Evaluating Directed Fuzzers: Are We Heading in the Right Direction?

TAE EUN KIM, KAIST, Korea
JAESEUNG CHOI*, Sogang University, Korea
SEONGJAE IM, KAIST, Korea
KIHONG HEO, KAIST, Korea
SANG KIL CHA, KAIST, Korea

Directed fuzzing recently has gained significant attention due to its ability to reconstruct proof-of-concept (PoC) test cases for target code such as buggy lines or functions. Surprisingly, however, there has been no in-depth study on the way to properly evaluate directed fuzzers despite much progress in the field. In this paper, we present the first systematic study on the evaluation of directed fuzzers. In particular, we analyze common pitfalls in evaluating directed fuzzers with extensive experiments on five state-of-the-art tools, which amount to 30 CPU-years of computational effort, in order to confirm that different choices made at each step of the evaluation process can significantly impact the results. For example, we find that a small change in the crash triage logic can substantially affect the measured performance of a directed fuzzer, while the majority of the papers we studied do not fully disclose their crash triage scripts. We argue that disclosing the whole evaluation process is essential for reproducing research and facilitating future work in the field of directed fuzzing. In addition, our study reveals that several common evaluation practices in the current directed fuzzing literature can mislead the overall assessments. Thus, we identify such mistakes in previous papers and propose guidelines for evaluating directed fuzzers.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Fuzz testing, Directed fuzzing, Fuzzer evaluation

1 INTRODUCTION

Directed grey-box fuzzing [Böhme et al. 2017] (directed fuzzing in short) has recently gained significant interest in software testing. While traditional (undirected) fuzzing tries to test every part of a program to maximize the number of bugs found, directed fuzzing aims at quickly testing a specific part of the program given by the user. Such a capability of directed fuzzing has been demonstrated in various applications such as testing recently modified code, reproducing crash reports, as well as verifying static analysis alarms.

---

*Corresponding author.

Authors' addresses: Tae Eun Kim, taeeun.kim@kaist.ac.kr, KAIST, Korea; Jaeseung Choi, jschoi22@sogang.ac.kr, Sogang University, Korea; Seongjae Im, seongjae114@kaist.ac.kr, KAIST, Korea; Kihong Heo, kihong.heo@kaist.ac.kr, KAIST, Korea; Sang Kil Cha, sangkilc@kaist.ac.kr, KAIST, Korea.

While various approaches have been proposed to improve the effectiveness of directed fuzzing [Böhme et al. 2017; Canakci et al. 2022; Chen et al. 2018; Du et al. 2022; Huang et al. 2022; Kim et al. 2023b; Lee et al. 2021; Luo et al. 2023; Meng et al. 2022; Nguyen et al. 2020; Österlund et al. 2020; Shah et al. 2022; Srivastava et al. 2022; Zong et al. 2020], there has been less interest in how to properly evaluate directed fuzzers. Although Klees et al. [2018] explored the problems in evaluating undirected fuzzers, such as the impact of initial seed corpus and timeout parameters, we find that there remain many open questions in evaluating *directed* fuzzers. Evaluating directed fuzzers poses unique challenges in that (1) the fuzzer under evaluation must be provided with a specific goal, and (2) one should be able to quantitatively measure how well the fuzzer achieves the goal.

Though one of the main practice for evaluating directed fuzzing is to reproduce a known bug, there exist ambiguities in specifying the target bug. For instance, previous work often specified the target bug with a CVE ID, but this could be an ambiguous specification because CVEs may not come with explicit buggy line information. Hence, it is difficult to know what is the expected buggy behavior to trigger for a given CVE, and which part of the code is associated with the CVE. Depending on the interpretation of the CVE, one may set a slightly different target for the fuzzers, which can change the evaluation result significantly. Previous work has largely overlooked this problem and does not explicitly reveal the target source line(s) used in the experiment.

Another issue arises when we try to measure the performance of fuzzers in reproducing the target bug. That is, it is not trivial to determine whether the fuzzers have found a crash that corresponds to the target bug. This process is often referred to as *crash triage*. Klees et al. [2018] recommend triaging crashes based on the patch that fixes the target bug. For instance, if a crash detected by the fuzzer disappears after applying the corresponding patch, we consider the fuzzer has found the target bug. On the other hand, another popular method is to use a sanitizer (such as AddressSanitizer [Serebryany et al. 2012]) to triage the crashes. For instance, one can check if the sanitizer report contains the crash location detected by the fuzzer. However, our study reveals that there are subtle issues in both methods, which can significantly affect the evaluation results according to our experiments.

In addition, previous work often overlooks the static analysis time despite its impact on overall performance evaluations of the fuzzers. In directed fuzzing, static analysis is often used to provide better guidance for the fuzzers. However, static analysis may be computationally expensive, sometimes even more expensive than the fuzzing itself. With our experiment, we show that the static analysis time can significantly affect the overall performance.

Lastly, there has been no systematic study on the impact of the repetition count and the appropriate use of statistical tests in directed fuzzing evaluation. Our experiment indicates that repetition counts adopted in previous work are often too small to effectively mitigate the intense randomness of directed fuzzing. Moreover, we highlight that the Mann-Whitney U test (*a.k.a.* MWU test) [Mann and Whitney 1947], which is commonly used in evaluating fuzzers, cannot account for the *timed-out* cases in directed fuzzing. Instead, we suggest that the standards from survival analysis should be employed more widely when interpreting the result of repeated runs with directed fuzzers.

In this paper, we identify common ambiguities and pitfalls in each step of directed fuzzing evaluation: target specification, triage method, static analysis, repetition of fuzzing, and statistical tests. We not only survey the previous work but demonstrate that these concerns are real threats in real-world fuzzing evaluations through our concrete experiments. Our experiments were conducted with 5 state-of-the-art directed fuzzers on a benchmark of 12 real-world bugs, scaling up to a total volume of 30 CPU-years. In particular, we tested the fuzzers with varying target lines for the same CVEs and with different crash triage methods. We also measured the static analysis time to study its impact on the overall performance of fuzzers. Furthermore, we tested different repetition counts

---

Algorithm 1: Evaluating Directed Fuzzer

---

    input :fuzzing algorithm *Fuzz*, program *P*, timeout *Timeout*,

             target bug *T*,   `// Problem` **P1** `in §4`

             crash triage method *Triage*,   `// Problem` **P2** `in §5`

             pre-processing algorithm *Preproc*,   `// Problem` **P3** `in §6`

             number of repetitions *Reps*,   `// Problem` **P4** `in §7.1`

    output:statistical number on the Time-To-Exposure (TTE) of target *T*

---

1 *P′* ← *Preproc*(*P*, *T*)

2 *TTEs* ← [ ]   `// List of measured TTEs`

3 **for** *i* ← 1 **to** *Reps* **do**

4   Crashes ← *Fuzz*(*P′*, *Timeout*)

5   *TTE* ← ComputeTTE (*Crashes*, *Triage*)   `// Shortest time to exposure`

6   *TTEs*.insert(*TTE*)

7 **return** StatCalc(*TTEs*)   `// Problem` **P5** `in §7.2`

---

and tried different statistical tests to investigate how the evaluation results are affected by these choices.

    Our contributions are summarized as follows:

- We identify common ambiguities and pitfalls in evaluating directed fuzzers including target specification, triage process, static analysis time, repetition count, and statistical analysis.
- We propose constructive guidelines to handle the issues for conducting transparent and reproducible evaluations.
- Our extensive experiments (30 CPU-year volume) demonstrate that the concerns above are realistic threats in directed fuzzing evaluation. We faithfully open-source the artifacts for our experiments [Kim et al. 2024].

2 EVALUATION OF DIRECTED FUZZING

In this section, we first present a generalized algorithm to evaluate the performance of directed fuzzers. We then introduce several issues that can arise in evaluating directed fuzzers.

2.1 Evaluation Algorithm

At a high level, directed fuzzing is a process of generating test cases that can reach a specific target location in a program and eventually trigger a bug in that location. For brevity, we assume that a single location is given to the fuzzer, but our study is generally applicable to multi-target cases. Our primary focus is on grey-box fuzzing because it is the most widely used approach today. Nevertheless, we believe our findings and discussions are also applicable to other approaches such as black-box [Wang et al. 2010] or white-box [Christakis et al. 2016] fuzzing, as our study does not make any assumption about the internals of directed fuzzing methodology.

    While existing directed fuzzers employ different techniques to guide the fuzzing process, we can define a generalized process to evaluate their performance as they share the same goal: reaching a target location and triggering a bug in that location. Algorithm 1 describes the common process for evaluating a directed fuzzer. The algorithm takes in seven parameters as input: a fuzzing algorithm *Fuzz*, a program *P*, a timeout *Timeout*, a target bug *T*, a crash triage method *Triage*, a preprocessing algorithm *Preproc*, and the number of repetitions *Reps*.

Using these parameters, the algorithm evaluates the effectiveness of the given fuzzing algorithm *Fuzz* based on the time required to expose the target bug, which is often referred to as *Time-To-Exposure* or *TTE*. A fuzzer with a shorter TTE is deemed more effective in reproducing the target bug. If we are to evaluate the effectiveness of *undirected* fuzzers, there are various metrics that have to be considered together, such as code coverage, number of unique bugs found, and the severity of the found bugs. However, note that the evaluation of *directed* fuzzer is always performed with regard to a specific target bug. Therefore, TTE is the most direct metric for measuring the effectiveness of a directed fuzzer in terms of finding the targeted bug.

In Line 1, the fuzzer performs a preprocessing step for the program under test. This step may involve a simple instrumentation for runtime feedback or a detailed static analysis for better guidance. The instrumented program is then used in Line 4 to perform directed fuzzing until the timeout *Timeout* is reached. After the fuzzing phase, we classify the discovered crashes with the provided triage method (*Triage*) and compute the TTE of the target bug in Line 5. This process is repeated *Reps* times (Line 3), and finally, the algorithm returns the effectiveness of the fuzzer in the form of statistical numbers, such as the median of TTEs and the p-value of a statistical test (Line 7). Note that most of the statistical tests are performed with the TTEs of two fuzzers to compare, not over the TTEs of a single fuzzer. We omitted this detail in our algorithm for simplicity.

One may adapt this algorithm to evaluate directed fuzzers in terms of the time to reach a target location, often dubbed *Time-To-Reach* or *TTR*, instead of the time to expose a target bug. Although our algorithm is applicable to such scenarios, we focus on evaluating the effectiveness of directed fuzzers in terms of TTE, as it is the primary metric used in most directed fuzzing papers.

Despite the simplicity of Algorithm 1, we find several critical issues in the evaluation of directed fuzzers, marked as **P1**–**P5** in the algorithm. We outline the pitfalls of directed fuzzing evaluation in §2.2 and discuss them thoroughly in the following sections.

2.2 Issues in Evaluating Directed Fuzzers

We now introduce several issues that can arise in evaluating directed fuzzing. We reviewed 14 directed fuzzing papers recently published in security or software engineering venues and investigated how they handled these issues. Table 1 summarizes several key aspects of their tools and the way they evaluated their tools, such as the open-sourcing status, benchmark selection, specification of the target line, triage method, static analysis time, repetition count, and the choice of statistical test method. In the rest of the paper, we identify and discuss the following five problems that can significantly affect the evaluation results.

P1 We observe that a target bug (*T*) to the fuzzer (Line 1 in Algorithm 1) is often not clearly specified. While most of the existing directed fuzzers take in buggy line number(s) as input, many existing papers simply report the CVE IDs used in their experiments without further details. Unfortunately, it is not always straightforward to identify the buggy location of a program from the CVE description [MITRE 2023]. Despite such an issue, many papers do *not* explicitly specify the target lines used in their experiments, as indicated in Table 1. We further investigate this problem in §4.

P2 We find that crash triage is often not discussed in detail, although it can significantly affect the measured TTE. After a fuzzing campaign, one should triage the found crashes (Line 5 in Algorithm 1) to determine whether the targeted bug was found. A common approach is *sanitizer-based* triage, which uses a sanitizer such as AddressSanitizer [Serebryany et al. 2012] (*a.k.a.* ASAN) to obtain the type and location of the crash. This allows us to check whether the expected type of bug occurred in the expected location. However, writing a concrete logic for such checks is error-prone and requires a deep understanding of the target bug. For instance, if the target bug can raise crashes in multiple locations, the triage logic must accept all such locations. Another popular method

Table 1. Summary of directed fuzzing papers we surveyed. For each column, ● indicates that the code or data is publicly available, while ◐ means the information is partially available and - means the information is not available. **Open-sourced** column denotes whether the tool is open-sourced. **Benchmark** column summarizes the types of benchmarks used in the paper. We only considered the benchmarks for TTE experiments. **Target Line** indicates whether the paper or artifact explicitly specifies the target lines provided to the fuzzers. **Triage Method** denotes how the found crashes were classified and **S.A. Time** denotes whether the paper reports the time spent on the static analysis. **Reps** denotes how many times each fuzzer was run repeatedly in the experiment. **Stat. Test** denotes the statistical test used to compare the performance of the fuzzers.

Fuzzer	Open-sourced	Benchmark[b]	Target Line[c] (P1)	Triage Method (P2)	S.A. Time[d] (P3)	Reps (P4)	Stat. Test[e] (P5)
AFLGo [Böhme et al. 2017]	●	CVE	◐	Patch-based	-	20	MWU, VDA
Hawkeye [Chen et al. 2018]	-	CVE	-	-	◐	8	VDA
FuzzGuard [Zong et al. 2020]	-	CVE	-	-	-	-	-
ParmeSan [Österlund et al. 2020]	●	CVE, FTS [Google 2021]	◐	-	◐	30	MWU
TargetFuzz [Canakci et al. 2022]	-	Magma [Hazimeh et al. 2021]	●	Assertion-based	-	20	-
UAFuzz [Nguyen et al. 2020]	●	CVE	●	Sanitizer-based	◐	10	VDA
CAFL [Lee et al. 2021]	-	CVE, LAVA [Dolan-Gavitt et al. 2016]	●	Sanitizer-based	●	3	MWU
Beacon [Huang et al. 2022]	◐[a]	CVE	-	-	●	10	MWU
LTL-Fuzzer [Meng et al. 2022]	●	CVE	-	Assertion-based	-	10	MWU
MC² [Shah et al. 2022]	●	Magma	-	-	●	20	MWU, VDA
SieveFuzz [Srivastava et al. 2022]	●	CVE, Magma, CGC [Bits 2017]	-	Assertion-based	◐	10	VDA
WindRanger [Du et al. 2022]	●	CVE, FTS	●	Sanitizer-based	-	20	MWU, VDA
DAFL [Kim et al. 2023b]	●	CVE	◐	Patch-based	●	40	-
SelectFuzz [Luo et al. 2023]	●	CVE, FTS	●	Sanitizer-based	-	5	MWU

a Beacon is not open-sourced, but the pre-built binary is available in the provided Docker image.
b FTS stands for Fuzzer Test Suite and CGC for DARPA Cyber Grand Challenge.
c Fuzzers marked with ◐ specified the target line only for the subset of the benchmark.
d Fuzzers marked with ◐ only report their own static analysis time.
e MWU stands for Mann-Whitney U test and VDA for Vargha and Delaney A measure.

is *patch-based* triage, which replays each found crash on a patched version of the program that no longer contains the target bug. If the patched program gracefully exits, we can classify the crash as the target bug. Unfortunately, obtaining a patch that precisely fixes the target bug can be challenging. Alternatively, there is *assertion-based* triage, which inserts an explicit assertion that checks whether the target bug occurred. If an input makes the assertion fail, we consider that the input has triggered the target bug. While it may appear simple, formulating concrete logic for the assertion also requires a careful investigation on the target bug. According to our observation, such pitfalls are largely overlooked in previous papers. Only 9 out of 14 papers specify the type of the crash triage method used in their experiments. Moreover, only 5 of them provide enough details about the triage process required to reproduce the experiment [Canakci et al. 2022; Kim et al. 2023b; Nguyen et al. 2020; Shah et al. 2022; Srivastava et al. 2022]. In §5, we provide an in-depth discussion of these pitfalls and show that a small mistake can significantly change the measured TTEs.

P3 We find that the static analysis time is often not considered in the measured TTE. Although many directed fuzzers perform static analysis in the preprocessing step (Line 1 in Algorithm 1), the time required for static analysis is often not included when comparing the TTEs, or not even reported in most cases. Unfortunately, our study shows that the static analysis time can have a significant impact on the measured TTE. In §6, we discuss the importance of considering the static analysis time and show its impact on the TTE.

P4 We find that the number of repetitions is often too small. Due to the random nature of fuzzing, it is well-known that fuzzing experiments must be repeated multiple times [Klees et al. 2018]. Accordingly, directed fuzzing papers repeat the fuzzing phase multiple times to mitigate the randomness of fuzzing (Line 3 in Algorithm 1). However, our study reveals that directed fuzzing evaluation requires a larger number of repetitions than normally expected by the researcher. In §7.1, we elaborate on this problem and provide guidelines for performing repeated experiments.

P5 We find that the current practice of statistical tests is often not appropriate for directed fuzzing. Directed fuzzing papers typically report (1) the median of TTE measured over repeated runs and (2) the *p-value* obtained from the MWU test (Line 7 in Algorithm 1) to report the result obtained from the repeated experiments. However, our study reveals that the MWU test is not appropriate for directed fuzzing, as directed fuzzers often fail to find the target bug within the given time limit. Additionally, we demonstrate that the result of the MWU test may mislead the readers in certain cases. In §7.2, we elaborate on this problem and provide guidelines for analyzing repeated experiments.

In summary, we found that the above issues are often not clearly discussed in the previous work. This poses a hurdle in identifying the exact settings for the evaluation. In the following sections, we demonstrate that these settings can change the evaluation results significantly.

3 EXPERIMENTAL SETUP

We conducted various experiments to show that the aforementioned issues are practical threats to the directed fuzzing evaluation. In this section, we describe the experimental setup used in our paper. The artifact of our experiment is available at Zenodo [Kim et al. 2024].

Fuzzers. We selected 5 open-sourced directed fuzzers for our experiment: AFLGo (commit b170fad), Beacon (Docker image a09c8cb), WindRanger (Docker image 8614ceb), DAFL (commit a6fcc56), and SelectFuzz (commit 9dea54f).

Target bugs. We used 12 target bugs from previous directed fuzzing papers, which are summarized in Table 2. We first included 6 bugs from the binutils package, since they are widely used in the papers we listed in Table 1. In addition, we added 6 more bugs that are shared by DAFL, Beacon, and partially by SelectFuzz. By default, we reused the target lines from the

Table 2. Our benchmark selection.

Package	Prog.	CVE	Previous Work
Binutils	cxxfilt	2016-4487	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL, UAFuzz
		2016-4489	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL
		2016-4490	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL
		2016-4491	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL, SelectFuzz
		2016-4492	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL
		2016-6131	AFLGo, Beacon, WindRanger, Hawkeye, ParmeSan, DAFL
Libming	swftophp	2016-9827	Beacon, DAFL, SelectFuzz
		2016-9829	Beacon, DAFL
		2016-9831	Beacon, DAFL, CAFL
		2017-9988	Beacon, DAFL
		2017-11728	Beacon, DAFL, SelectFuzz
		2017-11729	Beacon, DAFL

open-sourced artifact of DAFL [Kim et al. 2023a]. In §4, we also study the effect of using different target lines.

Sanitizer. We disabled sanitizers when preprocessing target programs for the fuzzing session because Beacon does not support programs instrumented with a sanitizer. However, we can still use sanitizers when replaying the test cases and analyzing the found crashes.

Crash Triage. We utilized patch-based triage as our default triage method. In §5, we further explore other triage methods.

Timeout & Repetitions. We repeated all the experiments 160 times, each with a timeout limit of 24 hours. In §7, we provide the justification for using such a large repetition number.

Machine Environment. Each run of fuzzing was performed in a Docker container assigned with 4GB of main memory and a single CPU core of Intel Xeon Gold 6226R (2.90 GHz).

Note that our objective is not to compare and rank the performance of the fuzzers. Instead, we aim to find out challenges in the evaluation process by thoroughly investigating each target bug. Therefore, we do not focus on using a comprehensive set of benchmarks. Despite the limited scale of our benchmark, our experiments identify realistic threats to the validity of the evaluation, indicating that such threats will persist with a larger benchmark.

4 TARGET SPECIFICATION

The first step of evaluating a directed fuzzer is to specify a target bug. In this section, we review the current practice and discuss the issues involved in this step.

4.1 Current Practice: Reporting CVE ID

The most common practice in the literature is to report the CVE ID of a target bug used in the experiment. While the CVE database provides useful information about the target bug—such as the bug type, relevant function names, a patch, and even a PoC input for triggering the crash—it is insufficient to decide which source line should be used as the target line for directed fuzzing.

This is mainly because a single CVE may be associated with multiple bugs that have different root causes. For instance, CVE-2016-4492 [MITRE 2016c] in Figure 1 has two distinct crashing locations, each of which is caused by a different root cause. In the provided code, get_count can return a negative integer in n. However, the two call sites of get_count in the figure use the returned n as an array index without checking its range, which leads to crashes at Line 9 and Line 14, respectively. The developer's patch [Böhme 2016b] adds a range check for n before Line 9 and Line 14, which also

```
1  int do_type(struct work_stuff *work, char **mangled, ...) {
2    int n;
3    ...
4    switch (**mangled) {
5      ...
6      case 'T':
7        get_count (mangled, &n);
8  +      if ( n < 0 ) break;                    // line added by patch
9        remembered_type = work->typevec[n];    // crash location 1
10       ...
11     case 'B':
12       get_count (mangled, &n);
13 +      if ( n < 0 ) break;                    // line added by patch
14       string_append (result, work->btypevec[n]);  // crash location 2
15     ...
16   }
17 }
```

Fig. 1. Simplified code and patch of CVE-2016-4492.

Table 3. Performance comparison of using different target lines for CVE-2016-4492.

| | Median TTE (s) | | | | |
Target Location	AFLGo	Beacon	WindRanger	SelectFuzz	DAFL
Line 9	373	333	2,460	432	787
Line 14	332	499	339	581	149

suggests that the two crashes are caused by different root causes. Nevertheless, these two crashes are grouped under the same CVE ID and the PoC inputs from the original bug report [Böhme 2016] also reproduce the crashes in both of the lines. Therefore, both lines can be considered as equally valid target sites for CVE-2016-4492, according to the bug report.

One may wonder whether choosing one line over the other makes any meaningful difference. While both lines reside in the same function and share conceptually similar code patterns, our experiment demonstrates that the choice of a target line makes a significant difference when evaluating directed fuzzers. Table 3 presents the median of TTEs computed over 160 repetitions, with directed fuzzers given different target lines for CVE-2016-4492. The result indicates that the performance significantly depends on which target line is used. For example, WindRanger underperformed when Line 9 was used as the target line, but it demonstrated significantly better performance when provided with Line 14 as input.

We believe that the ideal solution for this case is to split CVE-2016-4492 into two distinct bugs, each having its own crashing location as the target line. Then we can use each of them separately for directed fuzzing evaluation. It is understandable that the CVE maintainers may want to group the two bugs under the same CVE ID for their convenience since they have similar code patterns. From the perspective of directed fuzzing, however, the two lines should be treated as distinct bugs.

The case study of CVE-2016-4492 shows that each researcher can make different decisions during the evaluation process if a CVE ID is the only available information about the target bug. Moreover, such decisions make a significant difference in the evaluation result. Unfortunately, many directed fuzzing papers only provide a CVE ID as the target bug description. Out of the 12 papers that used

```
1  void string_appendn(string *p, char *s, int n) {
2    if (n != 0) {
3      string_need(p, n);
4      memcpy(p->p, s, n);
5      p->p += n;
6    }
7  }
8
9  int gnu_special(struct work_stuff *w, char **mangled, string *declp) {
10   ...
11   int n = consume_count(mangled); // returns -1 in n
12 + if (n == -1) break;            // line added by patch
13   string_appendn(declp, *mangled, n);
14   ...
15 }
```

Fig. 2. Simplified code and patch of CVE-2016-4489.

Table 4. Performance comparison of using two different target lines for CVE-2016-4489.

Target Location	Median TTE (s)			
	Beacon	**WindRanger**	**SelectFuzz**	**DAFL**
Line 4	252	243	274	246
Line 13	248	98	154	222

CVEs in the experiment, 6 papers simply report the CVE IDs of their target bugs. Thus, there is no guarantee that the experimental results of the previous work are consistent with each other.

4.2 Issues When Specifying Target Line

One natural solution is to explicitly provide a concrete target line, as shown in Table 3. However, we still have to be careful when selecting a specific line for the directed fuzzer. Suppose that a crashing line is given and various call contexts can reach this line. While some of these call contexts are buggy, other contexts are not, thus irrelevant to the crash. Under this circumstance, two options are available when choosing a target line. First, we can simply choose the final crashing line. Alternatively, we can choose a line that appears in the stack trace of the crash, which may better represent the context of the target bug.

A good example of such a case is CVE-2016-4489 [MITRE 2016a], which is presented in Figure 2. This bug is triggered when gnu_special calls string_appendn with −1 as the argument n. The crash occurs at Line 4 of string_appendn, which can be an obvious target line for directed fuzzers. However, one may also note that string_appendn has eight call sites and Line 13 of gnu_special is the only call site that passes an invalid value to the argument n and causes a crash. Moreover, the developer's patch [Böhme 2016a] also adds a graceful exit before Line 13 when n has the value of −1. In this respect, Line 13 can be a target line that is more closely related to the root cause of the bug.

To show the impact of this choice, we ran directed fuzzers with different target lines for CVE-2016-4489. Note that despite the different target lines, we checked for the same crash, the crash that occurs at Line 4. Table 4 summarizes the result. AFLGo is not included here because it failed

to run when Line 4 was given as input[1]. Interestingly, the four fuzzers demonstrated comparable performance with Line 4, the crashing line, being the target line, while WindRanger significantly outperformed the other fuzzers with Line 13.

Our observation shows there can be multiple possible options for choosing a target line for a bug, and different choices can lead to significantly different evaluation results. Particularly, choosing the final crashing line is not always the best option because it may miss the context of the target bug and mislead the directed fuzzers.

> **P1** *Lessons learned.* The current practice of using the CVE ID as the target specification is ambiguous. Our experiment shows that such ambiguities can significantly impact the evaluation results. Furthermore, it is nontrivial to choose the best target line for a bug in practice. Therefore, we believe that it is crucial to publicly open the target lines used in the evaluation so that the results can be reproduced and compared consistently.

5 CRASH TRIAGE

Crash triage in directed fuzzing is the process of determining whether a discovered crash is actually the targeted bug or not. Since the performance of directed fuzzers is typically evaluated with the TTE of the target bug, it is crucial to correctly identify the target bug. In this section, we explore the current practice of crash triage in the literature and discuss the potential pitfalls of each method.

5.1 Current Practice 1: Sanitizer-based Triage

The most widely used method is to classify crashes by investigating the sanitizer log obtained from each crash. A sanitizer log contains the crash location and the stack trace. Most of the CVEs previously used in directed fuzzing evaluation are reported with a sanitizer log of a crash, which can be used as a reference. By comparing it with the log of a crash found during fuzzing, we can determine if the found crash is the targeted bug.

However, there are several possible choices when comparing two sanitizer logs. We can simply compare the entire stack traces to decide if the two crashes are the same. Alternatively, we can compare the crashing lines only, without considering the call contexts. The former can be too strict and prone to false negatives when the target bug can raise crashes with different stack traces. The latter can be too permissive and result in false positives because different bugs with distinct root causes can share the same crashing line.

Moreover, the sanitizer log from the bug report may not provide comprehensive information about the target bug. For example, it is common for a single bug to potentially cause crashes in multiple lines depending on the runtime environment. In such a case, we have to carefully design a triage logic because some crashes may not be explicitly mentioned in the original bug report.

CVE-2016-9831 is a good example that illustrates this issue. The CVE database provides the following description for CVE-2016-9831 [MITRE 2016d]:

> "Heap-based buffer overflow in the `parseSWF_RGBA` function in parser.c in the `listswf` tool in `libming` `0.4.7` allows remote attackers to have unspecified impact via a crafted SWF file."

Figure 3 presents the simplified buggy code. The program assigns an input integer value to `g->NumGradients` at Line 3. However, the value is used in the loop condition without checking the upper bound. This leads to out-of-bound accesses of array `g->GradientRecords`, which can cause a crash in various locations. First, the original ASAN log from the CVE report [MITRE 2016d]

---

[1]While other papers often report the TTE of AFLGo for this bug by providing multiple target lines extracted from a stack trace, we believe this is an unfair setting.

```
1  void parseSWF_MORPHGRADIENT(FILE *f, SWF_MORPHGRADIENT *g) {
2    ...
3    g->NumGradients = readUInt8(f);
4    // no bound check on g->GradientRecords
5    for (i = 0; i < g->NumGradients; i++)
6      parseSWF_MORPHGRADIENTRECORD(f, &(g->GradientRecords[i]));
7  }
8
9  void parseSWF_MORPHGRADIENTRECORD(FILE *f, SWF_MORPHGRADIENTRECORD *r) {
10   r->StartRatio = readUInt8(f);  // potential crash
11   parseSWF_RGBA(f, &r->StartColor);
12   r->EndRatio = readUInt8(f);    // potential crash
13   parseSWF_RGBA(f, &r->EndColor);
14 }
15
16 void parseSWF_RGBA(FILE *f, SWF_RGBA *rgb) {
17   rgb->red   = readUInt8(f);    // crash location by the PoC input
18   rgb->green = readUInt8(f);    // potential crash
19   rgb->blue  = readUInt8(f);    // potential crash
20   rgb->alpha = readUInt8(f);    // potential crash
21 }
```

Fig. 3. Simplified code and patch for CVE-2016-9831.

Table 5. Performance comparison of using different sanitizer-based triage logic for CVE-2016-9831. The first row checks whether the crash occurred exactly at Line 17, while the second and third row accept crashes from more lines as CVE-2016-9831.

	Median TTE (s)				
Lines Checked	**AFLGo**	**Beacon**	**WindRanger**	**SelectFuzz**	**DAFL**
17	1,418	1,069	487	1,777	1,218
17-20	167	177	174	218	103
17-20, 10, 12	159	155	155	200	93

contains a single crash at Line 17. However, depending on the memory layout at runtime, this buffer overflow can also cause crashes in all the other lines of parseSWF_RGBA. Moreover, it can even lead to crashes in Line 10 and 12 in a different function, which is not mentioned in the CVE description. While we believe the crashes from these lines should be also classified as the same target bug, other researchers may choose to strictly follow the CVE description and reject these crash lines in their triage logic. To avoid such discrepancy, one must explicitly specify the logic used for the sanitizer-based triage.

Table 5 confirms that this issue in the triage logic can make a significant difference in the TTE of this bug. DAFL shows the average performance when we strictly check for Line 17, but becomes the best-performing fuzzer when we additionally check for other lines.

Since there is no standard approach for sanitizer-based triage that can be applied to all bugs, each researcher can write different triage logic, depending on one's understanding of the bug. As we have observed, such differences in triage logic can lead to different results in the evaluation. Thus, we believe a paper should avoid simply mentioning the use of sanitizer log as its triage

```
1  void d_print_comp(d_print_info *dpi, demangle_component *dc) {
2    d_component_stack self;
3    self.dc = dc;
4    self.parent = dpi->component_stack;
5    dpi->component_stack = &self;
6    d_print_comp_inner (dpi, options, dc);
7    ...
8  }
9
10 void d_print_comp_inner(d_print_info *dpi, demangle_component *dc) {
11   ...
12   switch (dc->type) {
13     ...
14     case DEMANGLE_COMPONENT_TAGGED_NAME:
15       d_print_comp(dpi, d_left (dc));  // call d_print_comp with left subtree of dc
16       d_print_comp(dpi, d_right (dc)); // call d_print_comp with right subtree of dc
17     ...
18   }
19 }
```

Fig. 4. Simplified code of CVE-2016-4491.

Table 6. Performance comparison of using different patches for patch-based triage for CVE-2016-4491. The number in parentheses denotes the number of times each fuzzer found the target bug out of 160 repetitions. Note that median cannot be computed if the fuzzer failed to find the bug for more than half of the times.

	Median TTE (s) (Success Iterations)				
Patch Used	**AFLGo**	**Beacon**	**WindRanger**	**SelectFuzz**	**DAFL**
Incomplete [Wielaard 2016a]	N.A. (16)	N.A. (11)	N.A. (14)	N.A. (8)	N.A. (63)
Complete [Wielaard 2016a,b]	1371 (158)	1393 (160)	3111 (159)	1754 (158)	N.A. (71)

method. Instead, researchers must publicize the concrete logic used in the crash triage, to facilitate cross-checks and public discussion within the research community.

5.2 Current Practice 2: Patch-based Triage

Another commonly used approach for triage is utilizing a patch of the target bug. If a patch for a given bug is available, we can build a fixed version of the target program that does not contain the bug anymore. In this case, if a crashing input of the original program does not crash the patched version, we can conclude that the input triggers the target bug.

However, obtaining a correct patch for a given bug can be challenging in practice. Sometimes, patches are incomplete and fail to fully remove the bugs. Moreover, developers occasionally write a single patch that fixes multiple bugs at once [Böhme 2016b], as pointed out by Klees et al. [2018]. If the targeted bug is fixed by such a patch, we must split it to obtain a patch that only fixes our target.

CVE-2016-4491 [MITRE 2016b] is an example that highlights the challenges in patch-based triage. This CVE is a stack overflow bug due to an infinite recursion. As shown in Figure 4, d_print_comp traverses a graph-like structure dc with recursive calls. The first patch [Wielaard 2016a] detects a cycle in dc and terminates the program, but it turned out that this fix cannot prevent the stack

```
1  void d_count_templates_scopes (int *num_templates, int *num_scopes, demangle_component *dc) {
2    ...
3    switch (dc->type) {
4      ...
5      case DEMANGLE_COMPONENT_DTOR:
6        d_count_templates_scopes (num_templates, num_scopes, dc->u.s_dtor.name);
7        break;
8      ...
9    }
10 }
```

Fig. 5. Simplified code of CVE-2019-9071.

overflow in d_print_comp completely. A subsequent patch [Wielaard 2016b] was applied afterward to check for more conditions.

Table 6 presents the result of measuring TTEs for this bug with different patches used for triage. When the initial incomplete patch is used for triage (first row), the result shows that most of the generated inputs by the fuzzers are not classified as the target bug. However, if the complete patch is used (second row), the measured TTEs of all the fuzzers are greatly reduced. This result indicates that the complete patch can classify more crashes as the targeted one. Furthermore, the relative ranking between fuzzers fluctuates depending on the patch used. When the incomplete patch is used, DAFL seems to perform best, but with the complete patch, it turns out that AFLGo has the shortest median TTE.

Moreover, even if we obtain a correct patch for the target bug, we should be aware of another pitfall. When a crashing input can actually trigger multiple bugs, patch-based triage may lead to a false negative. For example, one of the inputs we found can trigger CVE-2016-4491 in the original program and trigger a different bug, CVE-2019-9071 [MITRE 2019], in the patched program. As shown in Figure 5, CVE-2019-9071 is a stack overflow bug that occurs in the function d_count_templates_scopes. When an input with a pathologically constructed mangled name is given, d_count_templates_scopes recursively calls itself at Line 6 until the stack overflows. While CVE-2016-4491 and CVE-2019-9071 are both stack overflow bugs, they occur in different functions and have different root causes. If the triage scheme only checks whether the patched program gracefully exits or not, we will fail to recognize that this input originally triggered CVE-2016-4491. To cope with such inputs, we need to compare the two stack traces more carefully. For instance, we may additionally check whether the crashing function has changed in the patched version.

In order to avoid such pitfalls, we must carefully choose and explicitly specify the patch that completely and precisely removes the target bug. Moreover, as we have observed in the case of CVE-2016-4491 and CVE-2019-9071, we must not only specify the patch but also share the end-to-end process of triaging the bug with the patch.

5.3 Possible Solutions

As a possible solution, we suggest using a benchmark that comes with a triage logic. In general, there are two types of benchmarks that can be used for directed fuzzing: (1) benchmarks with explicit assertions and (2) synthesized benchmarks.

5.3.1 Assertion-based Triage. One possible solution is to use a benchmark with explicit assertions for each target bug. For example, the Magma benchmark [Hazimeh et al. 2021] consists of a set of known buggy programs whose bug conditions are manually annotated as assertions. Benchmarks

with such assertions provide a clear and unambiguous triage logic for each bug. Consider the previous example of CVE-2016-9831 in Figure 3. We can insert an assertion before Line 6 to check whether the index i is less than the size of GradientRecords. If a test case makes this assertion fail, we can consider that the targeted buffer overflow is triggered. Despite the difficulty of manually inserting correct assertions for the target bugs, constructing such a benchmark can be a promising direction for future research.

5.3.2   Synthesized Benchmark. Another possible solution is to use a synthesized benchmark. There are two types of synthesized benchmarks: (1) benchmarks that inject synthetic bugs into an existing program [Dolan-Gavitt et al. 2016; Pewny and Holz 2016; Roy et al. 2018; Zhang et al. 2022] and (2) benchmarks that are synthesized from scratch [Lee et al. 2022]. The former type of benchmarks may produce false positives when evaluating directed fuzzers. For instance, fuzzers may trigger other bugs that already exist in the original program, which is independent of the injected bug. Additionally, the process of injecting bugs might unintentionally alter the intended behavior of the original programs, potentially introducing unintended errors. Thus, triage logic is still required to distinguish the injected bug from other bugs. On the other hand, the latter type of benchmarks can be a promising solution [Lee et al. 2023]. Since the program is synthesized from scratch, we can guarantee that the synthesized benchmark contains only the target bug. Thus, directed fuzzers only have to check whether it has triggered any bugs in the synthesized benchmark.

> **P2** ***Lessons learned.*** Both sanitizer-based and patch-based triages can mislead the evaluation of directed fuzzers depending on the details of the triage logic. Thus, it can be a desirable direction to use benchmarks with explicit assertions or synthesized bugs that provide explicit triage logic. Furthermore, we argue that the whole triage process should be publicized for consistent evaluation.

6   STATIC ANALYSIS TIME

In the previous section, we focused on triaging the crashes found in the fuzzing phase. Once the targeted crash is found, we can obtain the TTE by measuring the time elapsed from the start of fuzzing phase. However, only focusing on the fuzzing phase can be misleading when evaluating the overall effectiveness of a directed fuzzer. Recall from Algorithm 1 that directed fuzzers require a preprocessing step before the start of actual fuzzing. In this step, most of the directed grey-box fuzzers run static analysis on the target program [Böhme et al. 2017; Canakci et al. 2022; Chen et al. 2018; Du et al. 2022; Huang et al. 2022; Kim et al. 2023b; Lee et al. 2021; Luo et al. 2023; Meng et al. 2022; Nguyen et al. 2020; Österlund et al. 2020; Srivastava et al. 2022].

The time spent on static analysis varies significantly among fuzzers[2] and even can be longer than the fuzzing time needed to expose the target bug. Unfortunately, such static analysis time is often overlooked in the papers, with only a few exceptions [Kim et al. 2023b; Nguyen et al. 2020; Shah et al. 2022].

Table 7 presents the static analysis time of the directed fuzzers we used for our experiment on our 12 target bugs. Note that the static analysis time of WindRanger is negligible and omitted here. The table indicates that static analysis time can have a significant impact on the performance comparison between fuzzers. For instance, Table 8 specifically shows the two cases comparing two fuzzers, AFLGo and Beacon. In both cases, AFLGo and Beacon perform similarly in terms of pure fuzzing time. However, AFLGo takes significantly less time to statically analyze the target program in the case of CVE-2016-4489, thus, outperforming Beacon when considering overall time.

---

[2]In contrast, the instrumentation times are mostly consistent across fuzzers.

Table 7. Static analysis time on each target. The unit is in seconds.

Program	CVE	AFLGo	Beacon	SelectFuzz	DAFL
cxxfilt	2016-4487	205	2676	173	16
	2016-4489	203	4633	173	17
	2016-4490	205	3900	173	16
	2016-4491	207	4348	173	16
	2016-4492	208	4355	173	17
	2016-6131	210	3125	174	16
swftophp	2016-9827	342	7	75	7
	2016-9829	331	9	76	7
	2016-9831	337	10	76	7
	2017-9988	334	12	77	7
	2017-11728	345	7	77	7
	2017-11729	345	14	76	7

Table 8. Performance comparison with and without consideration of static analysis time. T_f and T_{sa} denote pure fuzzing time and static analysis time, respectively. Note that both are the median of 160 repetitions.

	T_f (s)		$T_f + T_{sa}$ (s)	
Target CVE	AFLGo	Beacon	AFLGo	Beacon
2016-4489	226	248	429	4881
2016-9831	264	254	601	264

Interestingly, the vice versa holds true in the case of CVE-2016-9831, where Beacon outperforms AFLGo when considering overall time.

One may argue that static analysis time is not important because it is a one-time cost. However, a program is often frequently updated to fix bugs or add new features. In such cases, static analysis must be performed again to obtain accurate guidance for fuzzing. Therefore, static analysis time is not always a one-time cost. Furthermore, our experimental results indicate that static analysis time has a significant impact on the overall performance of fuzzers.

> **P3** *Lessons learned.* Static analysis time was often overlooked in previous work, yet has a significant impact on the overall performance of fuzzers. We suggest that directed fuzzing papers should include static analysis time in the report to provide readers with comprehensive information about the usefulness of each fuzzer.

7 REPETITION AND STATISTICAL TEST

Previous sections have discussed the measurement of the TTE from a single run of fuzzer. However, due to the random nature of fuzzing, it is necessary to run each fuzzing session multiple times. In this section, we explore whether previous papers have properly conducted such repetitions and analyzed the results.

Table 9. The minimum, maximum, and median TTE of each directed fuzzer, in seconds. If a fuzzer failed to find the target bug, we mark the maximum TTE as timeout (T.O.) and report the number of such timeouts in parentheses. If the timeout occurred for more than half of the repetitions, we mark the median TTE as not available (N.A.). Note that AFLGo failed to run with the target line for CVE-2016-4487.

Target CVE		AFLGo	Beacon	WindRanger	SelectFuzz	DAFL
2016-4487	min	-	11	30	7	7
	max	-	653	3,321	662	230
	median	-	116	474	102	43
2016-4489	min	19	22	28	13	26
	max	1,382	2,185	816	594	535
	median	226	248	98	154	222
2016-4490	min	6	6	16	9	5
	max	242	242	676	175	39
	median	69	62	154	48	13
2016-4491	min	666	850	498	646	33
	max	T.O. (2)	31,462	T.O. (1)	T.O. (2)	T.O. (89)
	median	1,371	1,393	3,111	1,754	N.A.
2016-4492	min	5	12	40	6	36
	max	T.O. (2)	T.O. (1)	T.O. (2)	11,442	T.O. (1)
	median	373	333	2,460	432	787
2016-6131	min	N.A.	510	N.A.	N.A.	4,825
	max	T.O. (160)	T.O. (154)	T.O. (160)	T.O. (160)	T.O. (159)
	median	N.A.	N.A.	N.A.	N.A.	N.A.
2016-9827	min	49	64	13	48	41
	max	T.O. (3)	T.O. (2)	3,112	3,756	9,079
	median	1,571	1,282	555	664	240
2016-9829	min	65	370	251	98	22
	max	T.O. (111)	T.O. (110)	T.O. (19)	T.O. (37)	T.O. (46)
	median	N.A.	N.A.	7,876	17,159	335
2016-9831	min	7	21	16	9	19
	max	9,409	4,377	1,888	1,764	2,562
	median	264	254	199	297	126
2017-9988	min	30	76	20	39	9
	max	T.O. (1)	T.O. (1)	3,762	24,912	T.O. (17)
	median	1,066	1,217	779	2,791	703
2017-11728	min	452	949	98	272	33
	max	T.O. (79)	T.O. (85)	30,013	T.O. (74)	T.O. (7)
	median	66,268	N.A.	1,231	53,877	282
2017-11729	min	102	219	13	39	10
	max	23,936	T.O. (1)	2,068	T.O. (17)	2,209
	median	2,458	2,759	126	2,044	69

7.1 Repetition of Experiment

It is well known that the performance of a fuzzer can vary from run to run, necessitating the repetition of the experiment [Klees et al. 2018]. Accordingly, previous papers on directed fuzzing have run each fuzzer multiple times and reported the median TTE. However, there has not been any study on the number of repetitions required to obtain a reliable conclusion about the performance
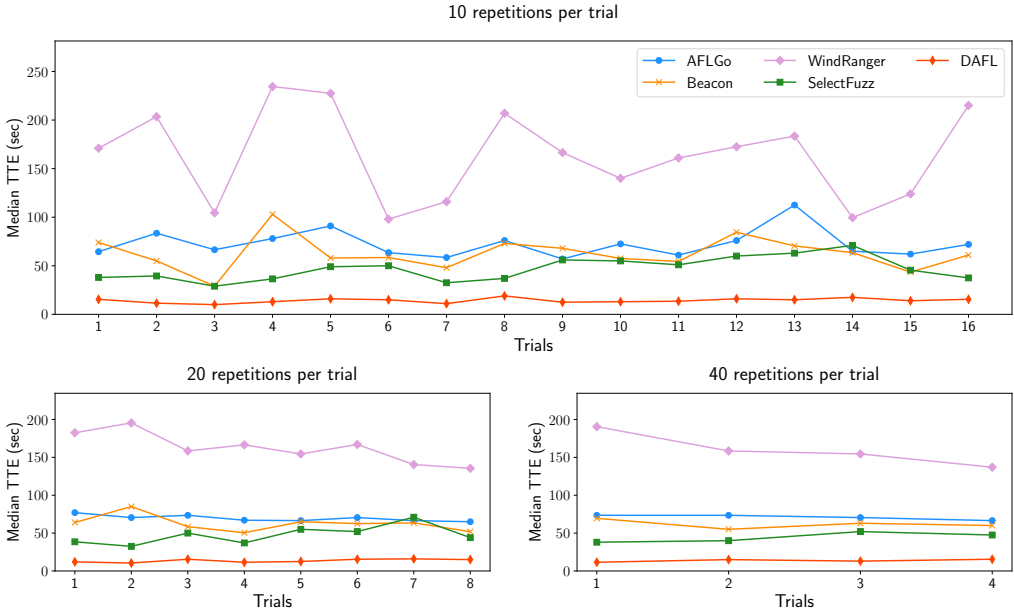
Fig. 6. The impact of repetition count on the stability of median TTEs for CVE-2016-4490. Each line plots how the median TTE changes over the trials. The three graphs are obtained by using different repetition counts (10, 20, or 40) per each trial.

of directed fuzzers. Our survey result in Table 1 shows that the number of repetitions employed in the papers was 16 on average. Moreover, half of the papers performed 10 or fewer times of repetitions.

However, our experiment indicates that such a small number of repetitions is not enough to mitigate the randomness in directed fuzzing. First, we summarize our overall experimental results with 12 target bugs in Table 9. We report the minimum, maximum, and median TTE computed over 160 repetitions. Note that average and standard deviation cannot be computed if there is any timeout (i.e., fuzzer failing to expose the target bug within the given time limit) among the repetitions. According to the table, timeouts are common even for cases where the median TTE is less than an hour. This implies the intense randomness of directed fuzzing.

We further investigate the result of CVE-2016-4490 to confirm that the current practices of directed fuzzing evaluation indeed involve an insufficient number of repetitions. To avoid extreme cases, we deliberately chose a target bug where no timeout was observed during the fuzzing. In Figure 6, we present three graphs that partition the 160 repetitions into 16, 8, and 4 groups respectively. That is, we reinterpreted our experiment into repeated trials, where each trial consists of 10, 20, or 40 repetitions. Despite the fact that we have chosen a moderate case of CVE-2016-4490, the median TTE of each trial varies significantly when the size of the group is 10 or 20. Moreover, the relative ranking between AFLGo, Beacon, and SelectFuzz changes drastically from trial to trial. While the degree of fluctuation will differ from bug to bug, it is obvious that many directed fuzzers have chosen repetition counts that are too small.

(a) Y-axis plotted in linear scale
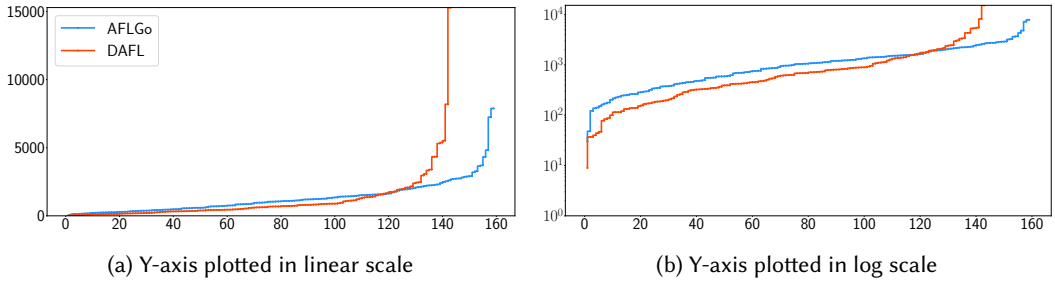

(b) Y-axis plotted in log scale

Fig. 7. Cactus plot that compares the performance of AFLGo and DAFL on CVE-2017-9988. X-axis denotes the number of repetitions, and Y-axis denotes the TTE in seconds.

> **P4** *Lessons learned.* Due to the randomness of fuzzing, directed fuzzing papers perform a repetition of experiments to mitigate the randomness. However, we found that the number of repetitions performed in previous papers is not quite enough to mitigate the randomness. We suggest that directed fuzzing papers should employ a higher number of repetitions (>20) to obtain a reliable conclusion.

7.2 Statistical Test and Presentation

According to Klees et al. [2018], comparing the median or average is not enough for evaluating the performance of fuzzers. Nowadays, the MWU test is taken as a *de facto* standard for interpreting repeated fuzzing experiments [Arcuri and Briand 2014; Klees et al. 2018]. Previous work on directed fuzzing also follows this practice and reports the p-value obtained from the MWU test [Böhme et al. 2017; Du et al. 2022; Huang et al. 2022; Luo et al. 2023; Meng et al. 2022; Österlund et al. 2020; Shah et al. 2022].

Unfortunately, it is often unspecified how the MWU test was performed in the presence of timeouts. One possible naive approach would be using the timeout parameter (e.g., 24 hours in our experiment) as the TTE of the repetition that failed to find the target bug [Böhme et al. 2017]. However, this can lead to an incorrect result because it uses TTE values that are smaller than actual observations. Another possible approach is excluding the timed-out repetitions from samples. However, this may also bias the result because the timeout itself is already a meaningful observation. Either possible approach can lead to an erroneous conclusion because the MWU test is not designed to handle censored data. Therefore, we suggest that MWU tests should be performed only when there is no timeout. Surprisingly, most of the directed fuzzing papers that employed MWU tests do not specify how timed-out cases were handled [Du et al. 2022; Huang et al. 2022; Luo et al. 2023; Österlund et al. 2020; Shah et al. 2022].

As an alternative approach, we propose to use the standards from *survival analysis* [Klein and Moeschberger 2003]. Survival analysis aims at estimating the time it takes for a specific event to occur. In directed fuzzing, the exposure of a target bug is the event of our interest.

First, the *cactus plot* is a popular presentation method in survival analysis that plots the time of events in ascending order. Figure 7 is a cactus plot that compares DAFL and AFLGo on CVE-2017-9988. The cactus plot presents rich information about the distribution of TTEs that cannot be captured in the median or p-value. Although DAFL has a smaller median TTE than AFLGo, it is more susceptible to randomness at the same time. In particular, DAFL resulted in a timeout of 24 hours in 17 out of 160 repetitions, whereas the AFLGo only timed out once. Despite the wide usage

of cactus plots in research on SMT solvers [Brain et al. 2017] or program synthesis [Lee 2021; Yoon et al. 2023], none of the directed fuzzing papers have used it. We argue that cactus plots should be adopted more widely in directed fuzzing evaluation.

For statistical tests, the *log-rank test* is used in survival analysis. Log-rank tests can accept censored data, such as timed-out cases in directed fuzzing. It is important to note that the log-rank test only supports the null hypothesis that two directed fuzzers are equally effective in discovering the target bug. Therefore, cactus plots must be provided along with the result of the log-rank test, if one wants to claim that one tool performs better than the other.

Interestingly, the log-rank test and MWU test yield totally different conclusions from the data plotted in Figure 7. Log-rank test returns $p > 0.5$ for its null hypothesis, implying that there is no significant difference between the effectiveness of DAFL and AFLGo. Meanwhile, if we insist on performing the MWU test by using 24 hours as a TTE value for timed-out cases, it returns $p = 0.014$ for the null hypothesis that AFLGo outperforms DAFL. In general, it is not trivial to determine if one statistical test is superior over another. However, in the specific context of directed fuzzing evaluation, we argue that using a log-rank test and cactus plot is more appropriate for handling the existence of timeouts.

> **P5** *Lessons learned.* Directed fuzzing papers employed statistical tests to interpret the result from multiple repetitions of fuzzing experiments. However, the most commonly used statistical test, the MWU test, is not suitable in the presence of timeouts. We suggest that directed fuzzing papers should jointly use survival analysis, such as log-rank test and cactus plot, to present the result.

8 RELATED WORK

The key idea of directed grey-box fuzzing (DGF) is to estimate the usefulness of generated test cases in terms of reaching the given target location. AFLGo [Böhme et al. 2017] is the first DGF that employs a CFG-based distance metric to evaluate the proximity of a seed to the target location. Based on this estimation, AFLGo allocates more fuzzing resources to seeds that are considered more useful in terms of reaching the target location. Since AFLGo, there have been numerous attempts to effectively guide the fuzzers using sophisticated distance metrics [Canakci et al. 2022; Chen et al. 2018; Du et al. 2022; Lee et al. 2021; Nguyen et al. 2020; Österlund et al. 2020], static analysis on program semantics [Huang et al. 2022; Kim et al. 2023b; Luo et al. 2023; Meng et al. 2022; Srivastava et al. 2022], or search algorithms [Shah et al. 2022]. While DGF generally targets at most a few targets, Savior [Chen et al. 2020] and FishFuzz [Zheng et al. 2023] bridge the gap between DGF and general fuzzing by targeting all vulnerable locations in a program, which usually scale up to thousands.

Due to the popularity of fuzzing, there have been several meta-science studies and surveys on fuzzing [Klees et al. 2018; Mallissery and Wu 2023; Manès et al. 2021; Wang et al. 2020]. Especially, Klees et al. [2018] performed a thorough study on how to evaluate fuzzers, focusing on the proper experimental configurations for fuzzing. Meanwhile, we focus on the unique characteristics of directed fuzzing and discuss the pitfalls specific to evaluating directed fuzzing. Wang et al. [2020] performed a survey on directed fuzzing, as Manès et al. [2021] did on fuzzing in general. Wang et al. [2020] categorized previous work based on the fuzzing methodology, but their discussion on directed fuzzing evaluation was limited. Our work is unique in that we provide a detailed discussion on the evaluation of directed fuzzing.

There exists previous work on crash triage as well [Arnold et al. 2007; Bartz et al. 2008; Kim et al. 2011; Runeson et al. 2007; Wang et al. 2008]. The triage process may utilize the execution trace of

the crashing program [Arnold et al. 2007; Bartz et al. 2008], or utilize the bug report written by the users [Runeson et al. 2007; Wang et al. 2008]. The main purpose of previous work on crash triage was to identify the severity of a crash or group similar crashes to help developers fix them. In contrast, the crash triage performed in directed fuzzing evaluation aims to identify whether a given crash corresponds to the target bug or not. Thus, more precise decision is required in the crash triage for directed fuzzing evaluation.

Various benchmarks have been proposed to evaluate fuzzers, ranging from a collection of real-world bugs [Google 2021; Hazimeh et al. 2021; Metzman et al. 2021] to a framework for injecting synthetic bugs [Dolan-Gavitt et al. 2016; Görz et al. 2023; Lee et al. 2022; Pewny and Holz 2016; Roy et al. 2018; Zhang et al. 2022]. However, a benchmark for directed fuzzing evaluation should not only contain a target bug description. Ideally, it should also contain the information on the specific target site and concrete criteria for crash triage. Only after so, the evaluation is truly reproducible with the benchmark.

9 CONCLUSION

In this paper, we address the potential pitfalls in the evaluation of directed fuzzers and propose guidelines to mitigate them. First, target bug specification and crash triaging are two important parts of the evaluation, yet not many papers have provided enough details on how they were performed. The ideal solution for target bug specification and crash triaging is clear, provide the root cause of the bug to specify the target and use a benchmark with ground truth bug for the crash triaging. However, in order to achieve this, we need more discussion and study on the evaluation process by open-sourcing the evaluation process. Second, we have discussed additional issues specific to the evaluation of directed fuzzers. For directed fuzzing evaluation, we must consider the static analysis time to correctly assess the usefulness of the fuzzer, set the number of repetitions robust to the randomness of the fuzzer, deal with timeout with the right statistical test. All of our claims are supported by empirical studies with a total computational effort equivalent to 30 CPU-years. We hope that our study will help the research community to conduct more transparent and reproducible evaluations of directed fuzzers.

10 DATA AVAILABILITY

We open-source the scripts and the data used in this paper on Zenodo [Kim et al. 2024].

REFERENCES

Andrea Arcuri and Lionel C. Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification & Reliability* 24, 3 (2014), 219–250.

Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA.* 1–10.

Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques, SysML 2008, December 11, 2008, San Diego, CA, USA, Proceedings.*

Trail Of Bits. 2017. CGC Challenge Dataset. https://github.com/trailofbits/cb_multios

Marcel Böhme. 2016. GCC Bug #70926. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70926.

Marcel Böhme. 2016a. Patch for CVE-2016-4489. https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=59dad006fa31fe3.

Marcel Böhme. 2016b. Patch for CVE-2016-4492 and CVE-2016-4493. https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=03ef0c6c55ab810.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2329–2344.

Martin Brain, James H Davenport, and Alberto Griggio. 2017. Benchmarking Solvers, SAT-style. In *Proceedings of ISSAC Workshop on Satisfiability Checking and Symbolic Computation*.

Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. 2022. TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. In *Proceedings of the Asia Conference on Computer and Communications Security*. 561–573.

Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2095–2108.

Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1580–1596.

Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proceedings of the International Conference on Software Engineering*. 144–155.

Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the IEEE Symposium on Security and Privacy*. 110–121.

Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *Proceedings of the IEEE Symposium on Security and Privacy*. 110–121.

Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *Proceedings of the International Conference on Software Engineering*. 2440–2451.

Google. 2021. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.

Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. 2023. Systematic Assessment of Fuzzers using Mutation Analysis. In *Proceedings of the USENIX Security Symposium*. 4535–4552.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*. 81–82.

Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the IEEE Symposium on Security and Privacy*. 36–50.

Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the International Conference on Dependable Systems and Networks*. 486–493.

Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023a. DAFL Artifact GitHub Repository. https://github.com/prosyslab/DAFL-artifact.

Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023b. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *Proceedings of the USENIX Security Symposium*. 4931–4948.

Tae Eun Kim, Jaeseung Choi, Seongjae Im, Kihong Heo, and Sang Kil Cha. 2024. Reproduction Package for the FSE 2024 Article 'Evaluating Directed Fuzzers: Are We Heading in the Right Direction?'. https://doi.org/10.5281/zenodo.10669580.

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2123–2138.

John P Klein and Melvin L Moeschberger. 2003. *Survival analysis: techniques for censored and truncated data*. Springer.

Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *Proceedings of the USENIX Security Symposium*. 3559–3576.

Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *Proceedings of the International Conference on Automated Software Engineering*.

Haeun Lee, Hee Dong Yang, Su Geun Ji, and Sang Kil Cha. 2023. On the Effectiveness of Synthetic Benchmarks for Evaluating Directed Grey-box Fuzzers. In *Proceedings of the Asia-Pacific Software Engineering Conference*. 11–20.

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. (2021), 1–28.

Changhua Luo, Wei Meng, and Penghui Li. 2023. SELECTFUZZ: Efficient Directed Fuzzing with Selective Path Exploration. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1050–1064.

Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the Fuzzing Methods: A Comprehensive Survey. *Comput. Surveys* (2023).

Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331.

Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *Proceedings of the International Conference on Software Engineering*. 1343–1355.

Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 1393–1403.

MITRE. 2016a. CVE-2016-4489. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4489.

MITRE. 2016b. CVE-2016-4491. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4491.

MITRE. 2016c. CVE-2016-4492. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4492.

MITRE. 2016d. CVE-2016-9831. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9831.

MITRE. 2019. CVE-2019-9071. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9071.

MITRE. 2023. MITRE CVE Database. https://cve.mitre.org.

Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*. 47–62.

Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-Guided Greybox Fuzzing. In *Proceedings of the USENIX Security Symposium*. 2289–2306.

Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the Annual Computer Security Applications Conference*. 214–225.

Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 224–234.

Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the International Conference on Software Engineering*. 499–510.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*. 309–318.

Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2595–2609.

Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. 2022. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In *Proceedings of the Annual Computer Security Applications Conference*. 388–399.

Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2020. SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *CoRR* (2020). arXiv:2005.11907

Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 497–512.

Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the International Conference on Software Engineering*. 461–470.

Mark Wielaard. 2016a. Initial patch of CVE-2016-4491. https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=a46586c34f32db5.

Mark Wielaard. 2016b. Supplementary patch for CVE-2016-4491. https://gcc.gnu.org/git/?p=gcc.git&a=commit;h=6b086d35b79425d.

Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 1657–1681.

Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *Proceedings of the USENIX Security Symposium*. 3699–3715.

Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *Proceedings of the USENIX Security Symposium*. 1343–1360.

Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *Proceedings of the USENIX Security Symposium*. 2255–2269.