as Google [9] and Microsoft [10] leverage fuzzing techniques to help detect bugs in their products. On the other hand, GitHub [11] hosts more than 2,000 fuzzing related repositories. In academia, over 200 fuzzing related research papers have been published since 2010, according to DBLP [12].

Despite the rapid development of fuzzing techniques, there are several open questions that need to be addressed. (1) *How do these fuzzers perform in practice?* (2) *How to compare different fuzzers under a fair and comprehensive set of performance metrics?* (3) *Which fuzzing primitives or techniques are promising and should be promoted?* However, previous works fail to answer these questions for the following reasons. First, many existing works do not conduct appropriate and sufficient experiments to provide trustworthy results. For instance, it is common to see that insufficient repetitions in the experiments make the results random and unreliable [13]. In addition, many fuzzing works, when comparing their methods with others, directly use previously reported results without re-running the experiments [7, 14], which is unfair as their experimental environments (e.g., CPU, memory) are different. Second, the evaluations of existing fuzzers are often biased due to the lack of uniform benchmarks. The choices of target programs in different fuzzing papers vary widely. Therefore, it is possible that the proposed fuzzers have preference over the selected programs. Third, the existing metrics are not suitable nor comprehensive for evaluating fuzzers. It is inappropriate to only utilize the number of unique crashes to represent a fuzzer's capability of finding bugs, as there is often a huge discrepancy between the number of unique crashes and the number of unique bugs [13]. In addition, most existing fuzzing works do not evaluate the consumption of computing resources of the fuzzers. Therefore, there is an urgent need to conduct comprehensive and pragmatic evaluations for the state-of-the-art fuzzers on a uniform platform.

Conducting comprehensive and pragmatic evaluations of fuzzers entails overcoming multiple important challenges. First, although many fuzzers have been open sourced, their usability in practice is often limited, as reported by recent research [7, 15], which results in reproducibility issues, impeding comparison. Thus, it is necessary to test and enhance fuzzers' usability. Second, the evaluation of fuzzers should be conducted on pragmatic benchmark programs. Existing benchmark programs are not satisfactory [13]. A reliable evaluation of fuzzers thus calls for pragmatic benchmarks. Third, the assessment must be conducted based on a comprehensive set of performance metrics. Nevertheless, existing metrics are insufficient and rough, leading to incomplete assessments. Thus, it is important to augment the performance metrics for comprehensive evaluation.

To address the above challenges, we design and implement UNIFUZZ [16], an open-source, holistic and pragmatic metrics-driven platform for evaluating fuzzers. In summary, we make the following main contributions.

1) An Open-source and Pragmatic Metrics-driven Platform. We design and implement UNIFUZZ, the first open-source platform for evaluating fuzzers in a comprehensive and quantitative manner, which to date has incorporated 35 popular fuzzers, a benchmark suite of 20 real-world programs, and six categories of performance metrics. For each of the 35 fuzzers, we test its usability and provide a Dockerfile for easy installation and deployment. In addition, we find and fix (partially) more than 15 flaws, which have been reported to their developers. For the 20 real-world benchmark programs, UNIFUZZ provides all necessary side information such as software installation and command arguments to ensure their usability. Furthermore, we implement tools in UNIFUZZ to facilitate the crash analysis process including triaging crashes into bugs, matching with the corresponding CVEs, and analyzing the severity of the bugs, etc. Specifically, we develop a *CVE keywords database* that includes the CVEs for the UNIFUZZ benchmark programs, which can significantly reduce the human efforts in CVE matching. We also propose a collection of performance metrics in six categories: *quantity of unique bugs*, *quality of bugs*, *speed of finding bugs*, *stability of finding bugs*, *coverage* and *overhead*, which can be used to assess a fuzzer's performance comprehensively.

2) Extensive Evaluations of Fuzzers. Leveraging UNIFUZZ, we conduct extensive experiments to compare eight prominent coverage-based fuzzers. The experimental results show that no fuzzer outperforms the others on all the tested benchmark programs, which are very different from the conclusions in their papers. This observation reveals that subjectivity and bias may exist in the evaluations of previous fuzzing works. Moreover, the experimental results reflect that using a single metric to evaluate fuzzers may lead to unilateral conclusions, which demonstrates the importance of using comprehensive metrics to evaluate the fuzzers.

3) New Findings and Insights for Future Fuzzing. From the evaluations, we gain important insights and findings for future fuzzing research. For example, we find previously unaccounted factors that can significantly affect the performance of fuzzers, e.g., *instrumentation methods* and *crash analysis tools*. The results demonstrate that even small changes of these factors can have a significant impact on the assessment of fuzzers. Therefore, fuzzing experiments should be conducted in a more rigorous and precise way to provide more reliable results.

2 Motivation of UNIFUZZ

To assess the performance of existing fuzzers and to enlighten the design of new ones, it is crucial to conduct in-depth comparative studies of different fuzzers. Unfortunately, there are many challenges for conducting such comprehensive evaluations on fuzzers as follows, which motivate the design of UNIFUZZ.

Usability Issues of Existing Fuzzers. Whether the existing implementation of fuzzers works in practice is often questionable. First, some fuzzers may be difficult or complicated to be used directly. For instance, Zhu et al. [15] stated that they could not appropriately run Driller [17], T-Fuzz [7] and VUzzer [8]. Second, we find that there are numerous flaws (e.g., incorrect judgment on crash, abnormal behaviors during the fuzzing process) with the implementation of many fuzzers, which may cause negative impacts on their performance. Therefore, it is necessary to test the usability of existing fuzzers and call for more community efforts to enhance fuzzers' usability in practice. We provide the detailed analysis of the flaws of several popular fuzzers on the UNIFUZZ open-source platform [16] due to space limitation.

Lack of Pragmatic Real-World Benchmark Programs. Benchmark programs are fundamental for evaluating the performance of fuzzers, which should be carefully designed such that a fuzzer can be evaluated in a fair manner. Thus, good benchmark programs should have the following characteristics. First, they should have similar features as the real-world programs, and these features include coding styles, sizes and vulnerabilities. In this way, a fuzzer's performance on these benchmark programs can be more indicative. Second, to provide comprehensive evaluations, benchmark programs should exhibit a diversity of functionalities, sizes, vulnerability types, etc. Third, from the perspective of conducting practical assessments on fuzzers' capabilities in discovering bugs, each benchmark program should contain at least one vulnerability that can be found within a reasonable amount of time, which implies two important properties of a pragmatic benchmark. (1) The program should contain at least one bug. Otherwise, it cannot effectively distinguish the capabilities of fuzzers in discovering bugs. (2) The difficulty in discovering a bug should be reasonable[1]. Otherwise, it may cause unaffordable evaluation overhead. For instance, a one-month fuzzing experiment for a single fuzzer on a single program with 30 repetitions requires 21,600 CPU hours, let alone conducting a reliable and comprehensive evaluation with multiple benchmark programs and seed sets [13]. Fourth, the benchmark programs should be easy to use. To this end, the developers should provide rich information of a benchmark program such as installation methods, command arguments, input types. Moreover, it would be better if the developers of the benchmarks can provide methods/tools for automatically analyzing the corresponding crash samples of benchmark programs.

Existing fuzzing benchmark programs can be grouped into two categories: synthetic programs and real-world programs. Typical examples of synthetic benchmarks include LAVA-M [18] and DARPA CGC [19]. Typical examples of real-world programs are `exiv2`, `mp3gain`, etc., which are Linux open-source programs with several vulnerabilities. However, existing benchmark programs, both synthetic and real-world are not satisfactory [13].

The existing synthetic benchmarks usually are small in size, and the artificial bugs are designed and injected following some relatively simple mechanisms. Thus, the developer of a fuzzer may improve its performance by understanding the bug-injecting patterns and the mechanisms, and the evaluation results can be biased. As a result, fuzzers that have good performance on these synthetic benchmark programs may not work well on the real-world programs.

The existing real-world benchmark programs are not satisfactory as well due to the following issues. First, we still lack standard and sufficient real-world benchmark programs, and existing fuzzers are usually evaluated on self-chosen programs, which may cause evaluation bias. Second, the real-world programs are not as convenient as the synthetic programs on validating bugs due to the lack of clear indicators of bug triggering. For example, existing works usually triage crashes and filter vulnerabilities by leveraging different tools such as AddressSanitizer (ASan) [20] and GDB [21]. However, due to their own limitations and inconsistency between different tools, these different crash triage methods may cause bias as well. Moreover, many papers either state that they validate the corresponding CVEs manually [2, 22, 23] or do not mention how they validate the CVEs. Nevertheless, the manual validation process is time-consuming and tedious, which may also cause bias and mistakes. All the issues call for the development of a suite of diverse and pragmatic benchmarks as well as automatic tools to analyze crashes.

Lack of Proper and Comprehensive Performance Metrics. Most previous works usually evaluate fuzzers using the three de facto metrics: the number of unique crashes, the number of unique bugs, and the coverage. However, these metrics alone often fail to fully account for a fuzzer's performance. For instance, solely relying on the number of unique crashes [2, 8, 24] may lead to misleading conclusions, as more unique crashes do not definitely represent more unique bugs [13]. Further, in addition to the number of unique bugs, the quality of bugs is also an important metric that needs to be taken into consideration. For example, when two fuzzers find a similar number of bugs in the same time, it is inappropriate to draw the conclusion that the two fuzzers have similar performance, if the bugs found by one fuzzer are rarer or more dangerous. Finally, overhead is also an important metric. The number of bugs found by fuzzer *A* may be twice as many as those found by fuzzer *B*, but it might be improper to consider fuzzer *A* performs better when it costs hundreds of times of computing resources than fuzzer *B*. Therefore, we need to enhance the metrics, so that they complement each other and provide comprehensive and reliable evaluations for fuzzers.

[1]Note that the difficulty is relative to the state-of-the-art fuzzers. With the development of the fuzzers, the new benchmarks with higher difficulty need to be proposed.

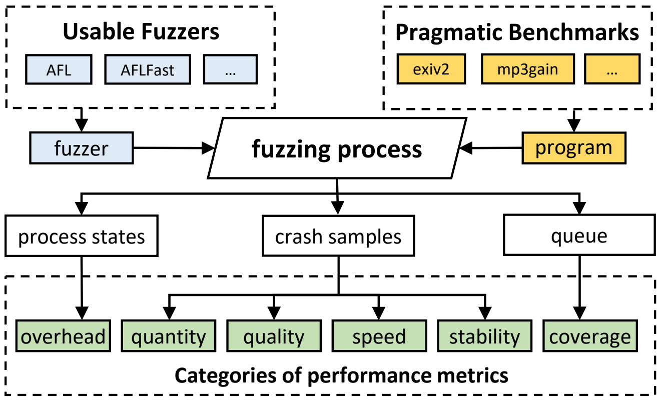


Figure 1: Overview of UNIFUZZ.

3 Design of UNIFUZZ

To address the challenges discussed in Section 2, we design and implement UNIFUZZ, an open-source platform for evaluating fuzzers. Figure 1 presents an overview of UNIFUZZ, which mainly consists of three components: *usable fuzzers*, *pragmatic benchmarks*, and *performance metrics*.

3.1 Usable Fuzzers

UNIFUZZ to date has incorporated 35 usable fuzzers including AFL [1], AFLFast [2], AFLGo [25], AFLPIN [26], AFLSmart [27], Angora [3], CodeAlchemist [28], Driller [17], Domato [29], Dharma [30], Eclipser [31], FairFuzz [32], Fuzzilli [33], Grammarinator [34], Honggfuzz [4], Jsfuzz [35], jsfunfuzz [36], LearnAFL [37], MoonLight [38], MOPT [5], NAUTILUS [39], NEUZZ [40], NEZHA [41], Orthrus [42], Peach [43], PTfuzz [44], QSYM [6], QuickFuzz [45], radamsa [46], slowfuzz [47], Superion [48], T-Fuzz [7], VUzzer [8], VUzzer64 [8] and zzuf [49]. The types of incorporated fuzzers are diverse, including grammar-based, mutation-based, directed and coverage-based fuzzers. Table 1 presents the detailed information of the usable fuzzers incorporated in UNIFUZZ. In order to test the usability of these fuzzers, we manually build and test each of these fuzzers. During this process, we find many design and implementation flaws in these fuzzers. Up to date, we have found more than 15 serious flaws among these fuzzers and reported them to the developers. With our help, some of these flaws have been promptly fixed and released. A more detailed description of these issues is presented in the UNIFUZZ open-source platform [16]. For each fuzzer in Table 1, we also implement a Dockerfile for installing and using it conveniently in a Docker container. We choose to conduct fuzzing experiments in a Docker container for the following reasons. First, compared with conducting fuzzing on a physical machine, Docker is more convenient for resource allocation and isolation, which can provide fair fuzzing evaluations. Second, compared with virtual machines, Docker is lighter-weight and costs less computing resources. Thus, with limited resources, users are able to conduct more fuzzing experiments simultaneously when using Docker. In addition, Docker can be operated and managed more conveniently. In addition to testing the usability of these fuzzers and making them available, we conduct comprehensive evaluations on eight prominent coverage-based fuzzers, with the details presented in Section 4.

Table 1: The fuzzers incorporated in UNIFUZZ.

Fuzzer	Mutation/Generation	Directed/Coverage	Target
AFL [1]	M	C	S/B [1]
AFLFast [2]	M	C	S/B
AFLGo [25]	M	D	S
AFLPIN [26]	M	C	B
AFLSmart [27]	M	C	S/B
Angora [3]	M	C	S/B
CodeAlchemist [28]	G	n.a.	B
Driller [17]	M	C	B
Domato [29]	G	n.a.	B
Dharma [30]	G	n.a.	B
Eclipser [31]	M	C	S
FairFuzz [32]	M	C	S
Fuzzilli [33]	M	C	S
Grammarinator [34]	G	n.a.	B
Honggfuzz [4]	M	C	S
Jsfuzz [35]	M	C	S
jsfunfuzz [36]	G	n.a.	B
LearnAFL [37]	M	C	S
MoonLight [38]	n.a.	n.a.	n.a.
MOPT [5]	M	C	S/B
NAUTILUS [39]	G+M	C	S
NEUZZ [40]	M	C	S
NEZHA [41]	M	C	L [2]
Orthrus [42]	n.a.	n.a.	n.a.
Peach [43]	G	n.a.	B
PTfuzz [44]	M	C	S
QSYM [6]	M	C	B
QuickFuzz [45]	G+M	n.a.	B
radamsa [46]	M	C	B
slowfuzz [47]	M	n.a.	L
Superion [48]	G+M	C	S
T-Fuzz [7]	M	C	S
VUzzer [8]	M	C	B
VUzzer64 [8]	M	C	B
zzuf [49]	M	n.a.	B

[1] S: source code, B: binary.
[2] L: user needs to write libFuzzer code.

3.2 Pragmatic Benchmarks

According to Section 2, pragmatic benchmark programs should have the following properties: (1) *similar to the real-world programs*, including coding styles, sizes, and vulnerabilities, etc. (2) *comprehensive*, which are various in terms of functionalities, sizes, and vulnerability types, etc. (3) *practical*, which means at least one bug should be found in a reasonable amount of time. (4) *conveniently to be used*, which means the users can easily use the benchmark programs and get the evaluation results. Following the above principles, we construct a pragmatic benchmark suite that consists of 20 real-world programs for evaluating fuzzers as shown in Table 2. Specifically, UNIFUZZ provides detailed and comprehensive information of each program including the version, size, installation information, input type and command arguments, etc., to ensure the usability. For each program, UNIFUZZ provides its source code and a Dockerfile for installing and using it. Furthermore, UNIFUZZ provides effective tools to analyze the corresponding crash samples of a target program conveniently. The analyses include but are not limited to: (1) de-duplicating and triaging the crash samples into bugs; (2) matching the crash samples into the corresponding CVEs and (3) analyzing the severity of the bugs triggered by the crash samples. It is worth noting that we do not modify the benchmark programs. As a result, the raw features of the real-world programs are preserved. Instead, we focus on how to select these programs and developing tools for analyzing the experimental results conveniently. Next, we describe the details of program selection and the crash analysis methods.

Programs Selection. In order to select suitable programs, we investigate fuzzing-related papers published on top conferences in information security and software engineering fields to find the real-world programs and the corresponding versions used in their evaluations[2]. Based on the above process, we finally select 20 real-world programs as shown in Table 2. The selected programs cover six functionality types including: image, audio, video, text, binary and network packet processing software. In addition, they cover various types of vulnerabilities including: *heap buffer overflow*, *stack overflow*, *segmentation fault*, *excessive memory allocation*, *global buffer overflow*, *stack buffer overflow*, *memory leak*, *free error*, *float point exception*, *alloc-dealloc mismatch*, *memcpy parameter overlap*, *use-after-free*, etc. Therefore, these programs are able to provide a comprehensive evaluation on the performance of fuzzers.

Triaging Crashes into Unique Bugs. Generally, there are two main approaches for triaging crashes into unique bugs:

[2] If a program is selected with multiple versions, we prefer to choose the one which has more vulnerabilities.

Table 2: The real-world programs of the UniFuzz benchmark. @@ represents the input file.

Type	Program	Version	Arguments
Image	`exiv2`	0.26	@@
	`gdk-pixbuf-pixdata (gdk)`	gdk-pixbuf 2.31.1	@@ /dev/null
	`imginfo`	jasper 2.0.12	-f @@
	`jhead`	3.00	@@
	`tiffsplit`	libtiff 3.9.7	@@
Audio	`lame`	lame 3.99.5	@@ /dev/null
	`mp3gain`	1.5.2-r2	@@
	`wav2swf`	swftools 0.9.2	-o /dev/null @@
Video	`ffmpeg`	4.0.1	(-y -i @@ -c:v \ mpeg4 -c:a copy -f \ mp4 /dev/null)
	`flvmeta`	1.2.1	@@
	`mp42aac`	Bento4 1.5.1-628	@@ /dev/null
Text	`cflow`	1.6	@@
	`infotocap`	ncurses 6.1	-o /dev/null @@
	`jq`	1.5	. @@
	`mujs`	1.0.2	@@
	`pdftotext`	xpdf 4.00	@@ /dev/null
	`sqlite3`	3.8.9	(stdin)
Binary	`nm`	binutils 5279478	(-A -a -l -S -s \ --special-syms \ --synthetic \ --with-symbol-versions \ -D @@)
	`objdump`	binutils 2.28	-S @@
Network	`tcpdump`	4.8.1 + libpcap 1.8.1	-e -vv -nr @@

one is based on analyzing the root cause of the bugs, and the other is based on analyzing the output results. One common implementation of the first approach is to patch the program for each vulnerability based on the analysis of the root cause of the vulnerability [13]. If crash file *a* and crash file *b* both trigger the bug of the target program, but neither does that on the bug-fixed one, they will be regarded as the same bug. Although this approach seems to be able to provide accurate ground truth information of the benchmark programs, the root cause analysis is hard [50, 51] and there are many challenges in implementing this approach in practice. (1) To provide all-side ground truth information of bugs in the program, it is required to access all the patched versions of the target program. (2) Each patched version should only fix one unique bug without overlap. Otherwise, it may cause huge false positives/negatives.

The second approach is usually implemented by analyzing the output information when bugs are triggered. Compared to the first approach, the second approach is more practical in implementation which can provide relatively fair evaluation results. For instance, one commonly used method is leveraging tools such as ASan [20] to produce the stack trace information when a bug of the program is triggered, then the stack hash method [52] can be used to extract *N* stack frames to de-duplicate the bugs. This approach may also cause false positives/negatives when choosing different values of *N*. Nevertheless, how to select the value of *N* to provide results with the lowest false positives/negatives is a difficult research problem which has not been completely solved and is out-of-scope of this paper. As a trade-off and guided by the previous work [13, 52], we select *N* as 3. In addition, as different tools use various methods to detect bugs, relying on a single tool may neglect certain types of bugs. Therefore, to have a more precise detection result, we prioritize the output report produced by ASan [20] and use the output reports produced by other tools such as GDB [21] as a supplement[3].

Matching CVEs. Common Vulnerabilities and Exposures (CVE) [53] provides information of existing vulnerabilities. Most existing fuzzing works evaluate their fuzzers' capability in finding bugs by leveraging CVE information [2,7,24]. Thus, it is important to figure out what and how many known/new CVEs (CVE vulnerabilities) are discovered by a fuzzer. However, matching crash samples with the corresponding CVEs is time-consuming and tedious for the following reasons. First, the description of each CVE is written in natural language without a well-defined structure. Thus, it is difficult to extract the key information (e.g., vulnerable function names) from the description directly. Second, although the references of each CVE may provide additional information such as the PoC (Proof-of-the-Concept) file that triggers a CVE and the output report generated by crash analysis tools, such information is usually incomplete or missing [54], which makes CVE matching even harder. Moreover, the references are also unstructured. Thus, human efforts are needed to figure out what content a reference link represents (e.g., the download link of a PoC file or the bug reports). Third, as different tools may be leveraged to obtain the output report, it is difficult to match with different output reports directly.

In order to reduce the human efforts in matching CVEs, we construct a *CVE keywords database* that includes the key information of the CVEs related to the UniFuzz benchmark programs. This database can be leveraged to match the crashes with the corresponding CVEs conveniently. Compared with the information provided on the official CVE website [53], the information in *CVE keywords database* is better structured. In *CVE keywords database*, each benchmark program has a CVE table. Each entry of the table represents the information of a CVE. The primary key of each entry is the CVE ID, and the other attributes are the pivotal information of this CVE, including vulnerability type, vulnerable functions, vulnerable files, stack trace, the tool that generates the stack trace, etc. Leveraging the *CVE keywords database*, we implement a method that can conveniently generate the initial matching results. Based on the *CVE keywords database*, the CVE matching process is as follows. (1) Compile the program with the corresponding tool (e.g., ASan) and execute the binary with the crash to obtain the output report. (2) Extract the necessary information from the output report, which includes the stack trace, vulnerability type, vulnerable functions, vulnerable file names, etc. (3) Match the extracted information with the CVE table of the program in *CVE keywords database* and report the initial matching CVEs sorted by the number

[3]We only use GDB to detect the bugs which cannot be found by ASan. Therefore, a crash sample can only be mapped with one unique bug at most.

of matched keywords. (4) Check the initial matching results manually to obtain the final matching results. Note that the official CVE website [53] has flaws and mistakes such as incomplete information [54] and overlapped CVEs [55]. On the other hand, it is possible that a 0-day vulnerability is found. Thus, in this case, it is necessary to conduct the last step to make the matching result more precise and accurate.

Discussion on the Ground Truth. In general, it is hard to obtain the complete ground truth bugs for both the synthetic and the real-world programs due to the nature of bugs. For the synthetic benchmarks, whether the other parts (except for the injected bugs) have bugs is unknown, which makes it hard to obtain the complete ground truth. Similar for a real-world program, except for the already known bugs, whether it has new bugs is unknown. Even though, we try our best to provide the information as accurate as possible for the benchmark in the following manners. First, to mitigate the incompleteness issue, we collect as many crash samples as possible to detect the possible bugs in the benchmark programs. Second, we use multiple tools to detect the bugs. In addition to the eight coverage-based fuzzers, we combine three static analysis tools (Flawfinder [56], RATS [57], Clang Static Analyzer [58]) with the directed fuzzer, AFLGo [25], to find more bugs of the UniFuzz benchmark. The details of the detection results are presented in the UniFuzz open-source platform [16], due to space limitation. Third, we analyze the bugs with multiple tools (i.e., ASan and GDB) to reduce the impact caused by the limitations of a single tool.

3.3 Performance Metrics

To address the problem of lacking comprehensive and pragmatic performance metrics, we systematically study the performance metrics of the existing fuzzing papers, summarize and propose a set of metrics, which can be classified into six categories: *quantity of unique bugs*, *quality of bugs*, *speed of finding bugs*, *stability of finding bugs*, *coverage* and *overhead*. Each category represents a property of a fuzzer's performance, and each property can be evaluated by many concrete metrics which are expandable. For example, when evaluating *quantity of unique bugs*, we can leverage many concrete mathematical metrics such as p value, $\hat{A}_{12}$ score [59]. In the following, we present concrete metrics for each category as suggestions to use in practical evaluations.

Quantity of Unique Bugs. As there exists randomness with a fuzzing process, a robust fuzzing experiment has to be repeated for multiple times to provide a more reliable result. Therefore, the quantitative metrics of unique bugs are based on statistical methods. We focus on two important questions: (1) *how many times should a fuzzing experiment be repeated?* and (2) *what statistical metrics can provide reliable results?* There are different opinions about these questions. For question (1), Klees et al. [13] suggested conducting each fuzz testing for 30 repetitions. For question (2), Klees et al. [13] stated that general statistical metrics such as *mean*, *median* and *variance* may result in misleading conclusions. Besides, they highly recommended to use statistical tests to calculate the p value to determine whether there is a statistically significant difference between the two fuzzers' performance. Specifically, they suggested using the Mann-Whitney U test as the statistical test method instead of other methods. The reason is that the Mann-Whitney U test is non-parametric which makes no assumption on the distribution of the population (as the distribution of fuzzing results, e.g., the number of unique bugs in all repetitions is still unknown), whereas some other methods are stricter. For instance, t-test assumes that the two populations must obey normal distributions and have the same variance. However, there are some different viewpoints about statistical tests. For example, Nuzzo [60] pointed out that the p value is not as reliable as many scientists assume, and Wasserstein et al. [61] suggested that we should not draw the conclusion that there is a statistically significant difference when $p < 0.05$, or there is not a statistically significant difference when $p > 0.05$.

Based on the above discussion and our experience, our suggestions for the two questions are as follows. For the statistical metrics, as no single metric is perfect, it is better to report a set of statistical metrics such as *mean*, *median*, the p value, etc. In addition to measuring whether fuzzer A performs better than fuzzer B, it is also important to measure how much fuzzer A performs better than fuzzer B. To quantify the *extent* of the difference between two fuzzers, it is recommended to use the Vargha and Delaney $\hat{A}_{12}$ score [59] to show the probability that fuzzer A performs better than fuzzer B. For the number of repetitions, which is related to the selected statistical metrics. For instance, the number of repetitions should be larger than 20 when using the Mann-Whitney U test [62]. In addition, it is necessary to conduct deeper research on these two questions in the future.

Quality of Bugs. We define the quality of bugs from the perspective of evaluating fuzzers' performance. That is, the quality of bugs should reflect not only the severity of bugs, but also the effectiveness of fuzzers in finding rare bugs. A fuzzer which can find more high quality bugs should be considered as better. Specifically, we measure the quality of a bug from two main aspects: (1) *whether a bug has a higher level of severity* and (2) *whether a bug is harder to be found*. For aspect (1), we can leverage analysis tools to measure the severity of a bug. For example, `Exploitable` [63] is a GDB extension that classifies Linux application bugs by severity. Moreover, we can map a bug with its corresponding CVE and assess its severity by the Common Vulnerability Score System (CVSS) score [64] of the CVE. For aspect (2), a bug which is hard to be found usually has the following characteristics: it can be found by few fuzzers or it is mapped with a small amount of crash samples.

Speed of Finding Bugs. Finding bugs quickly and efficiently is important, especially when the time budget is limited. We can measure the speed of finding bugs using the following two approaches. First, for all the bugs of a program, we can draw the cumulative curve of the number of all the detected unique bugs within a pre-defined time. A higher slope of the curve means a higher speed to find bugs, which is a relatively qualitative metric. Second, for a specific bug, we can record the *time-to-exposure* (TTE) metric to measure the time that it is found by a fuzzer for the first time, which is a relatively quantitative metric.

Stability of Finding Bugs. Stability is another important metric. A fuzzer with higher stability is more reliable and practical. We can quantify the stability of a fuzzer in the following manner. First, we can calculate the *relative standard deviation* (RSD) of the number of the found unique bugs among all the repetitions. Lower RSD means better stability. Second, for a specific bug, we can calculate the number of times that a fuzzer can find it successfully. Higher success rate represents that a fuzzer has better stability.

Coverage. Coverage metrics are used to measure a fuzzer's capability of exploring paths, which are also significant in quantifying the capability of a fuzzer. As the vulnerable code usually takes a tiny fraction of the entire code, only considering the number of bugs may not be able to distinguish the fuzzers' capability in exploring paths. Coverage metrics can be measured with different granularity levels such as function, basic block, edge and line coverage, etc.

Overhead. The overhead metrics instead aim to quantify how many computing resources a fuzzer costs during the fuzzing process, which are also important. For instance, we may determine that a fuzzer performs well when we only consider how many bugs it finds, but the determination may be misleading if it costs much more computing resources than others. This metric is instructive for users who have limited computing resources or need to run multiple fuzzers in parallel. The overhead of a fuzzer can be measured by the following concrete metrics: *CPU utilization*, *memory consumption*, and *the amount of disk read/write*, etc.

4 Evaluations of the State-of-the-art Fuzzers

Leveraging UniFuzz, we conduct extensive experiments on the state-of-the-art fuzzers, and comprehensively compare them in terms of the six categories of performance metrics. Following the guidelines in [13], we conduct fuzz testing for 24 hours, with 30 repetitions.

4.1 Experiment Settings

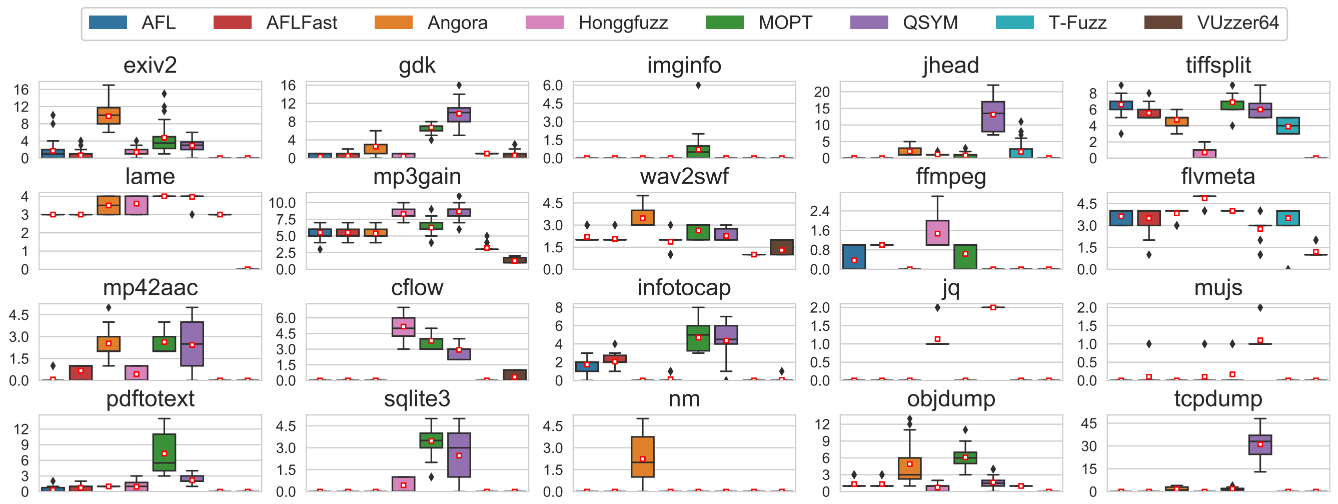
Fuzzers. In our evaluations, we select eight state-of-the-art coverage-based fuzzers from UniFuzz, including AFL [1], AFLFast [2], Angora [3], Honggfuzz [4], MOPT [5], QSYM [6], T-Fuzz [7] and VUzzer64 [8]. The reasons for selecting these fuzzers are as follows. First, they are prominent fuzzers at the time of writing this paper. AFL and Honggfuzz are proposed in industry which have been widely applied in practice. The other six fuzzers are presented at top security conference in recent years, which represent advanced fuzzing techniques in academia. Second, although there are other advanced fuzzers such as CollAFL [24], they are not open-source making them difficult to be evaluated. Third, the eight selected fuzzers have better scalability than others which can be used to test most of the programs. In comparison, other fuzzers such as QuickFuzz [45] can only generate limited types of test cases[4]. Thus, they can only be tested on a limited number of benchmark programs. Fourth, as it is not appropriate to compare between fuzzers of different types, here we only include coverage-based fuzzers to provide comparable evaluations. For other fuzzers incorporated in UniFuzz, we mainly focus on testing their usability and making them available. In addition, to make the evaluations fairer and more comparable, we make necessary modifications on several fuzzers. For Angora, we change the input size limitation from 15 KB to 1 MB, to make it fairly comparable with the other fuzzers. For VUzzer64, we fix the issues (#10, #11, #12 and #14) of its repository as these flaws are serious. In addition, we modify the value of the variable `GENNUM`, which determines the number of generations, from 1,000 to 1,000,000 to make VUzzer64 fuzz for a longer time. For T-Fuzz, we fix its "naming" flaw. All the above modifications are to make the evaluation fairer. For the other fuzzers, we keep them the same as the original design.

Programs. We leverage the 20 real-world programs (Table 2) provided by UniFuzz to evaluate the selected fuzzers. In addition, we also use LAVA-M to explore the gap between the synthetic and the real-world programs. Each benchmark program is compiled according to each fuzzer's requirements in instrumentation or compilation. When validating the found bugs, the programs are compiled with ASan and GDB, which is same for all the fuzzers.

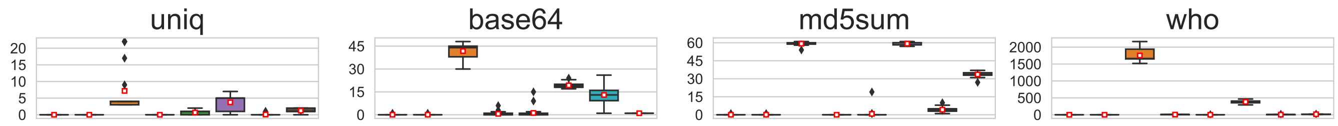
Initial Seeds. Following common practice in fuzzing, we utilize the same initial seeds for the same benchmark program. For the UniFuzz benchmark programs, we select the initial seeds by the following process. First, we collect some seeds with the corresponding file format from the Internet. Second, we exclude the seeds which do not satisfy a fuzzer's requirement (e.g., AFL requires the size of a seed be less than 1 MB). Then, for the rest ones, we randomly select 100 seeds for each program. For programs of LAVA-M, we use the default seed set provided by LAVA-M.

Environments. We conduct all the experiments on 5 servers with the same equipment: 20 Intel Xeon E5-2650 v4 CPU cores with 2.20 GHz, 64-bit Ubuntu 16.04 LTS. For each fuzzer, we assign one CPU core, 2 GB RAM, and 1 GB swap space. If a fuzzer cannot run successfully with 2 GB memory, we increase the memory limit to 8 GB, with the same

[4]For instance, as QuickFuzz cannot generate .mp3 file, we cannot test QuickFuzz on program `mp3gain`.



(a) Real-world programs



(b) LAVA-M programs

Figure 2: The number of unique bugs detected by fuzzers.

resources allocated for all fuzzers in the same fuzzing experiment[5]. We run each fuzzing experiment in an independent Docker container.

Next we present the main evaluation results based on the six categories of performance metrics, and we provide more detailed evaluation results and datasets on the UniFuzz open-source platform [16]. Note that T-Fuzz costs too much memory when fuzzing program `ffmpeg`, and VUzzer64 cannot test program `sqlite3` because it does not support input from `stdin`. Thus, we do not include the results of the above cases.

4.2 Quantity of Unique Bugs

The main objective of this subsection is to figure out *which fuzzer can find more unique bugs.* As described in Section 3.2, we leverage the output report produced by ASan [20] to extract the top three functions in the stack trace as a triple to de-duplicate bugs. The bugs that have the different triples and vulnerability types are considered as unique. For bugs that cannot be detected by ASan [20], we further leverage the output report produced by other tools such as GDB [21] as a supplement[6], as we find that there exist some bugs, e.g., *float point exception* bugs which can be detected by GDB, while not ASan.

Number of Unique Bugs. We visualize the number of unique bugs found by each fuzzer on the real-world programs of the UniFuzz benchmark and LAVA-M in 30 repetitions in Figure 2(a) and Figure 2(b) respectively. From these figures, we have the following observations. (1) No fuzzer outperforms others on all the programs. (2) For the 20 real-world programs, QSYM performs the best on five programs (`gdk`, `jhead`, `lame`, `mujs`, `tcpdump`). Angora performs the best on three programs (`exiv2`, `wav2swf`, `nm`). Honggfuzz performs the best on three programs (`ffmpeg`, `flvmeta`, `cflow`). MOPT performs the best on three programs (`imginfo`, `lame`, `pdftotext`). AFL performs the best on program `tiffsplit`. AFLFast, T-Fuzz and VUzzer64 fail to achieve the best performance on any target program. (3) For the programs of LAVA-M, Angora performs the best among the selected fuzzers, while QSYM only achieves similar performance as Angora on program `md5sum`.

Statistical Results. Here we present the statistical results of the number of unique bugs found by the fuzzers in 30 repetitions. Due to space limitation, we only present two statistical results: p value and $\hat{A}_{12}$ score. We defer the other statistical

[5]We do not start all experiments at 8 GB due to resource limitations. As the same resources are allocated for all fuzzers in the same fuzzing experiment, the evaluations are fair.

[6]When verifying the crashes found by T-Fuzz, if the crashes cannot make the original program crash, we then use CrashAnalyzer (provided by the developers of T-Fuzz) to generate new crashes and verify them. However, we find that whether using CrashAnalyzer has no impact on our experiment results. For the tested 20 programs, T-Fuzz does not generate transformed binaries for 17 programs (the reason is that T-Fuzz will not generate transformed binaries if there is no "stuck" state), which means there is no need to use CrashAnalyzer. For the three programs (`jhead, flvmeta and wav2swf`), the CrashAnalyzer does not produce new crashes.

Table 3: The p value and the $\hat{A}_{12}$ score of the number of unique bugs in 30 repetitions with AFL as the baseline.

		AFL	**AFLFast**		**Angora**		**Honggfuzz**		**MOPT**		**QSYM**		**T-Fuzz**		**VUzzer64**	
		Avg	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$	p-val	$\hat{A}_{12}$
Real-world Programs	`exiv2`	1.7	0.01	0.32	**< 0.01**	**0.97**	0.27	0.55	**< 0.01**	**0.83**	**< 0.01**	**0.78**	< 0.01	0.17	< 0.01	0.17
	`gdk`	0.3	0.11	0.58	**< 0.01**	**0.92**	0.21	0.55	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.85**	0.18	0.56
	`imginfo`	0	1.0	0.5	1.0	0.5	1.0	0.5	**< 0.01**	**0.75**	1.0	0.5	1.0	0.5	1.0	0.5
	`jhead`	0	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.78**	**< 0.01**	**1.0**	**< 0.01**	0.7	1.0	0.5
	`tiffsplit`	6.6	< 0.01	0.25	< 0.01	0.12	< 0.01	0	0.26	0.55	0.01	0.33	< 0.01	0.06	< 0.01	0
	`lame`	3	1.0	0.5	**< 0.01**	**0.75**	**< 0.01**	**0.8**	**< 0.01**	**1.0**	**< 0.01**	**0.98**	1.0	0.5	< 0.01	0
	`mp3gain`	5.5	0.28	0.46	0.13	0.42	**< 0.01**	**1.0**	**< 0.01**	**0.71**	**< 0.01**	**0.98**	< 0.01	0.03	< 0.01	0
	`wav2swf`	2.2	0.07	0.43	**< 0.01**	**0.94**	< 0.01	0.35	**< 0.01**	**0.72**	0.28	0.53	< 0.01	0	< 0.01	0.12
	`ffmpeg`	0.37	**< 0.01**	**0.82**	< 0.01	0.32	**< 0.01**	**0.87**	**0.02**	0.63	< 0.01	0.32	n.a.	n.a.	< 0.01	0.32
	`flvmeta`	3.63	0.39	0.48	**0.04**	0.6	**< 0.01**	**0.96**	**< 0.01**	0.68	< 0.01	0.21	0.37	0.52	< 0.01	0
	`mp42aac`	0.07	**< 0.01**	**0.8**	**< 0.01**	**1.0**	**< 0.01**	0.68	**< 0.01**	**1.0**	**< 0.01**	**0.9**	0.08	0.47	0.08	0.47
	`cflow`	0	1.0	0.5	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	0.67
	`infotocap`	1.73	0.08	0.6	< 0.01	0.03	< 0.01	0.06	**< 0.01**	**0.97**	**< 0.01**	**0.85**	< 0.01	0.03	< 0.01	0.05
	`jq`	0	1.0	0.5	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
	`mujs`	0	**0.04**	0.55	1.0	0.5	**0.04**	0.55	**0.01**	0.58	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
	`pdftotext`	0.3	**< 0.01**	0.69	**< 0.01**	**0.85**	**< 0.01**	**0.71**	**< 0.01**	**1.0**	**< 0.01**	**1.0**	< 0.01	0.37	< 0.01	0.37
	`sqlite3`	0	1.0	0.5	1.0	0.5	**< 0.01**	**0.72**	**< 0.01**	**1.0**	**< 0.01**	**0.93**	1.0	0.5	n.a.	n.a.
	`nm`	0	1.0	0.5	**< 0.01**	**0.93**	1.0	0.5	1.0	0.5	1.0	0.5	1.0	0.5	1.0	0.5
	`objdump`	1.33	1.0	0.5	**< 0.01**	**0.89**	< 0.01	0.31	**< 0.01**	**0.99**	0.12	0.58	0.01	0.42	< 0.01	0
	`tcpdump`	0	1.0	0.5	**< 0.01**	**0.88**	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	1.0	0.5	1.0	0.5
LAVA-M	`uniq`	0	1.0	0.5	**< 0.01**	**1.0**	1.0	0.5	**< 0.01**	**0.77**	**< 0.01**	**0.95**	0.08	0.53	**< 0.01**	**0.92**
	`base64`	0.03	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	0.64	**< 0.01**	0.69	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**0.98**
	`md5sum`	0.03	1.0	0.5	**< 0.01**	**1.0**	0.17	0.48	0.08	0.55	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**
	`who`	0	1.0	0.5	**< 0.01**	**1.0**	**< 0.01**	**1.0**	0.08	0.53	**< 0.01**	**1.0**	**< 0.01**	**1.0**	**< 0.01**	**1.0**

results such as mean and median values on the UNIFUZZ open-source platform [16]. The p value aims to quantify whether there is a significant difference between two populations (corresponding to two fuzzers in our setting). $\hat{A}_{12}$ score is used to measure the effect size (i.e., the probability that one fuzzer is better than the other according to all the repetitions). Here, we use AFL as the baseline fuzzer following most previous works [13]. Specifically, we leverage the Mann-Whitney U test to calculate the p value, and we consider $p < 0.05$ as an indicator that there exists a significant difference. For $\hat{A}_{12}$ score, we consider $\hat{A}_{12} \geq 0.71$ as an indicator that there is a large effect size [59]. Table 3 shows the p value and the $\hat{A}_{12}$ score of the number of unique bugs in 30 repetitions. From Table 3, we have the following observations. (1) None of the remaining seven fuzzers outperforms AFL significantly on all the real-world programs. Nevertheless, there exists fuzzers (Angora, QSYM and VUzzer64) that outperform AFL significantly on all four programs of LAVA-M. (2) Based on the results of p value, MOPT performs significantly better than AFL on 17 real-world programs, which is the most among the seven fuzzers. QSYM, Angora and Honggfuzz perform significantly better than AFL on 13, 11 and 11 real-world programs respectively. However, AFLFast only performs significantly better than AFL on 4 real-world programs. Even worse, T-Fuzz and VUzzer64 do not outperform AFL significantly on any real-world program. (3) Considering the $\hat{A}_{12}$ score metric, the fuzzers have the similar performance compared with the p value metric. The fuzzers that have large effect size ($\hat{A}_{12} \geq 0.71$) all outperform significantly (p value < 0.05) than AFL, but not vice versa. For instance, AFLFast outperforms significantly than AFL on `mujs` ($p = 0.04$), but the effect size ($\hat{A}_{12} = 0.55$) is not large.

Table 4: The number of CVEs with high severity.

	AFL	**AFLFast**	**Angora**	**Honggfuzz**	**MOPT**	**QSYM**	**T-Fuzz**	**VUzzer64**
`exiv2`	1	1	**4**	2	3	2	0	0
`gdk`	0	0	0	0	0	0	0	0
`imginfo`	0	0	0	0	**6**	0	0	0
`jhead`	0	0	**1**	0	0	**1**	**1**	0
`tiffsplit`	**3**	**3**	**3**	1	**3**	2	**3**	0
`lame`	1	1	**2**	**2**	**2**	**2**	1	0
`mp3gain`	3	3	3	**5**	**5**	**5**	2	1
`wav2swf`	**1**	**1**	**1**	**1**	**1**	**1**	0	**1**
`ffmpeg`	0	0	0	**1**	0	0	n.a.	0
`flvmeta`	0	0	0	0	0	0	0	0
`mp42aac`	0	0	0	0	1	**3**	0	0
`cflow`	0	0	0	0	0	0	0	0
`infotocap`	1	1	0	1	**2**	**2**	0	0
`jq`	0	0	0	**1**	0	**1**	0	0
`mujs`	0	0	0	0	0	**1**	0	0
`pdftotext`	2	2	1	2	**4**	2	1	0
`sqlite3`	0	0	0	0	0	0	0	n.a.
`nm`	0	0	0	0	0	0	0	0
`objdump`	0	0	0	0	0	0	0	0
`tcpdump`	0	0	4	0	3	**57**	0	0

Table 5: The average number of unique *EXPLOITABLE* bugs.

	AFL	**AFLFast**	**Angora**	**Honggfuzz**	**MOPT**	**QSYM**	**T-Fuzz**	**VUzzer64**
`exiv2`	1.3	0.5	**6.7**	0.1	4.9	0.3	0	0
`gdk`	0.0	0.3	1.2	0	**7.9**	2.3	4.7	0.5
`imginfo`	0	0	0	0	0	**0.03**	0	0
`jhead`	0	0	**0.2**	0	0	**0.2**	**0.2**	0
`tiffsplit`	0.7	**0.8**	0.2	0	**0.8**	0	0.7	0
`lame`	1.0	1.0	5.8	3.4	**9.2**	3.1	1.0	0
`mp3gain`	0.1	0.1	0	**2.0**	0.8	0.8	0	0
`wav2swf`	3.0	3.1	3.0	0.1	**10.0**	3.0	0	0.3
`ffmpeg`	0	0	0	**0.1**	0	0	n.a.	0
`flvmeta`	0.2	0.3	0.1	**0.6**	0.5	0.1	0.1	0
`mp42aac`	0	0	0.0	0	0	**0.5**	0	0
`cflow`	0	0	0	**0.8**	0.2	0	0	0
`infotocap`	0	0	0	0	**0.2**	0	0	0
`jq`	0	0	0	0	0	0	0	0
`mujs`	0	0	0	**0.1**	0	0	0	0
`pdftotext`	0.3	0.7	1.0	0.7	**7.2**	5.0	0	0
`sqlite3`	0	0	0	0	**3.1**	0	0	n.a.
`nm`	0	0	**4.8**	0	0	0	0	0
`objdump`	0.3	0.4	1.2	0.5	**3.2**	0	0.2	0
`tcpdump`	0	0	0	0	**0.3**	0	0	0

4.3 The Quality of Bugs

As explained in Section 3.3, we define bug quality based on their severity and the rareness.

4.3.1 Severity of Bugs

The severity of bugs can be quantified by the CVE CVSS score [64] and the results of `Exploitable` [63].

CVE CVSS. CVSS [64] provides a numerical score for each CVE to quantify its severity. A CVE is considered as highly severe when the score is greater than or equal to 7.0. We leverage the *CVE keywords database* and the matching method in Section 3.2 to get the initial CVE matching results. Then, we manually check the initial results to obtain the final matching results. Further, we associate each CVE with the corresponding CVSS score. During the CVE matching process, we also find six new CVEs: *CVE-2019-17450, CVE-2019-17451, CVE-2019-17594, CVE-2019-17595, CVE-2019-18359 and CVE-2019-19035*. Table 4 shows the number of CVEs with high severity found by the fuzzers, and we provide more detailed information of the found CVEs on the UniFuzz open-source platform [16], including the concrete CVSS score and the vulnerability type. As presented in Table 4, the fuzzers have preference on specific programs in discovering highly severe CVEs. For instance, QSYM discovers 57 highly severe CVEs on `tcpdump`, while Honggfuzz cannot discover any one. However, for `ffmpeg`, Honggfuzz can discover one highly severe CVE, while the remaining fuzzers (including QSYM) cannot find any one. Moreover, it is interesting to note that AFL and AFLFast are comparable with respect to the number of discovered highly severe CVEs on each program.

Results of `Exploitable`. `Exploitable` [63] is a GDB extension that uses a heuristic algorithm to assess the exploitability of a crash, which can be classified into four categories: *EXPLOITABLE*, *PROBABLY_EXPLOITABLE*, *PROBABLY_NOT_EXPLOITABLE* and *UNKNOWN*. Specifically, we de-duplicate the number of bugs of each category by the hash value produced by `Exploitable`. Table 5 presents the average number of unique bugs that are classified as *EXPLOITABLE*. As presented in Table 5, MOPT outperforms the other fuzzers on 9 programs in detecting *EXPLOITABLE* bugs. Angora, Honggfuzz and QSYM achieve the best performance on 3, 5 and 3 programs, respectively. Nevertheless, VUzzer64 does not perform well as it can only detect *EXPLOITABLE* bugs on 2 programs.

4.3.2 Rareness of Bugs

It is intuitive that a bug that can be found by fewer fuzzers is relatively harder to be found (e.g., is located in deeper path or has more complicated path constraints). Here, we call a bug that can be found by only one fuzzer a *rare bug*[7]. Correspondingly, a fuzzer that can find more unique *rare bugs* is relatively more powerful. Table 6 shows the number of unique *rare bugs* discovered by the evaluated fuzzers. For all the real-world programs, QSYM achieves the best performance by discovering 262 unique *rare bugs*. MOPT achieves the second best performance and discovers 90 unique *rare bugs*. Angora finds 56 unique *rare bugs* in total. Nevertheless, AFLFast only detects *rare bugs* on two programs. AFL, T-Fuzz and VUzzer64 only detect *rare bugs* on one program. It is worth noting that fuzzers also have preference on target programs in discovering *rare bugs*. For instance, QSYM discovers 204 unique *rare bugs* on `tcpdump`, while in comparison Angora only discovers one *rare bug* and the other fuzzers fail to find any *rare bug*. For `nm`, Angora can discover 25 unique *rare bugs*, while the remaining fuzzers including QSYM fail to discover any *rare bug*.

Table 6: The number of discovered unique *rare bugs*.

	AFL	AFLFast	Angora	Honggfuzz	MOPT	QSYM	T-Fuzz	VUzzer64
`exiv2`	8	1	17	0	**22**	0	0	0
`gdk`	0	0	2	0	1	**13**	0	1
`imginfo`	0	0	0	0	**7**	0	0	0
`jhead`	0	0	1	0	0	**15**	2	0
`tiffsplit`	0	0	0	0	**3**	2	0	0
`lame`	0	0	0	0	0	0	0	0
`mp3gain`	0	0	0	**1**	0	0	0	0
`wav2swf`	0	0	**2**	0	0	0	0	0
`ffmpeg`	0	0	0	**3**	0	0	n.a.	0
`flvmeta`	0	0	0	**1**	0	0	0	0
`mp42aac`	0	0	2	0	**8**	7	0	0
`cflow`	0	0	0	**2**	**2**	0	0	0
`infotocap`	0	0	0	0	3	**4**	0	0
`jq`	0	0	0	0	0	0	0	0
`mujs`	0	0	0	**2**	0	**2**	0	0
`pdftotext`	0	0	0	0	**35**	7	0	0
`sqlite3`	0	0	0	0	1	**3**	0	n.a.
`nm`	0	0	**25**	0	0	0	0	0
`objdump`	0	1	**6**	0	4	5	0	0
`tcpdump`	0	0	1	0	4	**204**	0	0
Total	8	2	56	9	90	**262**	2	1

4.4 Speed of Finding Bugs

Figure 3 presents the average number of unique bugs found over time in 30 repetitions, where we can see the fuzzers' speed of finding bugs. First, one intuitive observation is that no fuzzer wins the others on all the programs on this metric. Second, the comparisons among fuzzers' performance may get reverse over time. For instance, MOPT finds less unique bugs than QSYM on program `sqlite3` in the early time, but it finds more unique bugs than QSYM after 10 hours. Third, although some fuzzers find the similar number of unique bugs, their speeds of finding bugs are different. For instance, Angora, MOPT and QSYM find a similar number of unique bugs on program `mp42aac` (2.5, 2.6 and 2.4 unique bugs in average, respectively) within 24 hours, while MOPT finds the bugs more quickly than Angora and QSYM. This observation also indicates the importance of the speed metric, as only leveraging the number of unique bugs may overlook the difference of fuzzers in speed.

[7]The value of this metric is not an absolute value such as the number of unique bugs, while providing a relative measure that depends on the compared fuzzers.

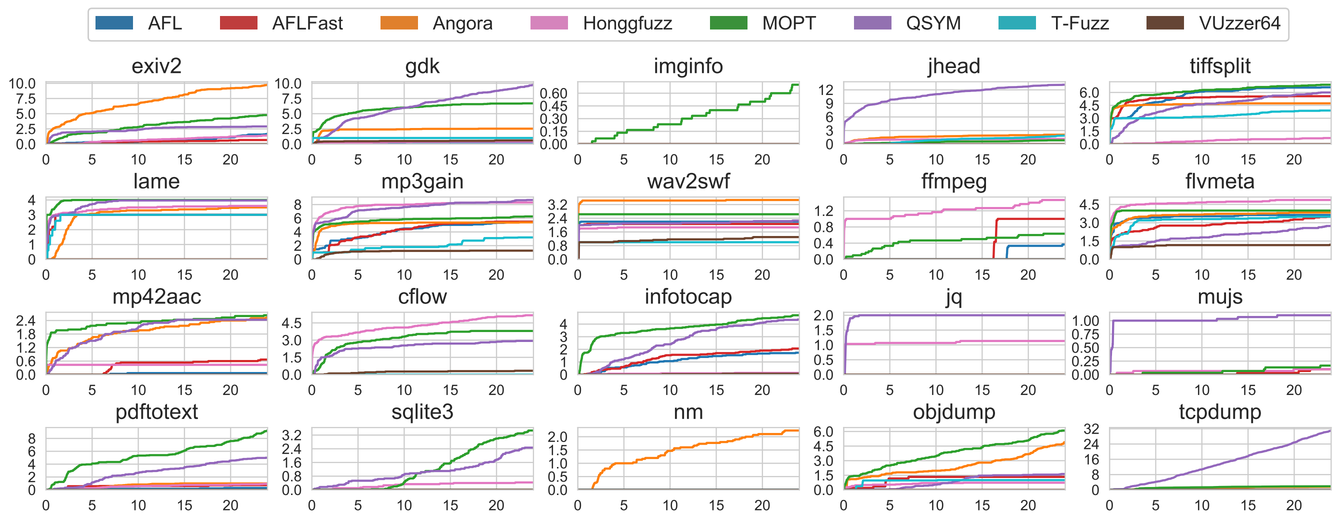


Figure 3: The average number of unique bugs found over time in 30 repetitions.

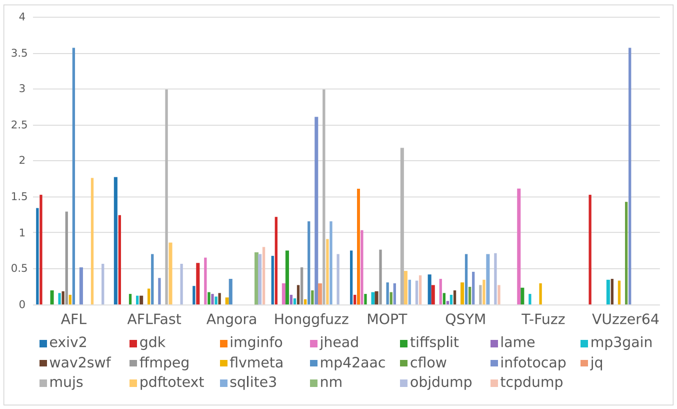


Figure 4: The RSD of the number of unique bugs.

4.5 Stability of Finding Bugs

Figure 4 presents the *relative standard deviation* (RSD) of the number of unique bugs in all the 30 repetitions, where a lower RSD represents better stability of a fuzzer. As depicted in Figure 4, first, all the fuzzers are not always stable in finding bugs, which reflects the randomness of fuzzing and the importance of repetitions. Second, among the seven fuzzers, Angora and T-Fuzz achieve lower RSD, while AFL and Honggfuzz achieve higher RSD. Third, the stability of a fuzzer varies with different programs. For instance, AFL has better stability on several programs such as `tiffsplit`, `mp3gain`, as compared to that on `mp42aac`. It needs to be noted that the stability of finding bugs metrics are auxiliary to the quantity of unique bugs metrics, as finding more bugs are more important than finding less bugs stably.

4.6 Coverage

Existing fuzzers track the coverage of a target program with different manners and granularities. For instance, AFL [1] leverages compile-time instrumentation and bitmap to track edge coverage. Honggfuzz [4] leverages SanitizerCoverage instrumentation [65] to track basic block coverage. In order to fairly compare these fuzzers' capability of finding paths, it is necessary to design a uniform method (with the same instrumentation method and under the same granularity) to track the coverage for different fuzzers. One intuitive method is to save all the test cases executed by the fuzzers, then calculate their coverage with the same instrumented binary program. Nevertheless, this method is impractical as the number of executed test cases is tremendous. To strike a balance between precision and efficiency, we develop an efficient method to track the coverage by only considering the test cases that improve the coverage. Specifically, we save all the test cases that increase the coverage during the fuzzing process, then we leverage afl-cov [66] to calculate the line coverage of each program with the saved test cases. Figure 6 shows the results of line coverage, from which we observe that no fuzzer stably achieves higher coverage than the others. By comparing Figure 2(a) and Figure 6, we observe that *higher coverage does not necessarily mean more unique bugs*. For instance, MOPT achieves the highest coverage on `tcpdump` among all the fuzzers while QSYM discovers the most unique bugs on `tcpdump`. To further explore the relationship between the number of unique bugs and line coverage, we calculate the *Spearman correlation coefficient* r_s between them, which is a non-parametric measure of correlation between two variables and $r_s \in [-1, +1]$. A positive r_s means that the two variables are positively correlated and vice versa. Figure 5 presents the value of r_s between the number of unique bugs and line coverage, where we observe that most of them are less than 0.60, which means that the correlation between the number of unique bugs and the line coverage is not strong.

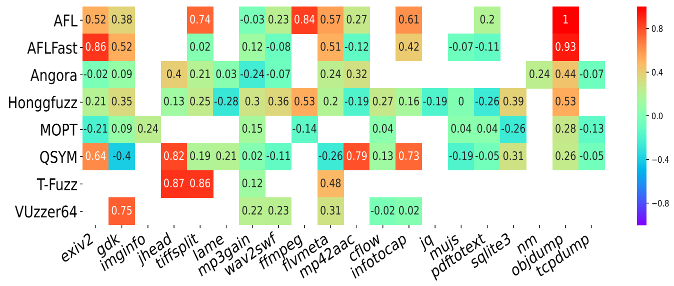


Figure 5: The Spearman's correlation coefficient r_s between the number of unique bugs and line coverage.

Table 7: The memory consumption (MB) of each fuzzer.

	AFL		AFLFast		Angora		Honggfuzz		MOPT		QSYM		T-Fuzz		VUzzer64	
	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max
`exiv2`	11	25	13	37	23	918	423	1,989	24	41	209	993	3,319	4,051	96	1,139
`gdk`	8	454	8	14	22	575	350	1,831	13	76	91	401	442	454	104	2048
`imginfo`	5	10	5	51	23	774	140	1,990	18	200	119	1,979	919	925	42	2,048
`jhead`	7	12	7	20	13	28	60	77	8	13	79	188	265	313	18	30
`tiffsplit`	16	213	14	73	22	55	100	1,930	18	654	42	97	567	765	122	474
`lame`	13	17	13	22	1,705	2,047	53	76	20	29	81	148	758	1,038	56	450
`mp3gain`	9	12	12	16	34	46	34	48	17	23	104	276	346	354	109	468
`wav2swf`	40	53	20	82	76	415	444	4,087	16	94	134	471	598	682	114	2,035
`ffmpeg`	17	596	19	502	27	246	734	5,533	77	1,254	212	1,780	n.a.	n.a.	849	8,195
`flvmeta`	9	14	9	14	19	27	24	50	12	15	598	1,873	470	694	154	318
`mp42aac`	8	23	9	15	58	532	60	1745	18	176	222	2,826	1,155	1,194	112	670
`cflow`	6	7	7	8	1,133	2,023	38	60	23	35	125	597	479	489	261	1,978
`infotocap`	14	23	15	40	24	27	316	428	24	38	496	1,361	597	606	184	636
`jq`	9	11	9	12	13	15	50	72	13	16	78	113	619	783	49	392
`mujs`	17	45	16	30	552	1,533	52	88	23	44	113	2,013	578	729	56	1,623
`pdftotext`	27	76	27	40	4,857	7,861	161	1,967	92	149	139	1,786	2,050	2,055	396	1,190
`sqlite3`	240	595	205	2,031	1,833	2,042	199	1,249	453	1,560	780	1,790	1,580	2,095	n.a.	n.a.
`nm`	8	34	8	26	102	1,350	279	2,046	35	50	78	350	1,739	2,265	57	460
`objdump`	13	171	13	70	108	574	495	2,048	49	1,953	137	2,698	2,625	3,472	849	1,368
`tcpdump`	15	38	16	38	264	607	330	2,040	83	107	160	350	1,464	2,296	119	322
Avg	24.6	121.5	22.2	157.1	545.4	1,084.8	217.1	1,467.7	51.8	326.4	199.8	1,104.5	1,082.6	1,329.5	197.2	1,360.2

4.7 Overhead

Table 7 shows the average and maximum memory consumption of each fuzzer, where we obtain the following observations. (1) From a holistic aspect, AFL, AFLFast and MOPT consume less memory during fuzzing than the other fuzzers, with average memory consumption 24.6 MB, 22.2 MB and 51.8 MB respectively. Nevertheless, T-Fuzz consumes 1,082 MB memory during fuzzing, which is almost 50 times more than that of AFLFast and is the most among the fuzzers. One possible reason is that T-Fuzz leverages Angr [67] to get the *Control Flow Graph (CFG)* of the programs, which may take much memory. (2) When fuzzing the same program, the memory consumption of different fuzzers varies significantly. For example, when fuzzing program `exiv2`, AFL uses no more than 25 MB memory, while in comparison, T-Fuzz uses about 4 GB memory. (3) For the same fuzzer, its memory consumption on various programs also differ greatly. For instance, Angora uses more than 7 GB memory when testing `pdftotext` while its memory consumption on the other programs is less than 2 GB.

5 Further Analysis

Here we conduct evaluations to investigate the previously overlooked factors that may significantly affect a fuzzer's performance, including *instrumentation methods* and *crash analysis tools*.

5.1 Instrumentation Methods

Fuzzers may implement instrumentations in different manners, which leads to diverse characteristics in the compiled binaries. For instance, AFL and Angora implement compile-time instrumentation by writing a wrapper of a compiler (e.g., `afl-clang`, `angora-clang`), while VUzzer leverages Intel PIN [68] to implement binary instrumentation. Therefore, a natural question is: *whether different instrumentation methods affect fuzzing evaluation?* We raise this question based on the following observations. For programs such as `infotocap`, certain crash samples can only make the AFL-instrumented binary crash rather than Angora-instrumented binary. By analyzing these bugs of `infotocap`, we find that they are related to the compilation methods. In this scenario, the failure for Angora to discover these crash samples is due to its instrumentation method rather than its capability of discovering bugs. However, if we overlook the employed different instrumentation methods, we may draw a potentially unfair conclusion that AFL is better than Angora with respect to the detected bugs in this scenario.

Here, we provide an example to show that the compilation methods can impact the bugs. Figure 7 shows a C/C++ code snippet, where there is a heap buffer overflow vulnerability in line 4. However, certain compile-time optimization may skip the erroneous assignment (`x[i]=0` in line 4) and treat the whole assignment statement as a constant zero to print. We compile this code with different compilers (`gcc` and `clang`) and with different optimization levels (O0 - O3). Then, we find that the heap buffer overflow vulnerability cannot be triggered when using `clang` with optimization level O1 - O3, but can be triggered when using `gcc` with optimization level O0 - O3 and `clang` with optimization level O0. As it is difficult to require all the fuzzers to use the same instrumentation method in practice, the difference caused by different compilation (instrumentation) methods cannot be avoided. Therefore, a potentially better solution is using *cross validation* when analyzing the crash samples, i.e., re-execute the crash samples with different complied binaries to check if they can only cause parts of binaries to crash.

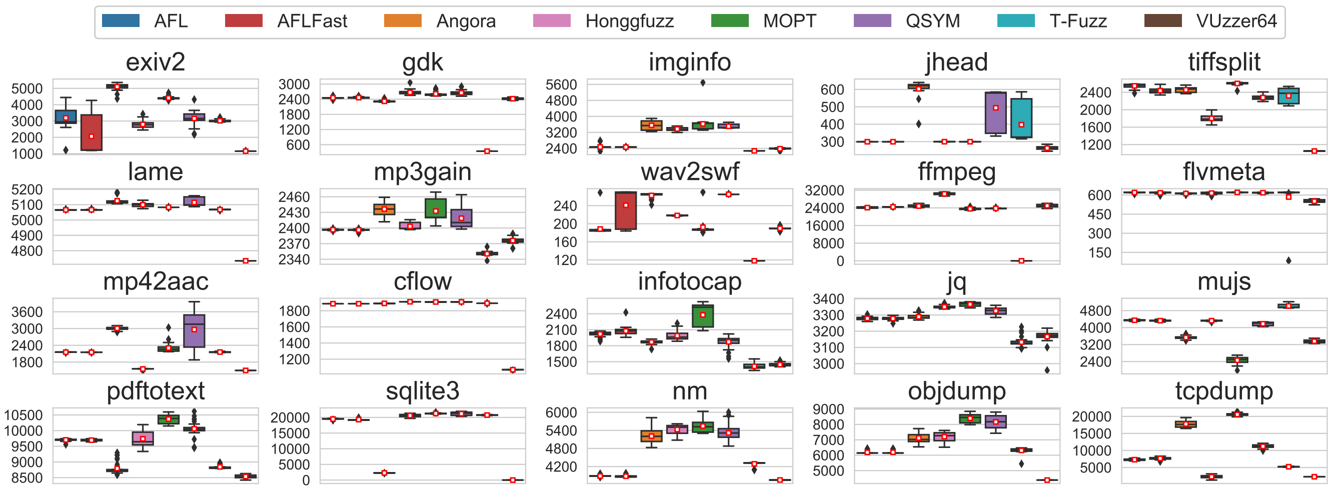


Figure 6: Line coverage on the real-world programs.

```
char* x=malloc(1);
for(int i=0;i<1000000;i++){
    /* x[i]=0 is a heap buffer overflow */
    printf("[%d]=%c\n", i, x[i]=0);
}
```

Figure 7: An example of heap buffer overflow vulnerability.

5.2 Crash Analysis Tools

Different tools have been proposed to analyze what bugs can be triggered by crash samples such as ASan [20] and GDB [21]. During our evaluations, interestingly, we find that using different tools to analyze crash samples can lead to different results, e.g., different numbers of discovered unique bugs. To further examine this, we use ASan and GDB to analyze the crash samples collected from the experiments in Section 4. If a bug can be discovered by executing the ASan- (resp., GDB-) compiled binary with a crash sample, we consider the corresponding crash sample as validated with ASan (resp., GDB). To show the influence of different tools in analyzing crash samples, we list the number of crash samples that can be validated by different tools in Table 8. For the collected 329,857 crash samples, only 61.1% of them can be validated by both ASan and GDB. 14.5% of them can only be validated by GDB and 12.2% of them can only be validated by ASan. Moreover, neither tool can validate the remaining 12.2% crash samples. It is a bit surprising to see that ASan, as a widely adopted analysis tool, only validates 73.3% (12.2%+61.1%) of these crash samples.

Using one analysis tool singly may limit the number of detected bugs, which may further fail to provide comprehensive evaluations on fuzzers. For instance, during our fuzzing experiments in Section 4, we find some crash samples that can trigger *float point exception* bugs on `ffmpeg`. However, we cannot discover the float point exception bugs by executing ASan-compiled binary with these crash samples, while GDB can discover them. Figure 8 shows the number of unique bugs discovered by the fuzzers on `ffmpeg` with ASan and GDB. As shown in Figure 8, the evaluation results are different when using different analysis tools. When using ASan, only Honggfuzz can discover bugs, while using GDB, AFL and AFLFast can also discover bugs. Therefore, it would be better to combine more tools together to analyze crash samples instead of relying on a single tool that may neglect some bugs. In our evaluation (Section 4), we use ASan as the main tool to detect bugs, while adopting GDB as a supplement.

Table 8: Validated crash samples by different tools.

Bug Type	Number	Rate
Neither ASAN or GDB can validate	40,122	12.2%
Only GDB can validate	47,910	14.5%
Only ASAN can validate	40,267	12.2%
Both ASAN and GDB can validate	201,558	61.1%
Total	329,857	100%

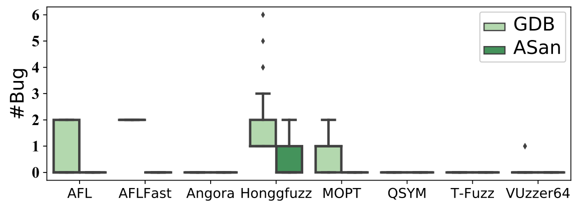


Figure 8: The number of unique bugs discovered on `ffmpeg` with GDB and ASan.

6 Discussion

We discuss the issues in the current fuzzing research field along with our work in this paper as follows.

6.1 The Usability of Fuzzers

A fuzzer with good usability can facilitate its application in practice. Nevertheless, based on our evaluations, we find that the usability issues of fuzzers, especially the academic fuzzers, are more serious than we thought. These usability issues include: having (serious) flaws in implementation, failing to be reproduced, etc. Even worse, some of the fuzzers that have these issues are published at premier conferences in recent years. In this paper, we test the usability of 35 fuzzers, make them available on the UNIFUZZ platform, and conduct extensive evaluations on eight of them. We hope that this work can facilitate further research on improving the usability and the performance of fuzzers.

How to conduct a comprehensive measurement on the usability of fuzzers is an interesting and significant research topic. Nevertheless, the usability of fuzzers is a relatively subjective topic, and it is affected by many factors. First, the usability of a fuzzer highly depends on the domain knowledge of the users. A fuzzing expert may easily use a fuzzer even without the documentation, while a beginner may feel hard to use a fuzzer with poor documentation. Second, there are many factors that can affect the usability of a fuzzer including documentation style, the issues of dependent libraries and tools, the issues of its implementation, the robustness in fuzzing process, etc. For the future research on this problem, we provide the following feasible ways as suggestions. (1) Check the correctness and completeness of a fuzzer's documentation (e.g., whether there exists inconsistency between the documentation and the implementation). It is also an interesting and meaningful research topic on providing guidance or standards on writing the documentation. (2) Test whether a fuzzer can be successfully installed and pass author-provided tests. (3) Test the robustness of a fuzzer during the fuzzing process, and observe whether it has abnormal behaviors (e.g., whether the fuzzer itself crashes during fuzzing). (4) Test whether a fuzzer can reproduce the experimental results as it reported in its paper.

6.2 Fuzzing Experiments

Conducting correct fuzzing experiments is the base of the appropriate evaluations. Klees et al. [13] proposed several guidelines in fuzzing evaluation such as multiple repetitions, using different seed sets, etc. In addition, here we discuss some practical issues that need to be considered when conducting fuzzing experiments. First, it is important to monitor the operating status of a fuzzing experiment such as CPU utilization to determine whether the experiment is executed normally. In general, if the CPU utilization rate is low (e.g., less than 80%), it may indicate that the fuzzing status is abnormal. For instance, when there is a large amount of disk I/O operations, CPU has to wait for these operations before it does real fuzzing work. Second, it is important to mitigate unnecessary disk I/O operations, especially when conducting many fuzzing experiments on a sever simultaneously, where disk I/O can easily become the bottleneck. For instance, the target program may output a large amount of new files during the fuzzing process, which may cause heavy disk output operations, while these output files are not important for evaluations. In this situation, it is suggested using a RAM disk or not saving the output files generated by the target benchmark program.

6.3 The Benchmarks for Evaluating Fuzzers

The current fuzzing benchmarks are not satisfactory [13]. Considering the practical usability issue, we construct a pragmatic benchmark suite which consists of 20 real-world programs at the current version and has the following advantages. First, the UNIFUZZ benchmark programs have various expressiveness in functionality, size, vulnerability, etc., which can provide comprehensive evaluations on fuzzers and can better reflect a fuzzer's performance on the real-world programs. Second, the UNIFUZZ benchmark can be used to provide more objective and fairer evaluations. As shown in Section 4, no fuzzer outperforms the others on all programs, which somehow demonstrates the bias and subjectivity in many existing fuzzing papers. Third, different from traditional benchmark design methods that inject artificial bugs, our method does not change the original code of the real-world programs in order to keep its raw features, but focusing on providing convenient offline result analysis methods. Specifically, for each program, we provide crash analysis methods including crash triage, CVE matching, bug severity analysis, etc. Therefore, the UNIFUZZ benchmark is easily usable like the synthetic benchmarks. In addition, to the best of our knowledge, we are the first to construct the *CVE keywords database* which greatly reduces the human labors in CVE matching.

Note that fuzzing benchmarks need to be updated and improved with the development of fuzzers, and there still needs more research on designing benchmarks. That is why we design UNIFUZZ as an open-source and extensible platform. There are still limitations with the UNIFUZZ benchmark and can be improved and extended from many perspectives. First, in this paper, we select the programs mainly from the top fuzzing papers. In the future, there are many other resources such as vulnerability-related websites [53, 69–72] that can be leveraged to select programs. Second, the current UNIFUZZ benchmark mainly focuses on the general program-level fuzzers. It would be better to incorporate UNIFUZZ with more benchmarks for other types of fuzzers such as compiler fuzzers and kernel fuzzers.

6.4 Performance Metrics

The existing metrics are rough and incomprehensive. To solve the problem, UNIFUZZ provides six categories of metrics which aim to provide more comprehensive evaluations on fuzzers. Here we discuss the limitation of these metrics along with the related interesting research questions which can be considered as the future work. First is the categories of the metrics. In this paper we classify the metrics into six categories. It calls for more research to provide a more reasonable classification. Second, it needs more research on the concrete metrics of each category. For instance, we use the CVSS score and the `Exploitable` tool to evaluate the severity of the bugs. However, each individual metric has its own limitation. As the CVSS score takes multiple factors (e.g., attack complexity and required privileges) into consideration, the single numeric score may not be able to accurately reflect the impact of each individual perspective. `Exploitable` determines the severity of a bug based on a list of rules, whose accuracy may be affected by the rationality of the rules. Thus, the choice of concrete metrics to evaluate the severity of bugs should be updated when better standards/methods are proposed. In addition, it needs more theoretical research on the metrics. For instance, when conducting statistical test on the number of unique bugs, we can only use non-parametric statistical methods such as the Mann-Whitney U test, which makes no assumption on the distribution of the population. It is interesting to study the distribution of the number of bugs in multiple repetitions and provide more suitable metrics to assess it. Third, it is necessary to study the priority of each metric. In our opinion, the number of bugs and the quality of bugs are more important than the stability of finding bugs, as finding less or trivial bugs stably is much less valuable than finding more high-risk bugs occasionally. Fourth, it could be desirable to design a scoring method which combines different metrics to generate a conclusive numerical score for assessing a fuzzer's performance.

7 Conclusion

In this paper, we propose and implement UNIFUZZ, an open-source, holistic, and pragmatic metrics-driven platform for evaluating fuzzers in a comprehensive and fair manner. UNIFUZZ has incorporated 35 fuzzers, 20 real-world benchmark programs, and six categories of performance metrics. We test the usability of the 35 fuzzers and discover a number of flaws. Leveraging UNIFUZZ, we systematically compare the state-of-the-art fuzzers. Based on the experimental results, we have the following important observations. First, no fuzzer always performs better than others, revealing potential subjectivity and bias in the evaluations of existing fuzzing works. Second, the performance of fuzzers on the synthetic benchmark programs may not be consistent with that on the real-world programs, which confirms the importance of using pragmatic benchmark programs. Third, the performance of fuzzers varies with different performance metrics, which indicates that the fuzzers need to be evaluated with more comprehensive performance metrics for reliable assessments. In addition, we identify new factors such as *instrumentation methods* and *crash analysis tools* that can significantly affect the evaluation of fuzzers. We have made UNIFUZZ publicly available to facilitate future fuzzing research.

Acknowledgments

We sincerely appreciate the anonymous reviewers for their valuable comments to improve our paper. This work was partly supported by the National Key Research and Development Program of China under No. 2018YFB0804102 and No. 2020YFB1804705, NSFC under No. U1936215, U1836202, and 61772466, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Zhejiang Provincial Key R&D Program under No. 2019C01055, the Fundamental Research Funds for the Central Universities (Zhejiang University NG-ICS Platform), the Ant Financial Research Funding, the Industrial Internet innovation and development project under No. TC190A449, the Key Research and Development Program of Zhejiang Province under No. 2020C01021, and Major Scientific Project of Zhejiang Lab under No. 2018FD0ZX01. Peng Cheng's research was partly supported by NSFC under grant 61833015. Ting Wang's research was partly supported by the National Science Foundation under Grant No. 1953893, 1953813, and 1951729. Wei-Han Lee's research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] M. Zalewski, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2017.

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[3] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE*

Symposium on Security and Privacy (SP), 2018, pp. 711–725.

[4] Google, "honggfuzz," https://google.github.io/honggfuzz/, 2017.

[5] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1949–1966.

[6] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 745–761.

[7] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Network and Distributed System Security (NDSS)*, 2017.

[9] Google, "OSS-Fuzz - continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2019.

[10] Microsoft, "Microsoft security development lifecycle," https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2018.

[11] "GitHub," https://github.com/, 2019.

[12] "DBLP: Computer science bibliography." https://dblp.uni-trier.de/, 2019.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.

[14] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[15] X. Zhu, X. Feng, T. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*, 2019, pp. 658–663.

[16] "UNIFUZZ platform," https://github.com/unifuzz/overview, 2020.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Network and Distributed System Security (NDSS)*, 2016.

[18] 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

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

[20] "Addresssanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer, 2017.

[21] "GDB: The GNU Project Debugger." https://www.gnu.org/software/gdb/, 2019.

[22] S. Ognawala, F. Kilger, and A. Pretschner, "Compositional fuzzing aided by targeted symbolic execution," *arXiv preprint arXiv:1903.02981*, 2019.

[23] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 499–513.

[24] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[25] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.

[26] "AFLPIN," https://github.com/mothran/aflpin, 2015.

[27] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[28] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Network and Distributed System Security (NDSS)*, 2019.

[29] I. Fratric, "Domato: A DOM fuzzer," https://github.com/googleprojectzero/domato.

[30] "dharma: Generation-based, context-free grammar fuzzer." https://github.com/MozillaSecurity/dharma, 2018.

[31] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 736–747.

[32] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 475–485.

[33] "Fuzzilli," https://github.com/googleprojectzero/fuzzilli, 2019.

[34] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2018, pp. 45–48.

[35] "Jsfuzz: coverage-guided fuzz testing for javascript," https://github.com/fuzzitdev/jsfuzz, 2019.

[36] "jsfunfuzz," https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz, 2019.

[37] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang, "LearnAFL: Greybox fuzzing with knowledge enhancement," *IEEE Access*, vol. 7, pp. 117 029–117 043, 2019.

[38] L. Hayes, H. Gunadi, A. Herrera, J. Milford, S. Magrath, M. Sebastian, M. Norrish, and A. L. Hosking, "Moonlight: Effective fuzzing with near-optimal corpus distillation," *arXiv preprint arXiv:1905.13055*, 2019.

[39] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *Network and Distributed System Security (NDSS)*, 2019.

[40] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program learning," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 803–817.

[41] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.

[42] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 26–47.

[43] "Peach fuzzer," https://www.peach.tech, 2018.

[44] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided Fuzzing With Processor Trace Feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[45] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: An automatic random fuzzer for common file formats," in *ACM SIGPLAN Notices*, vol. 51, no. 12. ACM, 2016, pp. 13–20.

[46] "radamsa: a general-purpose fuzzer," https://gitlab.com/akihe/radamsa, 2018.

[47] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2155–2168.

[48] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[49] C. Labs, "zzuf - multi-purpose fuzzer," http://caca.zoy.org/wiki/zzuf/, 2017.

[50] S. K. Lahiri, R. Sinha, and C. Hawblitzel, "Automatic rootcausing for program equivalence failures in binaries," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 362–379.

[51] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[52] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs." in *Proceedings of the 18th USENIX Security Symposium*, vol. 9, 2009, pp. 67–82.

[53] NVD, "CVE: Common Vulnerabilities and Exposures," https://cve.mitre.org/, 2018.

[54] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 919–936.

[55] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=overlap.

[56] "Flawfinder," https://dwheeler.com/flawfinder/.

[57] "Rough auditing tool for security (rats)," https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[58] "Clang static analyzer," https://clang-analyzer.llvm.org/.

[59] 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.

[60] R. Nuzzo, "Scientific method: statistical errors," *Nature News*, vol. 506, no. 7487, p. 150, 2014.

[61] R. L. Wasserstein, A. L. Schirm, and N. A. Lazar, "Moving to a world beyond "$p< 0.05$"," pp. 1–19, 2019.

[62] "scipy.stats.mannwhitneyu," https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html, 2019.

[63] "The exploitable GDB plugin." https://github.com/jfoote/exploitable, 2018.

[64] "Common Vulnerability Scoring System SIG," https://www.first.org/cvss/, 2019.

[65] "SanitizerCoverage: Clang documentation," https://clang.llvm.org/docs/SanitizerCoverage.html, 2018.

[66] "afl-cov: AFL Fuzzing Code Coverage," https://github.com/mrash/afl-cov, 2018.

[67] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[68] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[69] https://www.exploit-db.com/, 2020.

[70] "CVE details," https://www.cvedetails.com/, 2019.

[71] "Securityfocus," https://www.securityfocus.com/vulnerabilities.

[72] "Securitytracker," https://securitytracker.com/.