Computers & Security 124 (2023) 102946

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Improving fuzzing assessment methods through the analysis of metrics and experimental conditions

Maialen Eceiza[a,b,*], Jose Luis Flores[a], Mikel Iturbe[b]

[a] Department of Industrial Cybersecurity. IKERLAN Technology Research Center, Basque Research and Technology Alliance (BRTA), Jose Maria Arizmendiarrieta Pasealekua, 2, Arrasate-Mondragon, 20500, Gipuzkoa, Spain
[b] Department of Computing and Electronics, Faculty of Engineering, Mondragon Unibertsitatea, Goiru 2, E-20500 Arrasate-Mondragon, Spain

ARTICLE INFO

Article history:
Received 19 April 2022
Revised 26 August 2022
Accepted 3 October 2022
Available online 10 October 2022

Keywords:
Fuzzing
Evaluation methodology
Security
Software testing
Metrics

ABSTRACT

Fuzzing is nowadays one of the most widely used bug hunting techniques. By automatically generating malformed inputs, fuzzing aims to trigger unwanted behavior on its target. While fuzzing research has matured considerably in the last years, the evaluation and comparison of different fuzzing proposals remain challenging, as no standard set of metrics, data, or experimental conditions exist to allow such observation. This paper aims to fill that gap by proposing a standard set of features to allow such comparison. For that end, it first reviews the existing evaluation methods in the literature and discusses all existing metrics by evaluating seven fuzzers under identical experimental conditions. After examining the obtained results, it recommends a set of practices –particularly on the metrics to be used–, to allow proper comparison between different fuzzing proposals.

1. Introduction

Fuzzing is an automated testing technique that allows vulnerability detection by generating malformed inputs to trigger unwanted behaviors and find bugs against different systems, generally software applications.

Since the inception of the first fuzzer in 1990 by Miller et al. (1990), fuzzing research has evolved vastly. This evolution has occurred mainly in two different directions: On the one hand, fuzzing is being applied to audit targets that are very different, ranging from large-scale distributed applications to low-power embedded devices. On the other hand, fuzzer internals have been in constant improvement, searching for more efficient vulnerability discovery from the initial randomly mutating black-box fuzzers.

These evolution have led to the creation of a plethora of different fuzzers (refer to surveys Chen et al. (2018); Liang et al. (2018a) for a wide exploration of the field). However, even if the field of fuzzer creation has evolved rapidly, research on the means of comparing and evaluating different fuzzers using an objective set of criteria has not evolved at the same pace (Hazimeh et al., 2020).

The lack of standard assessment criteria has led to the issue that it is often not feasible to objectively compare different fuzzers, as experimental conditions or used metrics vary between proposals (Aschermann et al., 2019; Lemieux and Sen, 2017; Liang et al., 2018b; Yue et al., 2019).

This paper aims to fulfill this gap by analyzing existing fuzzing evaluation methods and proposing a standard set of metrics and experimental conditions to be used when evaluating fuzzers. To this end, the current literature has been analyzed to identify the current status of fuzzing evaluation. Particularly, the review focuses on the set of metrics, data, and experimental conditions used in the literature. Next, seven widely used fuzzers are executed and compared under identical experimental conditions to compute the metrics previously identified. Finally, those results are analyzed, and the most relevant metrics are identified, as well as outlining some recommendations when comparing fuzzers. In summary, the main objective of the manuscript is to describe and define an objective and standard evaluation methodology for fuzzers.

The main contributions of the paper are the following:

- Identification and analysis of existing fuzzing evaluation methods to detect the necessary elements for fuzzing algorithms assessment.
- Definition of the experimental environment, where the resources used to perform the experiments and the conditions

* Corresponding author.
E-mail address: meceiza@ikerlan.es (M. Eceiza).

https://doi.org/10.1016/j.cose.2022.102946

under which the tests are carried out are specified. The resources are online available.[1]
- A case study based on the experimentation of seven different widely used fuzzers.
- Experimental identification of the most relevant fuzzing metrics for fuzzer evaluation and comparison.

For this purpose, the paper has been organized as follows: Section 2 analyzes the features of current techniques to evaluate the fuzzing algorithms and the current evaluation methodologies. Section 3 presents the experimental setup: fuzzers, metrics, data, and experimental conditions. Section 4 explains the analysis of the results, and in Section 5, we give some recommendations to assess fuzzers. Finally, Section 6 concludes the paper.

2. The current state of experimental evaluation of fuzzers

Fuzzing is an automated testing technique that allows finding vulnerabilities in different systems. For this end, it generates malformed inputs to break the system and find bugs that trigger such behavior.

A wide variety of fuzzers have been created since 1990, when Professor Miller created the first fuzzer. This fuzzer generated the inputs randomly and was a black-box type fuzzer. Since then, fuzzers have evolved using different techniques such as instrumentation or taint analysis, giving rise to other types of fuzzers: white-box and gray-box.

Black-box fuzzers do not need any feedback or knowledge about the internal workings of the system under test (SUT) to test them (Chen et al., 2018). White-box fuzzers, on the other hand, need complete visibility to operate correctly since they use the additional information to generate new inputs (Liang et al., 2018a). Finally, gray-box fuzzers need to have some knowledge about the SUT, but what they need to know will depend on the technique they use, as gray-box fuzzers may be closer to the black-box or white-box scope (Jun Li and Zhang, 2018). Therefore, there is a considerable variety of fuzzers, each with its own characteristics.

From the conception of the first fuzzing algorithm, different methods have been defined and used to evaluate them. These methods are based on a set of measurements or metrics under a set of experimental conditions that score the performance of the algorithm. The authors of each fuzzer have mainly defined these methods on an ad hoc basis. Due to this lack of consensus in the evaluation of fuzzers, there are many experimental conditions and metrics used in the literature. This is not a purely scientific issue, as it also affects practitioners by complicating the choice of a fuzzer in constrained scenarios where it is not feasible to evaluate a set of fuzzers to choose the best performing one. It is necessary to define a set of metrics known to best represent the actual performance of the fuzzers.

Moreover, in an attempt to solve this issue, several specific automated assessment frameworks have been developed to automate this process. In the following sections, we will describe metrics, experimental conditions, and the automation frameworks found in fuzzing-related literature.

2.1. Fuzzing metrics

In order to measure the performance of fuzzing algorithms, a wide range of metrics have been defined in the literature. This set of metrics is shown and compared in Table 1. Among the thirty-six analyzed papers, metrics can be categorized into three groups:

- Bug detection: these metrics are aimed to account for the number of detected bugs. In the analyzed literature, most works consider a bug as any unwanted behavior in a system or program (Chen et al., 2018). Such unwanted behavior does not necessarily mean that a crash happens, nor the exploitable nature of the bug. Measuring bug exploitability may require further (manual) analysis (Hazimeh et al., 2020) which would require an additional step to fuzzing itself. Moreover, any seemingly minor bug can be the cause of a future larger issue in some systems (particularly resource-constrained ones) because they do not react instantly to inputs (Muench et al., 2018). Therefore, in this paper, we will be using the more generic definition of bug –prevalent in the literature– instead of only considering the directly exploitable crashes. While the main objective of fuzzing is to detect bugs, the literature shows different manners to measure the performance of a fuzzer in bug finding. For instance, it is possible to count the number of found bugs that other fuzzers have failed to detect (*B.2*) or the total number of target crashes while testing (*B.3*).
- Coverage: metrics belonging to this category aim to quantify the percentage of the code that has been executed at least once during fuzz testing. These metrics can only be measured in scenarios where it is possible to instrument the source code in such a way that it is possible to detect the execution of the program with different levels of granularity, such as lines, branches, paths, or even functions. In black-box based approaches is not possible to measure coverage metrics, because there are not resources and/or access to measure them. However, in white-box and gray-box based approaches it is usually possible to measure them.
- Performance: this group of metrics measures fuzzer performance in terms not directly related to the previous two groups. These metrics include the number of tests or runs that it can execute within a specific time frame, the time needed to find the first bug, or the execution speed.

Analyzing Table 1, it can be observed that the most used metrics are the ones related to bugs. Only twelve of the proposals do not explicitly count the number of found bugs. Metrics related to coverage come second. Nineteen proposals use at least one coverage metric, while six measure over one. This is because of the importance of coverage to assess the quantity of executed code, as it is not possible to detect bugs in its unexecuted parts. Finally, it is shown that performance-related metrics are used in eight papers. The number of tests metric (*P.1*) is the only metric used more than once.

2.2. Experimental conditions

Apart from the metrics used, other factors can strongly influence the outcome of the assessments. As such, a set of experimental conditions must be defined to ensure that all experiments are performed under the same conditions. This equality in the experimental conditions allows a fair comparison between different experiments. In the case of fuzz testing, experimental factors can be listed as follows:

- Datasets: In the case of fuzzers, data corresponds to the software tested in the fuzzing session. While any program can be tested, its nature and characteristics, such as its algorithmic complexity, will dictate the difficulty of finding bugs. It will be easier to find bugs in simple, known-to-be-buggy software, whereas it will be more challenging in robust programs with complex algorithmic. Therefore, the results of the fuzzer are highly linked to the system under test. Therefore, to evaluate and compare fuzzers, it is necessary to use a common set of applications.
- Repetitions: The stochastic nature of fuzzers implies that results can vary from one execution to another. That means that each

[1] https://www.github.com/Mai722/FuzzingResults.

Table 1
Metrics set in fuzzing literature. (These reference cited in this table (Godefroid et al., 2008; Grieco et al., 2017; Jitsunari and Arahori, 2019; McNally et al., 2012; Ognawala et al., 2017; Takanen et al., 2008; IFuzzer, 2016; Wang and Cartmell, 1997; Xie et al., 2019; Zhao et al., 2011)).

Category	Metrics	Definition	AFLfast [11]	AFLgo [12]	Angora [13]	BFF [14, 15]	Buzzfuzz [16]	Deephunter [17]	Driller [18]	Enfuzz [19]	Exploitmeter [20]	H-fuzzing [21]	Ifuzzer [22]	Intriguer [23]	IoTFuzzer [24]	kAFL [25]	KLEE [26]	Matryoshka [27]	PAFL [7]	Quickfuzz [28]	SAGE [29]	Skyfire [30]	SLF [31]	Steelix [32]	T-fuzz [33]	Taintscope [34]	V-fuzz [35]	VUzzer [36]	Fuzzbench [37]	Magma [4]	Unifuzz [38]	Klees et al. [39]	Fell [40]	Jitsunari and Arahori [41]	Ognawala et al. [42]	Takanen et al. [43]	Macnally et al. [44]	Muench et al. [10]
BUGS	B.1 Bugs detected (total)	Number of bugs detected during the fuzzing process.	■	□	■	■	■	■	□	■	■	□	■	■	□	□	□	■	■	■	□	■	□	■	■	■	■	■	□	■	■	■	□	□	□	□	□	□
	B.2 Distinct bugs	Number of bugs only detected by a single fuzzer.	□	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
	B.3 Number of crashes	Number of crashes occurred during the fuzzing process.	□	□	□	□	□	□	■	□	■	□	□	□	■	□	□	□	■	□	■	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□
	B.4 Vulnerability detection speed	The relation between the number of bugs detected and the execution time.	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
	B.5 Stability of finding bugs	It measures the variation of the detected bugs between repetitions.	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
COVERAGE	C.1 Line coverage	The number or percentage of executed lines during the fuzzing session.	■	□	■	□	□	□	□	□	□	□	□	■	□	□	■	■	□	□	□	□	□	■	□	□	□	□	■	□	■	■	■	■	□	□	□	□
	C.2 Branch coverage	The number or percentage of executed executed branches during the fuzzing session.	□	□	■	□	□	□	□	■	□	■	□	■	□	□	□	■	■	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□
	C.3 Function coverage	The number or percentage of executed functions during the fuzzing session.	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	■	□	■	□	□	□	□	□	□	□	□	□	■	■	□	□	□
PERFORMANCE	P.1 Number of tests	The number of generated and executed in a fuzzing session.	■	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□
	P.2 Density	The relation between the bugs found and the number of executed test cases.	□	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
	P.3 Execution speed (discrete)	The number of executed tests per second.	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
	P.4 Execution speed (total)	The relation of the total executions and the testing time.	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
	P.5 Time to crash	The time needed until the first crash.	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
	P.6 Time to bug	The time needed to find the first bug.	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□

Table 2
Testing time used in fuzzing literature.

	2h	5h	6h	8h	12h	24h	45h	4d14h	5d	10d	2w	864h	1000h	Long. Unexpecified
AFL(Rawat et al., 2017)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
AFLfast (Klees et al., 2018)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
AFLgo(Klees et al., 2018)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
Angora(Chen and Chen, 2018)	□	■	□	□	□	□	□	□	□	□	□	□	□	□
BFF(Cha et al., 2015)	□	□	□	□	□	□	□	□	□	□	□	□	■	□
BUZZFUZZ(Ganesh et al., 2009)	□	□	□	□	■	□	□	□	□	□	□	□	□	□
Dowser(Istvan et al., 2018)	□	□	□	■	□	□	□	□	□	□	□	□	□	□
Driller(Stephens et al., 2016)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
FIE(Davidson et al., 2013)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
FuzzSim(Woo et al., 2013)	□	□	□	□	□	□	□	□	□	■	□	□	□	□
GWF (Godefroid et al., 2008)	■	□	□	□	□	□	□	□	□	□	□	□	□	□
IoTFuzzer(Chen et al., 2020)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
kAFL(Schumilo et al., 2017)	□	□	□	□	□	□	□	■	□	□	□	□	□	□
Learn&Fuzz(Godefroid et al., 2017)	□	□	□	□	□	□	■	□	□	□	□	□	□	□
MoWF(Pham and Böhme, 2016)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
PAFL(Liang et al., 2018b)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
Peach(Luo et al., 2020)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
RedQueen(Aschermann et al., 2019)	□	■	□	□	□	□	□	□	□	□	□	□	□	□
S2E(Chipounov et al., 2011)	□	□	■	□	□	□	□	□	□	□	□	□	□	□
SAGE (Godefroid et al., 2012)	□	□	□	□	□	□	□	□	□	□	□	□	□	■
Skyfire(Klees et al., 2018)	□	□	□	□	□	□	□	□	□	□	□	□	□	■
SLF(You et al., 2019)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
SmartFuzz (Molnar et al., 2009)	□	□	□	□	□	□	□	□	□	□	□	■	□	□
Steelix(Li et al., 2017)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
SYMFUZZ(Cha et al., 2015)	□	□	□	□	□	□	□	□	□	□	□	□	■	□
Syzkaller(Li and Chen, 2019)	□	□	□	□	□	□	□	□	□	□	■	□	□	□
T-fuzz(Peng et al., 2018)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
V-fuzz(Li et al., 2019)	□	□	□	□	□	■	□	□	□	□	□	□	□	□
zzuf(Lin et al., 2016)	□	□	□	□	□	□	□	□	■	□	□	□	□	□
TOTAL	1	2	1	1	1	13	1	1	1	1	1	1	2	2

repetition will yield different results, even if the used fuzzer, the tested code, and all other conditions are kept the same. Therefore, to analyze the full potential of a fuzzer, it is necessary to execute more than one fuzzing run. However, in most of the analyzed papers, this number is not specified, nor whether more than one repetition is performed.

- Testing time: This factor refers to the duration of the fuzzing session. That is, it ranges from the moment the first input is generated until the last response is collected. Table 2 lists different testing times found across different fuzzing proposals, ranging from 2 to 1000 hours. However, 24-hour long tests are the most popular option, while ten of them set a longer testing time, and six of the papers specify it as less than twelve hours.

2.3. Automated fuzzing assessment frameworks

The complex landscape of fuzzing is related to the difficulties of fuzzing evaluation, and the lack of proper tools has motivated the development of evaluation frameworks to facilitate and automate the evaluation of fuzzers:

- MAGMA evaluation framework (Hazimeh et al., 2020): published at the end of 2020, this evaluation framework contains seven real-world targets. Regarding the experimental conditions, Magma offers two possibilities for choosing the testing time, the runs could be 24 hours or 7 days, which are repeated ten times. Regarding the metrics, it measures the number of bugs and the time to bug (Hazimeh et al., 0000).
- FuzzBench evaluation framework (Metzman et al., 2020): a free service and application launched by Google, FuzzBench permits to evaluate fuzzers automatically, either by using the service itself or by running it locally It includes 24 real-world programs to test fuzzers against them. The executions are 24 hours long, and 20 repetitions are made. The results are returned in a report where the coverage is analyzed.

However, in most published articles regarding fuzzing, authors design their ad-hoc framework. This has lead to the proliferation of a wide range of experimental conditions (see Table 2) and metrics (see Table 1). In each of the papers, they with no additional information to justify their choice. In contrast to previous cases, there is more consensus when considering the testing time because most authors set it at 24 hours.

3. Experimental set up

In this section, we describe the experimental environment used to assess different fuzzers in a fair and repeatable manner, as well as to compare them and identify the necessary standard set of experimental aspects for a fair comparison. To reach this aim, several aspects need to be defined:

1. Candidate fuzzing algorithms: To conduct the experimentation, we have selected seven fuzzers, which, although they are all graybox ones, they use different techniques to generate the test cases. This way, we will be able to observe which metrics can be evaluated more fairly. Even so, we have also considered that with the selected fuzzers, we could calculate all the metrics mentioned in Table 1, as not all fuzzers are able to yield the information needed to compute the listed metrics.
   - AFLfast (Böhme et al., 2017b): It is a coverage-based graybox fuzzer based on AFL. This fuzzer introduces several significant changes to different parts of AFL. Firstly, the coverage is carried out by exploring the state space of a Markov chain which models the probability of generating an input that exercises a specific path by randomly mutating a previous input that exercises another path. Secondly, new power schedules are introduced with the aim of exercising a larger number of low-frequency paths. Finally, unlike the classical policy of selecting a seed based on the number of times a seed has been fuzzed, AFLfast introduces two new search strategies: *Prioritize small s(i)* focused on selecting those in-

puts such that the number of times that the input has been fuzzed being minimal and *Prioritize small f(i)* which those seeds that the number of generated inputs that exercise a specific path is minimal.

- AFLgo (Böhme et al., 2017a): This fuzzer is the real-world implementation of the *Directed Graybox Fuzzing* (DFG). DFG is a coverage-based graybox fuzzer that, unlike other alternatives, is designed to target specific program locations efficiently. On a high level, the efficient reachability of specific locations is considered an optimization problem. As such, DFG uses a specific meta-heuristic called *Simulated Annealing* to minimize the distance of the generated seeds to the targets. In order to carry out the computation of the distance, a value is assigned to each node in the call graph on the function level and in the intra-procedural control-flow graphs on the basic-block level. Based on the assigned values, the function distance is defined as *the number of edges along the shortest path between functions in the call graph* which is used as the basis among other factors to compute the *normalized seed distance*, that is, the distance of a seed to the set of target locations.
- AFLplusplus (Fioraldi et al., 2020): This is a novel fuzzing framework that integrates multiple novel features rather than a new fuzzing algorithm. This framework incorporates novelties in the mechanisms involved in the fuzzing process. Firstly, this framework incorporates different seed scheduling mechanisms such as *fast, coe, explore*,etc. Secondly, new mutators have been incorporated, such as *Input-to-State* mutator and *Mopt* mutator, but it also provides a *Custom Mutator API*. Finally, AFL++ supports several backends for instrumenting code, such as *LLVM, GCC, QEMU, Unicorn, and QBDI*. In addition to this, AFL++ facilitates the researchers the development of new fuzzers by incorporating an extensible API to build upon.
- Fairfuzz (Lemieux and Sen, 2017): It is a coverage-based graybox fuzzer built on top of AFL. This algorithm integrates two main novelties into AFL associated with the selection and mutation stages. Regarding the selection process, Fairfuzz, has a different approach than AFL as it never removes inputs from the queue and it also replaces `isWorthFuzzing` with a function called `HitsRareBranch` which returns true if the input hits a *rare* branch. A branch is considered rare if it has been hit by a number of inputs less than or equal to a dynamically chosen threshold. Regarding the mutation process, this is carried out in two stages: first a *branch mask* is calculated, which designates at which positions in the input can bytes be (1) overwritten, (2) deleted, or (3) inserted based on the coverage information; secondly, the mutation is performed at a position if the branch mask indicates the resulting input can still hit the target branch.
- LearnAFL (Yue et al., 2019): It is a coverage-based graybox fuzzer based on AFL that is characterized by the fact that it does not require any prior knowledge of the application or input format. This fuzzer is able to learn partial format knowledge of some paths by means of the analysis of the test cases that traverse the paths. In order to accomplish this task, two algorithms are proposed: a first 1 aimed to generate the format or *enhanced expression of magic bytes* which is based on the longest substring searching algorithm, and a second 1 aimed to improve the mutation process with the format knowledge *assistant mutation algorithm*.
- MoPT (Lyu et al., 2019): It is a coverage-based graybox fuzzer based on AFL that improves the process by introducing a novel mutation scheduler algorithm. MOpt aims to choose the next optimal mutation operator by finding the optimal probability distribution of mutation operators. This search is carried out by optimizing the optimal probability of each operator and then optimizing the global optimal probability distribution of mutation operators. In order to accomplish this task, the *Particle Swarm Optimization (PSO)* is leveraged to find the optimal distribution.
- Superion (Wang et al., 2019): It is a grammar-aware coverage-based graybox fuzzer approach that processes structured inputs. This fuzzer takes as input the target program and the grammar of the test input and parses each test input into an abstract syntax tree (AST). Using these ASTs the fuzzer implements two novel strategies for improving the fuzzing. Firstly, a strategy allows trimming test inputs while maintaining the valid input structure. Secondly, two grammar-aware mutation strategies, a first strategy dictionary-based to insert and overwrite tokens in a grammar-aware manner, and a tree-based mutation strategy that replaces one subtree in the AST of a test input. This fuzzer has been implemented as an extension to AFL.

2. Target programs: this dataset is one of the critical factors when evaluating a candidate algorithm. In this sense, we have included a total of 15 real-world programs that are based on varying types of input: image, network, text, audio, XML, or binary (see Table 3). We have based our target selection by choosing the most frequently used targets in the analyzed literature while also trying to keep the input types as varied as possible. This variety allows a fairer comparison by not favoring fuzzers that might be more proficient in generating inputs of a particular type.
3. Experimental conditions: this aspect is especially relevant for having fair and comparable results. To this aim, we have established the following common set of conditions for all combinations of candidate fuzzing algorithms and target programs as follows:

- Number of repetitions: the random nature used by many fuzzers to mutate means that a single-run experiment does not yield significant enough results to assess fuzzer performance. It is necessary to perform several repetitions. However, this aspect is often overlooked in the literature, as most fuzzing contributions do not mention the number of repetitions/runs used to achieve the presented results. Therefore, establishing a canonical repetition number based on the previous consensus remains challenging. Nevertheless, fuzzing frameworks do mention this number: MAGMA (Hazimeh et al., 2020) mentions ten repetitions, and FuzzBench (Metzman et al., 2020), twenty. It is important to note that to obtain statistically significant results, it is necessary to perform at least 15 repetitions (Belle and Millard, 0000). In addition, considering that each extra repetition consumes computing resources, it is advisable to perform the minimum number of required repetitions while, at the same time, maximizing bug hunting. Therefore, we defined fifteen as the most convenient number of repetitions to start and statistically analyze the results.
- Testing time: based on the values in Table 2, the testing time is set at 24 hours.

Regarding existing automatic frameworks, such as MAGMA or FuzzBench, while they provide a standard environment for fuzz testing, they do not provide the necessary means to obtain all the previously covered metrics (Table 1). They are, thus, not suitable for the task at hand.

Concerning the technical setup, all combinations of fuzzers and target programs have been considered under the previously defined experimental conditions. In order to facilitate experimentation and reproducibility, the candidate fuzzers have been deployed and executed in Docker containers, and a Docker image has

Table 3
Real targets.

Code	Version	Type	Works where it is used as target
libjpeg-turbo: jpegtran (lib, 0000)	1.5.2	IMAGE	Chen et al. (2019b); Jung et al. (2019); Lemieux and Sen (2017); Metzman et al. (2020); Rawat et al. (2017); Rebert et al. (2003); Yue et al. (2019)
libpng: pngtest (lib, 0000)	1.2.45	IMAGE	Chen et al. (2019b); Hazimeh et al. (2020); Jung et al. (2019); Lemieux and Sen (2017); Li et al. (2017); Metzman et al. (2020); Peng et al. (2018); Rawat et al. (2017); Yue et al. (2019)
tcpdump: tcpdump (tcp, 0000)	4.9.0	NETWORK	Chen et al. (2019a); Lemieux and Sen (2017); Li et al. (2017, 2020); Metzman et al. (2020); Rawat et al. (2017); Yue et al. (2019)
pcre2: pcre2test (pcr, 0000)	10.19	TEXT	Chen et al. (2019b); Jung et al. (2019); Lia (2021); Liang et al. (2018b)
sqlite3: sqlite3 (sql, 0000)	3.30.1	TEXT	Chen et al. (2019b); Hazimeh et al. (2020); Li et al. (2020); Metzman et al. (2020)
mp3gain: mp3gain (mp3, 0000)	1.5.2-R2	AUDIO	Li et al. (2020, 2019); Rawat et al. (2017); Rebert et al. (2003)
libtiff: tiffsplit (lib, 0000)	4.0.9	IMAGE	Chen et al. (2019a); Jung et al. (2019); Li et al. (2017, 2020); Peng et al. (2018); Wang et al. (2010); You et al. (2019)
libxml2: xmllint (lib, 0000)	2.7.7	XML	Chen et al. (2019a,b); Hazimeh et al. (2020); Lemieux and Sen (2017); Metzman et al. (2020); Wang et al. (2017)
libxml2: xmlcatalog (lib, 0000)			
xpdf: pdftops (Lopez, 2018)	4.0	TEXT	Li et al. (2020, 2019); Rebert et al. (2003); Yue et al. (2019)
xpdf: pdftotext (Lopez, 2018)			
binutils: objdump (bin, 0000)	2.28	BINARY	Böhme et al. (2017a,b); Chen et al. (2019a); Cho, Mingi (Yonsei University); Kim, Seoyoung (Yonsei University); Kwon (2019); Lemieux and Sen (2017); Li et al. (2020); Yan and Lu (2017); Yue et al. (2019)
binutils: nm-new (bin, 0000)			
binutils: strings (bin, 0000)			
binutils: size (bin, 0000)			

Table 4
A sample that represents the format of the complete.csv.

Algorithm	SUT	Repetition number	Time	Crashes	Hangs	Exec/s
AFLfast	jpegtran	0	6	0	0	421
...	...	...	...	...	...	...
AFLplusplus	string	14	86,167	0	6	1

been created containing a fuzzing algorithm. The execution of the fuzzers in this environment allows using more than one fuzzer simultaneously with a high grade of isolation and without influencing other executions (e.g., other fuzzers or different instances of the same fuzzer running in parallel)

Once the tests are finished, the algorithm itself generates several files reporting the results, allowing analysis of the session:

- `fuzzer_stats.csv`. This file summarizes the statistics of the process, that is, the total time of the process, the speed of execution, and a summary of everything that has been detected.
- `plot_data.csv`. This file contains the value of all variables (time, crashes, hangs, execution speed...) that the fuzzer collects continuously.
- `Crashes`. This directory contains the data patterns that cause a target crash, as well as the signals that have been activated when causing the crash.
- `Hangs`. This directory collects the use cases that cause the target application to hang, as well as the timeouts.
- `Queue`. This is a directory containing the collection of all test cases.

After the experimentation, it is necessary to post-process the data to measure some metrics, where the results of each of the repetitions of all the fuzzer-code combinations are stored in a CSV file. First, a file collects the results obtained directly from the fuzzers (`complete.csv`), and its structure can be seen in Table 4. while another file gathers the coverage data (`covresults.csv`), which is structured as shown in Table 5.

4. Analysis of the results

The objective of the analysis is to determine which metrics and experimental conditions are valid to assess a fuzzing algorithm. For this purpose, in this section, we have analyzed a set of metrics for each of the groups that have been categorized previously in Table 1.

4.1. Bugs

With five different existing metrics in the literature, bug-related metrics remain important as they are directly related to the main goal of a fuzzer, that is, to find bugs. This section analyzes the bug-related metrics based on the obtained results.

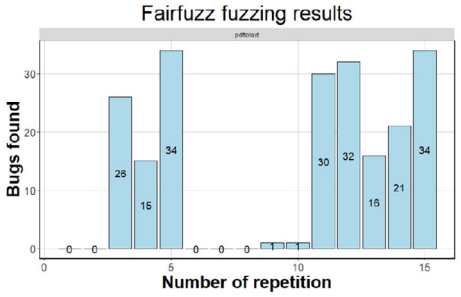
4.1.1. B.1.Bugs detected

This metric rates the performance of the fuzzing algorithm (by counting the number of bugs) independently of the used approach facilitating the use of this metric not only in white-box or gray-box approaches, also in black-box approaches. When measuring the number of bugs, undesired system behavior is measured, i.e., any result that is not as expected is considered a bug.

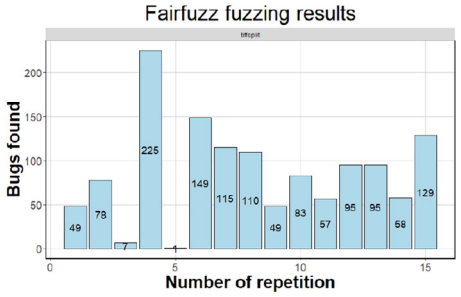
Although this metric rates the algorithm correctly, there are some drawbacks if only one execution is considered. Firstly, an early selection of winners can take place if the only execution is very successful compared with the rest of the alternatives. However, it can be something casual and not a general behavior of the algorithm. Secondly, an early rejection, if the only execution is particularly poor due to the stochastic nature of the algorithm, we can be rejecting a fuzzer that in other executions performs very well.

Table 5
A sample that represents the format of the coverage.csv.

Algorithm number	SUT Line %	Repetition Lines	Branch %	Branches	Function %	Function	Bugs		
Fairfuzz	jpegtran	0	17	2606	23.8	161	25.9	1490	2
...	...	...	...	...	...	...	...	...	...
MoPT	xmlcatalog	14	0.1	9	0.1	2	0.1	1	-



(a) Fairfuzz fuzzing `pdftotext` during 15 repetitions



(b) Fairfuzz fuzzing `tiffsplit` during 15 repetitions

Fig. 1. Results of the number of bugs found during 15 repetitions of Fairfuzz fuzzing `pdftotext` and `tiffsplit` codes.

Thus, several repetitions are required, showing significant differences between repetitions. Fig. 1 shows the results of the fuzzers LearnAFL and Fairfuzz with the SUT `tiffsplit`. This is an example of how the results vary between repetitions (using the same fuzzer with the same code). In each repetition, the number of bugs detected is different. However, in the case of Fairfuzz (see Fig. 1), there is a higher variation because the number of detected bugs ranges from 1 to 225 in different runs.

Thus, it is necessary to set a minimum number of repetitions that can show the potential of the fuzzer. Because, looking at the results in Fig. 1, more specifically Fig. 1, it is possible to see that more than one execution is required for each fuzzer-code combination. On the one hand, considering the fourth and fifth repetitions of the figure, they have significant differences. In the fourth repetition, 225 bugs have been detected, but only one bug is found in the fifth repetition. On the other hand, if repetitions 12 and 13 are considered, would be found 95 bugs in both cases. So, there is a significant difference between repetitions.

Moreover, every single repetition is valuable. The bugs found in each repetition could be different from those in other runs. For instance, if a repetition detects more bugs than others does not directly mean that the one that finds more repetitions detects all the bugs that are found by the other repetitions, including some new bugs as well. Fig. 1 shows that if all the repetitions are not taken into account, it will not be possible to know how the fuzzer works. No error has been detected in five of the repetitions, and in two, only a single bug has been detected. If we only consider these repetitions, it seems that the fuzzer is not able to detect anything. On the other hand, at least fifteen bugs are detected in each of the remaining eight repetitions. Therefore, when running a fuzzer, it is necessary to consider all the repetitions to know how it works.

It is not helpful to measure the mean or the median, as there would be bugs not being considered, as each of the repetitions is highly variable. These variables will be analyzed in more detail in the *Stability finding bugs* metric.

Additionally, testing a single target is not enough (see Fig. 1 and 2) because the number of bugs found depends on the used target system. It is necessary to run different types of programs to assess fuzzers, as it may be the case that a fuzzer works well with a specific type of target but not with the rest. Looking at Fig. 1, both results are of the same fuzzer, but the amount of found bugs are

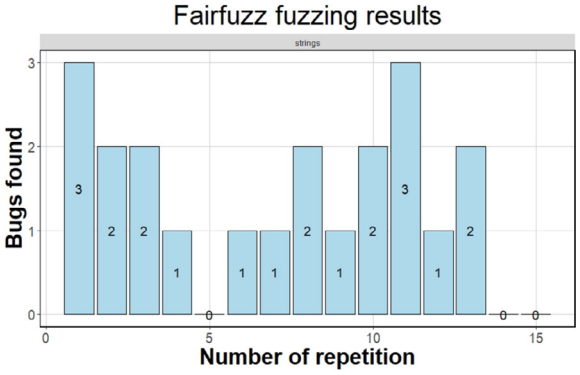


Fig. 2. Fairfuzz fuzzing `strings` during 15 repetitions.

different. For instance, if there are ten bugs in one target and one hundred in another, finding ten bugs means that 100% of the bugs are found in the first case. However, finding ten bugs in the second case means that only 10% of the bugs are detected. Nevertheless, testing different programs makes it possible to see if a fuzzer works better with some targets belonging to a specific domain.

In conclusion, this metric can provide interesting information about the fuzzing performance whenever the experimental conditions are correct, and several repetitions are performed against different targets.

4.1.2. B.2.Distinct bugs

The goal of this metric is to account for the bugs that only a fuzzer can find, that is, it is a differential measure between fuzzers. In order to compute this metric, two fuzzers are required to compute the score.

This metric is problematic from its conception concerning the goal of the fuzzing for the following reasons:

- It is not possible in some cases to establish a clear and fair ranking when several fuzzers are being assessed. A simple example can show this effect, let us assume that we have a set of fuzzers: A, B, C, D, and E, and the bugs are identified by lowercase letters (see Table 6).

Table 6
Distinct bugs.

Fuzzer A	Fuzzer B	Fuzzer C	Fuzzer D	Fuzzer E
a	b	c	d	e
b	c	d	e	f
c	**d**	**e**	**f**	**g**

Table 7
Analysis of distinct bugs fuzzing `strings`.

	AFLfast	AFLgo	Fairfuzz	Superion
Total	15	16	21	6
Distinct bugs between repetitions	4	5	7	6
Distinct bugs	3	-	-	4

Based on the table, if the metric is computed between fuzzers A, B, and E, the result shows that Fuzzer A has only one distinct bug compared with Fuzzer B, but three different bugs compared with Fuzzer E as well as Fuzzer B compared with Fuzzer E. In this situation, if we only consider A and B, these are not the best fuzzers. However, if we consider A and E or B and E, these are the best.

- This metric is focused on the differences and not on the global accountability of bugs that can incorrectly score a set of fuzzers. There can be situations where a fuzzer that finds more bugs globally is poorly scored by this metric because these are not distinct from the alternative. Table 7 shows the real results of the experimentation of the distinct bugs detected with the `strings` target. The fuzzers AFLplusplus, LearnAFL, and MoPT have not detected anything in `strings`, so they are not shown in the table. Regarding the number of bugs detected, the fuzzer that detects the most is Fairfuzz, and the one that detects the least is Superion. However, if we consider the bugs that only one of the fuzzers has found, i.e., distinct bugs, Superion is the one that detects the most, even if it is not the fuzzer with the best total bug detection count.

4.1.3. B.3.Number of crashes

Crashes are external expressions of a bug where the program stops from running, and therefore it is easily detected. The accounting of these events is the goal of this metric. So, it estimates the number of bugs, but these will always be lower than the real number of bugs. In the best cases, this metric will provide the same result as the metric *B.1.Bugs Detected*. However, in the worst cases, this metric will not provide any result even when bugs are present. Therefore, the use of this metric seems secondary in nature to the *B.1 Bugs Detected* and thus redundant.

In Table 8 we summarize the results of the metrics *B.1.Bugs Detected, B.3.Number of crashes* and *B4.Vulnerability Speed*. These results show previously exposed drawbacks of this metric. That is, firstly, in all cases, the number of crashes is always equal (see LearnAFL and Farfuzzz with codes `tiffsplit`) or lower (`jpegtran`) than the number of found bugs (see rows 2 and 3). Secondly, there are many cases where the number of crashes is even zero for different fuzzers, but there are bugs (`jpegtran`) that have not been accounted (see rows 6 and 7). Finally, the worst case is when a fuzzer obtains more crashes than the alternative, but the number of bugs shows the opposite (see rows 9 and 10).

4.1.4. B.4.Vulnerability speed

This metric aims to estimate the speed for finding bugs computed as the ratio of the number of bugs found during a period of time.

$$V_B = \frac{Bugs}{\Delta t} \tag{1}$$

The main drawback of this metric lies in the definition of the period of time. A partial period of time can provide a too optimistic and wrong value, whereas a period of time that considers the whole testing time will always be more precise and closer to the final results. Partial periods of time can have the risk of considering as better a specific fuzzer with regard to the competitor when the final results demonstrate the opposite. On the other hand, if the period of time corresponds to the total experimentation time, the metric does not provide any additional information because it is directly proportional to metric *B.1.Bugs detected*. Under these conditions, this metric becomes redundant and does not provide any additional information. These drawbacks can occur for the same fuzzer in different repetitions and when comparing different fuzzers.

Examples of the previous situations can be observed in the experiments when plotting the evolution of the number of bugs over time in different situations. Fig. 3 shows the results of the evolution of the number of bugs in different repetitions of the same algorithm-code combination and also for different fuzzers.Fig. 3(a) as well Fig. 3(b) show examples of the risk of considering different periods of time. Fig. 3(a) shows three different periods where the same fuzzer scores better, worse, or equal, which can be used when comparing with other alternatives. This situation is even worse in Fig. 3(b) where there are many periods of time where changes the best scoring execution.

On the other hand, Fig. 3(c) and Fig. 3(d) show examples of the risk of considering wrong periods of time for two different combinations of fuzzer-program. In the first case, a repetition finds more bugs in the first moments of the run, and later, the second repetition overcomes it near the middle of the run. In the second case, this lead change takes place earlier. In both cases is a problem if this metric is used without considering the total testing time, as partial duration can lead to misinterpretations.

However, as a collateral consequence, these plots can show whether a fuzzer is systematically more effective in a short-time in contrast to other alternatives. This information can be used in those scenarios where it is necessary to maximize fuzzer performance in a shorter testing duration.

4.1.5. B.5.stability finding bugs

The stochastic nature of fuzzing causes differences between the results of each execution, which in turn shows a certain degree of uncertainty in the usage of the fuzzer. Therefore, the goal is to measure this uncertainty when different fuzzers are being compared. A fuzzer is considered more *stable* than another when the first one shows a low variability in different executions, that is, a low uncertainty. The goal of this metric is precisely to quantify the dimension of the variability in different executions and use this value to establish a relative order when comparing with other alternatives that find the same quantity of bugs.

This metric was proposed in Li et al. (2020) and in statistics is known as *Relative Standard Deviation (RDS)* or *Coefficient of variation* and is defined as:

$$c_v = \frac{\sigma}{\mu} \tag{2}$$

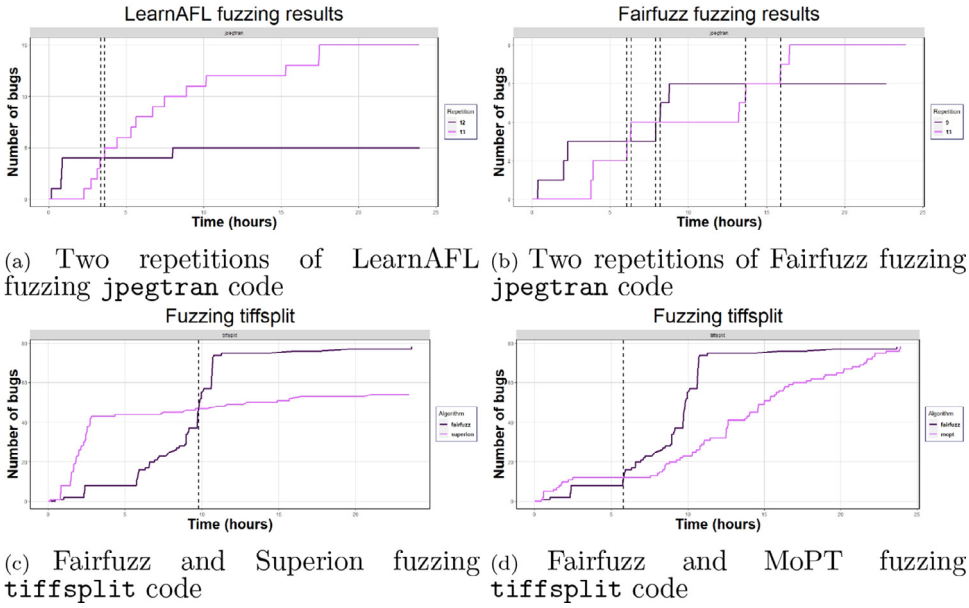
This value is a standardized measure of the dispersion of a probability distribution or, in other words, the extent of variability concerning the mean of the population.

In general, this is a good estimator of variability and could provide an insight of the fuzzer. However, this metric shows some issues:

- This metric should be computed on a ratio scale, that is, it must be defined as a zero value as a reference, which involves that values close to zero or zero excludes from the application of this metric. Furthermore, this situation can worsen because, in

Table 8
The analysis of vulnerability detection speed of LearnAFL and Fairfuzz in `tiffsplit` and `jpegtran`.

Fuzzer	Code	Repetition	Bugs Found	Number of crashes	Number ofhangs	Vulnerability detectionspeed (bugs/h)
LearnAFL	tiffsplit	9	352	352	0	**14.667**
LearnAFL	tiffsplit	14	141	139	2	**5.875**
Fairfuzz	tiffsplit	3	7	7	0	**0.333**
Fairfuzz	tiffsplit	6	149	149	0	**6.208**
LearnAFL	jpegtran	1	1	0	1	**0.041**
LearnAFL	jpegtran	14	15	0	15	**0.625**
Fairfuzz	jpegtran	3	25	0	25	**1.042**
Fairfuzz	jpegtran	6	0	0	0	**0**
LearnAFL	pdftops	0	104	24	80	**4.333**
AFLgo	pdftops	3	67	37	30	**2.792**
AFLfast	pdftops	2	27	21	6	**1.125**
AFLfast	pdftops	3	34	14	20	**1.417**



(a) Two repetitions of LearnAFL fuzzing `jpegtran` code (b) Two repetitions of Fairfuzz fuzzing `jpegtran` code

(c) Fairfuzz and Superion fuzzing `tiffsplit` code (d) Fairfuzz and MoPT fuzzing `tiffsplit` code

Fig. 3. Results of the vulnerability detection speed.

most fuzzing scenarios, it is impossible to know the number of bugs to be found beforehand. In the best case, that is, the last case, this metric is very sensitive, providing high values.

- The use of this metric lies in the number of repetitions, in statistical terms, a sufficient number of executions are required to have enough samples to compute c_v. The variability of the c_v itself considering a different number of executions is, in turn, a problem and must be considered.

The experiments show some examples of these drawbacks, in Table 9 we show an example of the effect of the proximity of the values close to zero for `sqlite3`. In this example, we also show the number of bugs and the computed value of RDS. The results show high values of RDS when the number of bugs is low but different.

On the other hand, we show in Tables 10 and 11 the effect of the number of repetitions on the computation of RDS with different combinations of fuzzers and programs. In Table 10, it is possible to observe the high variability of c_v in different executions such as in `pngtest` and Superion. In the first case, there is an increase of 2700% with the same algorithm, this may be the most representative case of the impact of considering an insufficient number of repetitions. Secondly, in Table 11 there are cases where this change can involve an incorrect relative order in terms of performance among fuzzers, such as in `pdftops` and AFLFast, AFLgo, and MoPT. In these cases, the worst initial value corresponds to

Table 9
Bugs detected and RDS values of AFLgo and MoPT fuzzing `sqlite3`.

sqlite3				
Repetitions	MoPT		AFLgo	
	Bugs	RDS	Bugs	RDS
2	0	-	3	141.42
3	0	-	3	173.21
4	0	-	3	200
5	1	223.61	3	223.61
6	1	244.95	3	244.95
7	1	264.58	3	264.57
8	1	282.84	3	282.84
9	1	300	3	300
10	1	316.23	3	316.23
11	1	331.66	3	331.66
12	1	346.41	3	346.41
13	1	360.56	3	360.56
14	1	374.17	3	374.17
15	1	387.30	3	387.30

MoPT, which ends after 15 repetitions, as the best, but this is not the case if the number of repetitions does not reach 11 repetitions.

4.2. Coverage

The metrics that belong to the coverage category are line coverage, branch coverage, and function coverage. Collectively, they

Table 10
Results of Relative Standard Deviation fuzzing `jpegtran` and `pngtest`.

jpegtran							
Repetitions	AFLFast	AFLgo	AFLplusplus	Fairfuzz	LearnAFL	MoPT	Superion
2	0.00	109.99	99.30	28.28	106.07	141.42	94.9
3	0.00	90.14	87.89	130	96.44	96.44	62.98
4	66.66	88.83	91.51	148.76	104.45	103.44	58.48
5	91.28	79.47	77.60	156.21	126.49	85.14	54.80
6	77.45	78.42	78.46	174.68	126.49	78.63	49.79
7	68.31	69.14	85.75	177.39	141.75	81.63	54.52
8	82.80	83.57	86.49	185.16	155.33	73.22	65.57
9	75.00	80.92	81.94	180.24	167.71	67.76	62.39
10	131.74	74.95	78.06	164.05	151.34	64.79	57.93
11	117.02	69.53	75.56	164.56	147.66	70.79	55.81
12	107.66	66.51	77.11	161.43	156.79	67.34	53.91
13	115.64	68.11	72.78	155.03	165.37	69.24	52.70
14	107.54	70.31	92.93	141.64	173.51	66.31	52.67
15	100.88	70.11	88.75	137.39	181.27	63.54	67.00
Pngtest							
Repetitions	AFLFast	AFLgo	AFLplusplus	Fairfuzz	LearnAFL	MoPT	Superion
2	-	141.42	-	28.28	0	-	4.56
3	-	114.56	-	130	19.39	-	71.00
4	-	141.42	-	156.44	43.22	-	55.75
5	-	154.88	-	178.34	40.47	-	56.09
6	-	173.31	-	197.59	41.26	-	77.62
7	-	149.54	-	215.00	43.20	-	77.76
8	-	163.30	-	231.05	54.52	-	91.66
9	-	175.89	-	246.02	59.32	-	100.06
10	-	179.80	-	187.83	66.28	-	99.15
11	-	177.12	-	198.80	65.91	-	102.74
12	-	181.49	-	209.16	72.59	-	108.39
13	-	190.63	-	202.27	74.90	-	113.41
14	-	177.55	-	185.30	79.57	-	120.82
15	-	178.52	-	193.31	83.49	-	124.83

Table 11
Results of Relative Standard Deviation fuzzing `pdftops`.

Pdftops							
Repetitions	AFLFast	AFLgo	AFLplusplus	Fairfuzz	LearnAFL	MoPT	Superion
2	0	41.59	31.43	141.42	66.38	141.42	414.42
3	0	149.39	26.82	162.04	52.86	152.89	124.90
4	0	119.47	34.26	110.40	45.16	105.91	97.59
5	0	139.41	80.87	87.92	99.12	86.45	119.52
6	0	155.38	74.91	106.32	115.10	102.00	123.29
7	44.10	170.95	72.93	121.52	111.35	111.73	125.57
8	74.17	184.40	72.51	134.78	102.65	124.93	107.63
9	70.36	197.32	84.47	146.71	91.47	136.71	119.31
10	72.98	208.68	94.67	151.30	98.67	137.18	106.27
11	227.33	172.56	100.91	131.88	95.96	122.17	115.76
12	190.18	149.26	108.18	118.39	89.66	110.67	105.26
13	196.51	157.46	108.18	126.42	85.91	117.12	113.24
14	182.05	163.14	107.17	118.88	83.14	114.51	120.65
15	189.98	170.69	113.82	125.82	90.19	121.42	127.60

refer to the specific part of the target that has been executed at least once during the fuzzing session. However, coverage metrics cannot be measured in all cases. It is necessary to have resources and access to specific tools that provide this data, thus in black-box approaches is not possible to measure it. Moreover, it cannot be measured in black-box environments since, to measure it, it is necessary to be able to instrument the code in order to know what is the structure of the target, that is, the total number of lines, branches, and functions (Rash, 0000).

Depending on the structure of the code, getting a high code coverage with fuzzing is a difficult task (Jun Li and Zhang, 2018). Furthermore, reaching a %100 coverage does not involve that all possible execution paths have been performed, and as a consequence, all possible errors could not have been detected (Inozemtseva and Holmes, 2014). In other words, the coverage is a necessary condition but not sufficient.

An (extreme) example of this type of situation is shown in Algorithm 1. In such an algorithm, the bug will only appear when

Algorithm 1 Example of covering code without detecting any bug.

```
1: function FLOAT DIV(float x, float y)
2:     return x/y
3: end function
```

the value of y is 0, but it will work correctly, and there will be no error with any other value. Therefore, if it first passes with any value different than y=0 through that function in the coverage analysis, it will already count as executed, but no error has been found.

Fuzzers must not only deal with previous difficulties, but with time constraints, experimental conditions represent an additional

Table 12
Line coverage results of Fairfuzz, LearnAFL and MoPT fuzzing `jpegtran`.

Fuzzer	SUT	Repetition	Line (%)	Bugs
Fairfuzz	jpegtran	1	17	2
	jpegtran	2	15.1	3
	jpegtran	3	13.2	25
	jpegtran	4	16	1
	jpegtran	5	14.9	2
	jpegtran	6	14.6	0
	jpegtran	7	14.6	2
	jpegtran	8	15.2	1
	jpegtran	9	14.2	3
	jpegtran	10	15.1	6
	jpegtran	11	15	2
	jpegtran	12	15.2	3
	jpegtran	13	14.8	4
	jpegtran	14	14.6	8
	jpegtran	15	15.8	4
LearnAFL	jpegtran	1	17.3	1
	jpegtran	2	16	4
	jpegtran	3	15.5	3
	jpegtran	4	17.5	3
	jpegtran	5	15.4	1
	jpegtran	6	15.4	2
	jpegtran	7	15.3	3
	jpegtran	8	15.4	2
	jpegtran	9	15.8	10
	jpegtran	10	15.4	1
	jpegtran	11	15.7	1
	jpegtran	12	15.4	4
	jpegtran	13	15.3	5
	jpegtran	14	15.3	15
	jpegtran	15	16.3	10
MoPT	jpegtran	1	17.1	4
	jpegtran	2	16.5	0
	jpegtran	3	15.4	3
	jpegtran	4	17.4	11
	jpegtran	5	15.2	6
	jpegtran	6	15.3	4
	jpegtran	7	15.4	2
	jpegtran	8	15.4	6
	jpegtran	9	15.4	5
	jpegtran	10	15.4	4
	jpegtran	11	15.6	1
	jpegtran	12	15.3	8
	jpegtran	13	15.3	2
	jpegtran	14	15.4	8
	jpegtran	15	16.5	5

added difficulty. This depicts a landscape where the fuzzers must show their potential, knowing a priori all previous difficulties.

4.2.1. C.1.Line coverage

This metric measures the coverage in terms of the percentage of lines that have been tested. For valid code coverage, it is necessary to exceed 70% coverage Cornett (0000).

The results of the experiments analyzing this metric have been summarized in Table 12 where the columns describe the fuzzer, repetition, percentage of line coverage, and the corresponding bugs found. The results show relevant aspects:

- The maximum coverage does not reach 30%, that is, more time would be required to reach 70% of coverage. This poor coverage is justified by the constraints regarding the experimental conditions that limit the execution time to 24 hours. In general, the fuzzers can generate millions of tests, therefore, it is not a problem generating enough tests but rather the available testing time.
- The results also show no clear relationship between the coverage and the number of bugs under the specified experimental conditions. Indeed, after computing Pearson's correlation coefficient for the three fuzzers, the results are: -0.59, -0.09, and 0.22, respectively. This consolidates that the coverage is a necessary condition but not sufficient to find bugs and especially with limited periods of testing time.
- The algorithmic complexity of a program with many branches, complex conditions, and lines can hinder the usefulness of such a metric by giving a partial view. This aspect is illustrated in Algorithm 2 where it can be seen how only measuring the line coverage does not show the complete picture of executed program logic. If the value of i is greater than or equal to 5 all the time, more than 65% line coverage is achieved, but if it is less than 5, only 40% can be explored. It is worth noting that it is on the lower line coverage section where all the decision making logic of the program is. That is, while most of the program lines are executed, most branching logic is not. However, higher line coverage numbers could lead to the belief that most program logic is executed as more lines have been executed. As illustrated with the example at hand, this is not always the case. Therefore, it is necessary to complement this metric with other specific ones to depict a clearer view of the exploration level of the code and dispel potential misunderstandings.
- When two fuzzers discover the same quantity of bugs, the coverage is useful for scoring them, considering previously exposed limitations.

Algorithm 2 Example of code where there is necessary to measure two coverage metrics.

```
1: if i ≥ 5 then
2:     i ← i − 1
3:     X ← X − 3
4:     i ← i ∗ X
5:     i ← i^2
6:     i ← i − 8
7:     i ← i/2
8:     i ← i + 2
9:     i ← i − 1
10:    X ← X/2
11:    i ← i − 1
12:    i ← i + X
13:    i ← DIV(X, i)
14:    i ← i^2 − X
15:    i ← DIV(X, i)
16:    i ← i − 1
17:    X ← DIV(i, X)
18:    X ← X/2
19:    i ← i ∗ X
20:    i ← i^2
21:    i ← DIV(X, i)
22:    i ← i − 1
23: else
24:    if i ≤ 3 then
25:        i ← i + 2
26:    end if
27:    if X ≤ 5 then
28:        X ← X + 2
29:    else
30:        if i ≥ 3 then
31:            i ← DIV(X, i)
32:        end if
33:    end if
34: end if
35: function FLOAT DIV(float x, float y)
36:    return x/y
37: end function
```

Table 13
Branch coverage results of Fairfuzz, LearnAFL and MoPT fuzzing `jpegtran`.

Fuzzer	SUT	Repetition	Branch (%)	Bugs
Fairfuzz	jpegtran	1	25.6	2
	jpegtran	2	21.3	3
	jpegtran	3	17.2	25
	jpegtran	4	23.8	1
	jpegtran	5	20.8	2
	jpegtran	6	19.8	0
	jpegtran	7	20	2
	jpegtran	8	23.3	1
	jpegtran	9	19.1	3
	jpegtran	10	21.2	6
	jpegtran	11	20.9	2
	jpegtran	12	22.7	3
	jpegtran	13	22	4
	jpegtran	14	20.1	8
	jpegtran	15	23.2	4
LearnAFL	jpegtran	1	28	1
	jpegtran	2	25.2	4
	jpegtran	3	24.2	3
	jpegtran	4	28.3	3
	jpegtran	5	23.8	1
	jpegtran	6	23.7	2
	jpegtran	7	23.6	3
	jpegtran	8	23.8	2
	jpegtran	9	24.4	10
	jpegtran	10	23.7	1
	jpegtran	11	24.2	1
	jpegtran	12	23.7	4
	jpegtran	13	23.6	5
	jpegtran	14	23.7	15
	jpegtran	15	24.5	10
MoPT	jpegtran	1	27.7	4
	jpegtran	2	26.7	0
	jpegtran	3	23.8	3
	jpegtran	4	28.6	11
	jpegtran	5	23.4	6
	jpegtran	6	23.6	4
	jpegtran	7	23.8	2
	jpegtran	8	23.7	6
	jpegtran	9	23.7	5
	jpegtran	10	23.7	4
	jpegtran	11	24.2	1
	jpegtran	12	23.6	8
	jpegtran	13	23.7	2
	jpegtran	14	23.8	8
	jpegtran	15	26.8	5

4.2.2. C.2.Branch coverage

Unlike the previous metric, this metric is aimed to estimate the coverage of the branches of a program. However, as well as the previous metric, we observe in our experimentation similar problems that have been summarized in Table 13:

- The maximum coverage does not reach 30%, it is only 23.8% being necessary to spend more time to reach 70% of coverage and like in the previous case for the same reasons.
- The results also show no clear relationship between the coverage and the number of bugs under the specified experimental conditions. Indeed, after computing Pearson's correlation coefficient for the three fuzzers, the results are: -0.57, -0.17, and 0.19, respectively consolidating that the coverage is a necessary condition but not sufficient to find bugs, especially with limited periods of testing time.
- As well as with the previous metric, the algorithmic complexity can hinder the interpretation of this metric. As it is the case with line coverage, branch coverage provides a partial view of the percentage of executed code. For instance, let us consider a program has two main branches, one with a large set of lines and the other with less lines but more sub-branches. Measuring only branch coverage may give partial results because if only the branch with many sub-branches has been tested, this coverage will be significantly higher than when the other main branch is tested. Nevertheless, a large section of the program would remain unexecuted, even if the branch coverage is high. Again, as it is the case with line coverage, this can lead to misinterpretations. Analyzing Algorithm 2, when the value of i is greater than 5, no complete branch is executed, while if it is less than 5, 75% of the branches can be executed. Nevertheless, in this second case, most of the program remains unexecuted, while the high value of the branch coverage metric might lead us to think otherwise. As in the case with line coverage, it is necessary to complement this metric with other specific ones to better represent the actual exploration level of the target code.
- When two fuzzers discover the same quantity of bugs, the branch coverage helps to score them, taking into account previously exposed limitations. This scoring must consider the experimental conditions and the same procedure to score a fuzzer based on the number of bugs detected.

4.2.3. C.3.Function coverage

This metric aims to estimate the coverage based on the number of functions that have been executed but does not provide information regarding the part of the code that has been executed within the function. Therefore, it is a weaker estimator of coverage than the previous metrics. This metric does not provide information regarding the lines of code of the function or the branches that are being executed, that is, it is a metric that has more uncertainty.

In our experimentation that has been summarized in Table 14, we observe almost similar problems. However, new additional problems emerge in this case, so we will only highlight the new problems. When two fuzzers discover the same quantity of bugs, the function coverage is not useful for scoring them. If several fuzzers execute the same percentage of functions and exactly the same functions, it is not possible to ensure whether they have executed the same sequence of instructions. So, it is not possible in the best case to score better a fuzzer concerning the rest, as this metric introduces a particular uncertainty.

Nevertheless, a high line and branch coverage will implicitly provide information regarding the function coverage, although it is necessary to consider low coverage. When the coverage increases significantly, the correlation emerges between the first two metrics and the function coverage.

Finally, the values computed by this metric implicitly include line and/or branch coverage, a function is a set of lines and branches.

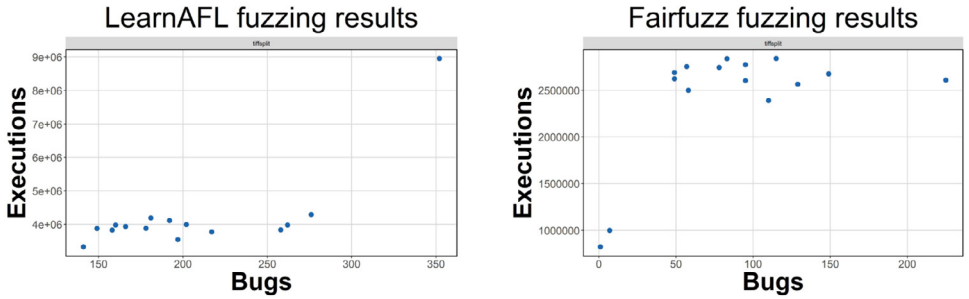
4.3. Performance

In this category, metrics that do not belong to the previous two categories are analyzed, mainly related to data patterns or test creation, successful tests, and efficiency in finding bugs concerning the number of tests, among others.

4.3.1. P.1.Number of tests.

This metric shows the number of data patterns/tests created during a specified period of time, that is, 24 hours. The experimental results show no relationship between the capability of finding bugs and the number of tests created (or data patterns). Indeed, there is no direct or indirect relationship.

This absence of relationship can be seen in Fig. 4 where we show a more revealing result with two plots where each point represents the results of a single execution for LearnAFL and Fairfuzz. The trend of the dispersion of the bugs found is linear but clearly



(a) Relation between executions and bugs of LearnAFL fuzzing `tiffsplit` during 15 repetitions

(b) Relation between executions and bugs of Fairfuzz fuzzing `tiffsplit` during 15 repetitions

Fig. 4. Relation between executions and bugs during 15 repetitions of LearnAFL and Fairfuzz fuzzing `tiffsplit` code.

Table 14
Function coverage results of Fairfuzz, LearnAFL and MoPT fuzzing `jpegtran`.

Fuzzer	SUT	Repetition	Function (%)	Bugs
Fairfuzz	jpegtran	1	23.8	2
	jpegtran	2	22.6	3
	jpegtran	3	22.3	25
	jpegtran	4	23.8	1
	jpegtran	5	22.3	2
	jpegtran	6	22.6	0
	jpegtran	7	22.3	2
	jpegtran	8	22.3	1
	jpegtran	9	22.9	3
	jpegtran	10	22.3	6
	jpegtran	11	22.8	2
	jpegtran	12	22.3	3
	jpegtran	13	22.3	4
	jpegtran	14	22.3	8
	jpegtran	15	22.8	4
LearnAFL	jpegtran	1	23.8	1
	jpegtran	2	22.6	4
	jpegtran	3	22.63	3
	jpegtran	4	22.8	3
	jpegtran	5	23.3	1
	jpegtran	6	22.6	2
	jpegtran	7	22.3	3
	jpegtran	8	22.3	2
	jpegtran	9	22.9	10
	jpegtran	10	22.3	1
	jpegtran	11	22.8	1
	jpegtran	12	22.3	4
	jpegtran	13	22.3	5
	jpegtran	14	22.3	15
	jpegtran	15	22.8	10
MoPT	jpegtran	1	23.7	4
	jpegtran	2	22.9	0
	jpegtran	3	22.3	3
	jpegtran	4	23.7	11
	jpegtran	5	22.2	6
	jpegtran	6	22.3	4
	jpegtran	7	22.3	2
	jpegtran	8	22.3	6
	jpegtran	9	22.3	5
	jpegtran	10	22.3	4
	jpegtran	11	22.3	1
	jpegtran	12	22.3	8
	jpegtran	13	22.3	2
	jpegtran	14	22.3	8
	jpegtran	15	22.9	5

independent of the number of tests. As a consequence, this metric can not be used to assess or qualify the performance of a fuzzer, but only the number of tests it is able to generate.

Table 15
Density results of LearnAFL and Fairfuzz in tiffsplit code.

Fuzzer	SUT	Repetition	Density	Density normalized
LearnAFL	tiffsplit	3	$6.73^{*}10^{-5}$	0.259
LearnAFL	tiffsplit	6	$3.84^{*}10^{-5}$	0.148
Fairfuzz	tiffsplit	5	$1.22^{*}10^{-6}$	0.005
Fairfuzz	tiffsplit	6	$5.57^{*}10^{-5}$	0.215

4.3.2. P.2.Density

It relates the number of bugs with the executions, i.e., the ratio of bugs to executions performed (see Eq. 3).

$$Density = \frac{Bugs}{Number\ of\ tests} \tag{3}$$

Although in this analysis this metric has been only used with gray-based approaches it can be used with other approaches such as with black-box based approaches.

The results of density can see in Table 15, where there is no clear relation between the number of bugs and the executions. Considering the density, the repetitions that will need fewer tests to find the bugs are going to be more effective.

In conclusion, density is an interesting metric that will provide additional information about whether the fuzzer can generate good inputs to detect bugs more efficiently, as the fewer inputs that need to be generated to detect a bug means that the fuzzer is able to generate better inputs.

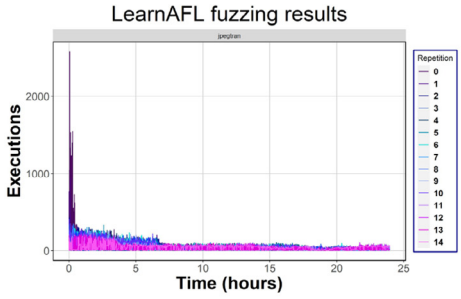
4.3.3. P.3.Execution speed (discrete)

This is the ratio between the tests that have been performed during a specific time and that time, as shown in Eq. 4.

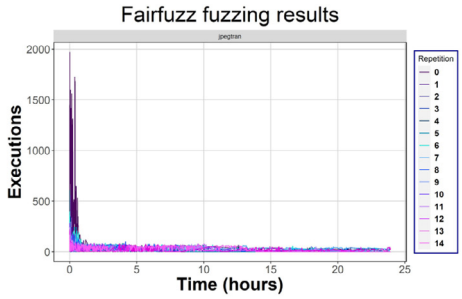
$$Execution\ speed = \frac{\Delta Executions}{\Delta t} \tag{4}$$

Fig. 5 shows the evolution of the speed during the fuzzing process. In both cases, the number of executions per second decreases considerably from the first hour onwards. This decrease is more evident when working with `jpegtran`, but with `tiffsplit` its signal is much noisier, so the results are less clear. This may be because the fuzzer generates better inputs, which need more time to execute and are not directly ejected.

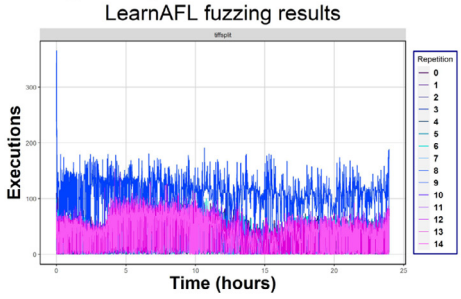
So, knowing the executions made at a specific time does not indicate how the fuzzer is working because knowing what happens at a specific instant of the process does not indicate that it will work better or worse. Moreover, if the testing time is predefined from the beginning, therefore, the execution speed gives the same information like the number of executions.



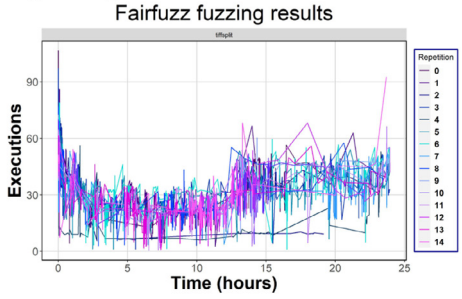
(a) Relation between executions and bugs of LearnAFL fuzzing `jpegtran` during 15 repetitions



(b) Relation between executions and bugs of Fairfuzz fuzzing `jpegtran` during 15 repetitions



(c) Relation between executions and bugs of LearnAFL fuzzing `tiffsplit` during 15 repetitions



(d) Relation between executions and bugs of Fairfuzz fuzzing `tiffsplit` during 15 repetitions

Fig. 5. Relation between executions and bugs during 15 repetitions of LearnAFL and Fairfuzz fuzzing `jpegtran` and `tiffsplit` code.

Table 16
Performance results of LearnAFL and Fairfuzz in tiffsplit code.

Fuzzer	SUT	Repetition	Time to crash	Bugs	Number of executions	Executions speed
LearnAFL	tiffsplit	2	1508	166	3,938,464	4.95
LearnAFL	tiffsplit	5	2583	158	3,828,011	52.16
LearnAFL	tiffsplit	9	1753	352	8,951,351	176.31
LearnAFL	tiffsplit	12	74	217	3,772,847	52.06
LearnAFL	tiffsplit	14	784	141	3,323,951	69.47
Fairfuzz	tiffsplit	3	2226	7	998,650	9.38
Fairfuzz	tiffsplit	5	83,271	1	821,728	34.61
Fairfuzz	tiffsplit	7	883	115	2,838,407	54.47

4.3.4. P.4.Execution speed (total)

It is the ratio between the tests that have been performed and the testing time, as shown in Eq. 5.

$$Total\ Execution\ speed = \frac{Total\ number\ of\ executions}{Testing\ time} \tag{5}$$

Table 16 shows the mean execution speed of each fuzzing repetition. However, as the testing time is predefined before starting the tests, the speed will not give additional information than *P.1*.

To sum up, this metric will not give additional information about the fuzzing efficiency, and it is also redundant to other metrics when experimental conditions are fixed between runs.

4.3.5. P.5.Time to crash

It is the time needed to detect the first crash. It is worth noting that not all bugs cause crashes, and therefore this time may be different to *B.4*. The matter about measuring bugs or crashes has already been covered previously, so we will not cover it again.

Table 16 collects the time to crash of LearnAFL and Fairfuzz with `tiffsplit`. Analyzing the table, although repetition 5 of Fairfuzz finds the least number of bugs is the one that takes the longest to cause the crash, this is because the crash occurs at the end of the execution. We conclude with the other values that there is no direct relationship between the number of detected bugs and when the first crash is detected.

4.3.6. P.6.Time to bug

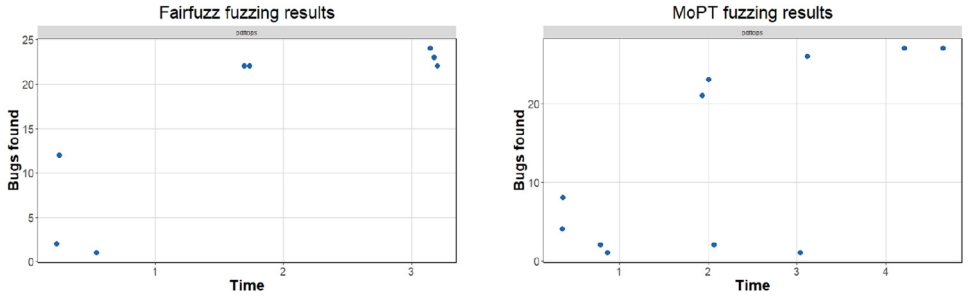
This metric represents the time needed to find the first bug without considering the type of bug.

Table 6 shows the minimum times for detecting bugs of Fairfuzz and MoPT fuzzers and the total bugs found on each repetition. It can be observed that there is no clear relationship between the instant at which the first bug is detected and the number of bugs found. In Fairfuzz (see Fig. 6a), two repetitions find their first bug in less than half an hour. However, one of them finds two bugs and the other twelve. Moreover, Fig. 6(b) shows that the repetition that detects more bugs is the one that detects the first bug latest.

In conclusion, there is not any relation between finding a bug faster and finding more.

5. Recommendations

The metrics aim to fair and objectively qualify and therefore rank the performance of any fuzzing algorithm. The experiments have shown positive and negative aspects of the metrics as well as the different aspects of the relevant experimental conditions. We



(a) Time to the first bug and the bugs find in each repetition with Fairfuzz fuzzing `pdftops` code

(b) Time to the first bug and the bugs find in each repetition with MoPT fuzzing `pdftops` code

Fig. 6. Time to the first bug and the bugs find in each repetition of Fairfuzz and MoPT fuzzing `pdftops` code.

propose a set of general recommendations based on the results to compare fuzzers. For this purpose, we have divided this set of recommendations into two groups: recommendations related to experimental conditions and recommendations associated with the selection of the best metrics that best qualify a fuzzing algorithm.

5.1. Experimental conditions

Regarding the experimental conditions, three factors impact the evaluation of fuzzers:

- Datasets: The results show the high variability of results depending on the program and the difficulty of having a reference set of programs. Therefore, we propose the following:
  - Public access. A set of public and accessible programs to be used for the reference datasets to be tested. One of the main difficulties that we have found is precisely the impossibility of accessing a broader set of programs.
  - Quantity. It is necessary to test every candidate fuzzer with enough quantity of program in order to demonstrate the potential of the algorithmic approach. The experimentation with enough quantity of programs can reveal whether a fuzzer provides good performance with specific programs or not.
  - Diversity. It is necessary to test every candidate fuzzer with programs of different nature and complexity. The nature of some programs is most susceptible to having specific types of bugs which leads to providing advantages to specific fuzzers. There is an exception to this point for cases aimed at measuring the performance of different fuzzers in a single domain (e.g., web browsers). In that case, the variety of the targets would be limited to a single domain, but targets with different complexity levels should still be considered.
- Testing time: The most important aspect of this factor is using the same period of time predefined in 24 hours as the primary recommendation for a fair comparison. The experiments show that this period of time is adequate because the main metric (the number of bugs) tends to stabilize before 24 hours. In contrast, shorter periods of time do not show this stabilization. This confirms the predilection of many authors for selecting this period of time. Nevertheless, we must consider and assume that in some situations (such as in continuous development and delivery), some other time constraints can exist for different reasons. In these cases is necessary to know the behavior of the fuzzers (some fuzzers are good at the beginning but not globally) to take advantage of this.
- Repetitions: The stochastic nature of most fuzzers leads to a high variance of results depending on the program to be fuzzed or the repetition. Thus, to make a decision based on a single repetition will lead to incorrect assessment of the fuzzer. The execution of several repetitions in order to reduce the variance is essential to determine whether a fuzzer is excelling or is something coincidental. Therefore, we recommend the several possible alternatives:
  - Statistical estimation of repetitions. This is the most strict case based on a numerical estimation of variance reduction. That is, we recommend the use of a coefficient ρ which computes the differences of the variances from cumulative repetitions in order to detect the stabilization of the variance.

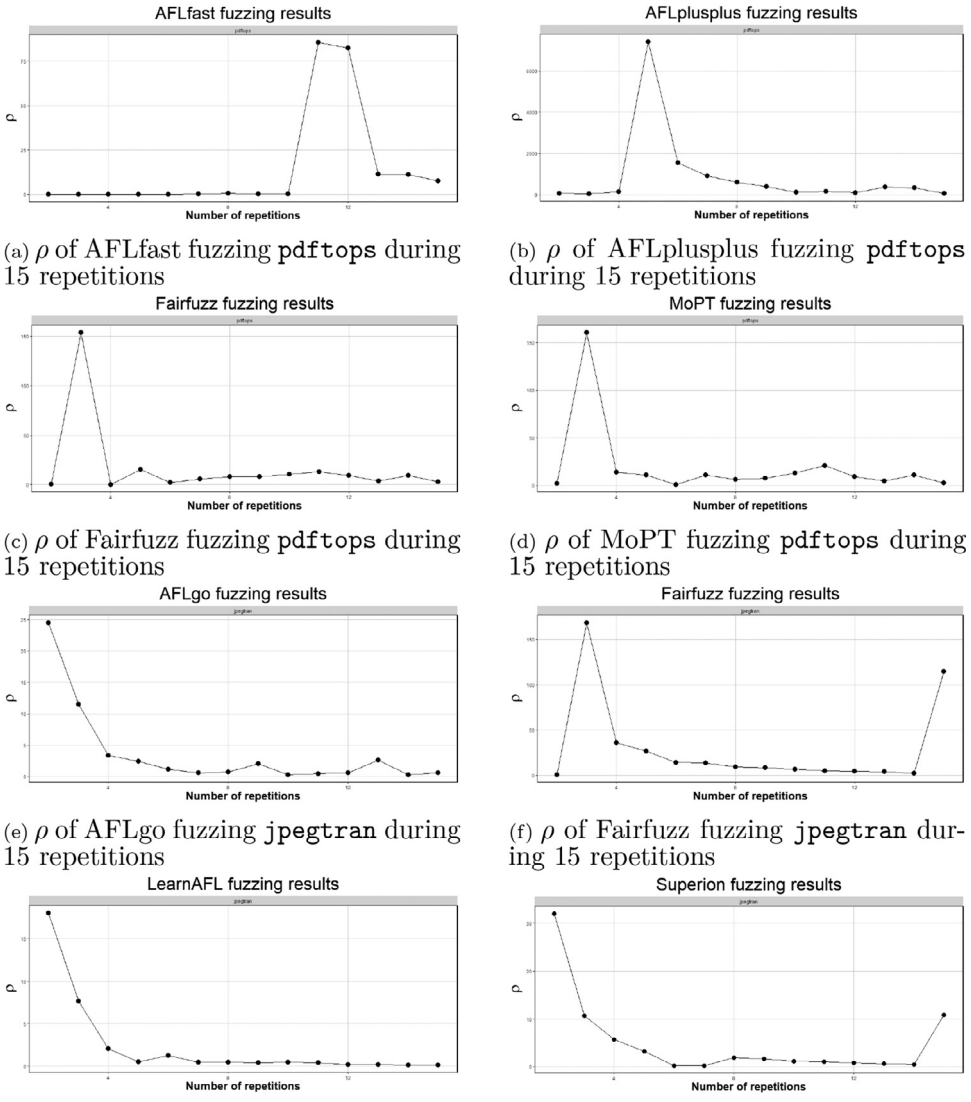
    $$\rho(r+1) = |\sigma^2(r+1) - \sigma^2(r)| \text{ with } \rho(1) = \sigma^2(1) \qquad (6)$$

    This computation shows whether the repetitions are enough or necessary to execute more repetitions providing some clues of the algorithm's performance. The use of this coefficient in our experimental setup is shown in Fig. 7 where the majority of the variance of the fuzzers for the same program tends to be stabilized before reaching 15 repetitions (for the sake of clarity, only the results of two programs are shown) when the stability is not reached means that are necessary more repetitions.
  - Fixed number of repetitions. In some cases, it is impossible to spend so much time performing repetitions, which leads to fixing a number of repetitions. There should not be less than ten repetitions, in our setup, we have selected 15 repetitions, although it will depend on the specific scenario because there can be some constraints in time or resources.
  - Improved fixed number of repetitions. In those cases where the number of repetitions can not be increased due to the costs of performing repetitions is possible to use a very known statistical technique typically used in machine learning called Bootstrapping. The basic idea is that inference about a population from sample data can be modeled by resampling the sample data. This would allow to infer the expected performance.

5.2. Fuzzing metrics

The metrics are the main factors to assess the performance of fuzzing algorithms, according to the goal of fuzzing, these must be prioritized as follows:

- Priority I. Bugs. Based on the experiments, we recommend the use of the metric *B.1 Bugs Detected* since the rest of the bug-related metrics (B.2 to B.5) fail in some way for the previously exposed reasons. However, this metric must be computed according to the new experimental conditions. This metric must



(a) ρ of AFLfast fuzzing `pdftops` during 15 repetitions

(b) ρ of AFLplusplus fuzzing `pdftops` during 15 repetitions

(c) ρ of Fairfuzz fuzzing `pdftops` during 15 repetitions

(d) ρ of MoPT fuzzing `pdftops` during 15 repetitions

(e) ρ of AFLgo fuzzing `jpegtran` during 15 repetitions

(f) ρ of Fairfuzz fuzzing `jpegtran` during 15 repetitions

(g) ρ of LearnAFL fuzzing `jpegtran` during 15 repetitions

(h) ρ of Superion fuzzing `jpegtran` during 15 repetitions

Fig. 7. Evolution of ρ during 15 repetitions.

be computed as the cumulative sum of the unwanted behaviour found through all repetitions for all predefined programs, average values shall not be used. This method of computation takes into account the stochastic nature of the fuzzers without giving any kind of advantage to fuzzers that have found many bugs in a specific repetition but few in the rest of the repetitions. This metric looks for assessing a general performance and not local or specific performance independently of the used approach which represents an advantage when comparing different methods with different approaches.

Furthermore, the values obtained by this metric can be used even to detect whether there is a predilection of a fuzzer for specific types of programs or not and demonstrate the prevalence of the fuzzer in any condition. Finally, this metric can be used in all scenarios (black-box, gray-box, and white-box).

When two or more fuzzers obtain the same values, is when is necessary to proceed with the metrics of Priority II.

- Priority II. Coverage. This category of metrics provides information regarding the performance of the exploration process carried out by the fuzzers to find bugs. Although 100% of coverage does not involve 100% of bug detection, a low capability of covering code is an indirect estimator of the capability of finding bugs. Hence, these metrics will be useful for assessing the performance of a fuzzer. The experimental results and previous simple examples show that more than one metric is necessary to quantify the coverage. The combination of the required metrics is the following:
  - C.1 Line Coverage.
  - C.2 Branch Coverage.

  As in the previous case, the computation of these metrics must be the cumulative sum of the coverage through all repetitions for all predefined programs. This can limit the effect of causal success and reflect the overall performance in terms of coverage. Finally, these metrics can be applied in white or gray scenarios, but not in black-box scenarios. Therefore, this metric will be measured as additional information when it is possible to measure it, i.e., in the black-box scenario it will not be possible to measure it.

  Additionally, the use of these two metrics is aligned with what is required to comply with the safety requirements of the IEC61508 standard, which specifies the coverage requirements for a system to be considered safe (LLC, 0000). It specifies that

in systems with a high SIL, it is necessary to achieve 100% coverage of both branches and lines.
When the fuzzers that are being compared obtain the same values, is when is necessary to proceed with the metrics of Priority III.

- Priority III. Performance: This category is aimed to reflect the efficiency of the fuzzer finding bugs, that is, the number of generated data patterns to find bugs. In this sense and based on the experimental results, the metric *P.2 Density* is the recommended metric. Density is a metric that computes the ratio of the tests performed with regard to the number of bugs independently of the used approach which facilitates the use of this metric specially with black-box approaches. A fuzzer that needs fewer runs to detect a bug will be able to detect more bugs and will overload the SUT less. As in previous cases, the computation of this metric will be carried out by summing the values of all repetitions and programs.

6. Conclusions

In this paper, we propose a set of recommendations regarding the experimental methodology and the metrics based on extensive experimentation with different algorithms, metrics, and experimental conditions. The extensive experiment has shown gaps, limitations, weaknesses, and strengths when assessing a fuzzing algorithm. Indeed, we show that from an experimental perspective, significant gaps can influence the scoring of an algorithm, but these are not unique. We have also identified problems regarding the metrics revealing that some of these can not be used to assess a fuzzing algorithm or the weaknesses of valid metrics that require a better experimental methodology. In addition, we have also identified that the quantity and diversity of target programs are also relevant when scoring different fuzzers.

The goal of these recommendations is, in essence, to cover the detected gaps, improve the experimental methodology, including target programs, as well as to select the right metrics.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Maialen Eceiza: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Funding acquisition. **Jose Luis Flores:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Mikel Iturbe:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Data Availability

The result data and the used code are available in Github.

Acknowledgments

This work has been partially funded by the IDUNN and REMEDY projects. IDUNN has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101021911.REMEDY has received funding from the Department of Economic Development, Sustainability and the Environment of the Basque Country under grant agreement KK-2021/00091. Maialen Eceiza is partially supported by the Bikaintek program (grant agreement no. 20-AF-W2-2019-00006) from the Department of Economic Development, Sustainability and the Environment of the Basque Country. Mikel Iturbe is part of the Intelligent Systems for Industrial Systems research group of Mondragon Unibertsitatea (IT1676-22), supported by the Department of Education, Universities and Research of the Basque Country.

References

Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T., 2019. Redqueen: fuzzing with input-to-state correspondence. NDDS Symp.
Index, of /gnu/binutils. https://ftp.gnu.org/gnu/binutils/.
Belle, G. V., Millard, S. P.,. STRUTS : Statistical rules of thumb. Statistics and Information.
Böhme, M., Pham, T., Nguyen, M.-D., Roychoudhury, A., 2017. Directed greybox fuzzing. ACM SIGSAC Conference 2329–2344. doi:10.1145/3133956.3134020.
Böhme, M., Pham, V.-t., Roychoudhury, A., 2017. Coverge-based greybox fuzzing as Markov chain. IEEE Trans. Software Eng. XX (X).
Cha, S., Woo, M., Brumley, D., 2015. Program-adaptive mutational fuzzing. IEEE Symposium on Security and Privacy 2015, 725–741. doi:10.1109/SP.2015.50.
Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W., 2018. A systematic review of fuzzing techniques. Computers & Security 75. doi:10.1016/j.cose.2018.02.002.
Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., 2020. Iotfuzzer : discovering memory corruptions in IoT through app-based fuzzing. Network and Distributed Systems Security (NDSS) Symposium 2018 (February 2018).
Chen, P., Chen, H., 2018. Angora: efficient fuzzing by principled search. Proceedings - IEEE Symposium on Security and Privacy 2018-May, 711–725. doi:10.1109/SP.2018.00046.
Chen, P., Liu, J., Chen, H., 2019. Matryoshka: fuzzing deeply nested branches. CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.
Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Su, Z., Jiao, X., 2019. EnFuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium.
Chipounov, V., Kuznetsov, V., Candea, G., 2011. S2E: A platform for in-vivo multi-path analysis of software systems. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 265–278. doi:10.1145/1950365.1950396.
Cho, Mingi (Yonsei University); Kim, Seoyoung (Yonsei University); Kwon, T. Y. U., 2019. Intriguer : Field-Level Constraint Solving for Hybrid Fuzzing, 515–530.
Cornett, S.,. Minimum acceptable code coveragehttps://www.bullseye.com/minimum.html.
Davidson, D., Moench, B., Jha, S., Ristenpart, T., 2013. FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution. Proceedings of the 22nd USENIX Security Symposium 463–478.
Fell, J., 2017. A review of fuzzing tools and methods. PenTest Magazine.
Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M., 2020. AFL++: Combining incremental steps of fuzzing research. WOOT 2020 - 14th USENIX Workshop on Offensive Technologies, co-located with USENIX Security 2020.
Ganesh, V., Leek, T., Rinard, M., 2009. Taint-based directed whitebox fuzzing. Proceedings - International Conference on Software Engineering 474–484. doi:10.1109/ICSE.2009.5070546.
Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. ACM SIGPLAN Notices 43 (6), 206–215. doi:10.1145/1379022.1375607.
Godefroid, P., Levin, M.Y., Mol, 2008. Automated whitebox fuzz testing. Network and Distributed System Security Symposium (July) 1. doi:10.1007/978-3-642-02652-2_1.
Godefroid, P., Levin, M. Y., Molnar, D., 2012. SAGE: Whitebox Fuzzing for Security Testing.
Godefroid, P., Peleg, H., Singh, R., 2017. Learn & fuzz: Machine learning for input fuzzing, 50–59, doi:10.1109/ASE.2017.8115618.
Grieco, G., Ceresa, M., Mista, A., Buiras, P., 2017. QuickFuzz testing for fun and profit. Journal of Systems and Software 134, 340–354. doi:10.1016/j.jss.2017.09.018.
Hazimeh, A., Herrera, A., Payer, M.,. Home magma. https://hexhive.epfl.ch/magma/.
Hazimeh, A., Herrera, A., Payer, M., 2020. Magma: A Ground-Truth Fuzzing Benchmark850868,2009.01120, arXiv:2009.01120.
Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. Proceedings - International Conference on Software Engineering (1) 435–445. doi:10.1145/2568225.2568271.
Istvan, H., Slowinska, A., Neugschwandtner, M., Bos, H., 2018. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. Journal of Applied Research in Intellectual Disabilities 31 (5), 820–832. doi:10.1111/jar.12436.
Jitsunari, Y., Arahori, Y., 2019. Coverage-guided learning-assisted grammar-based fuzzing, 275–280, doi:10.1109/ICSTW.2019.00065.
Jun Li, B.Z., Zhang, C., 2018. Fuzzing: a survey. Cybersecurity doi:10.1186/s42400-018-0002-y.
Jung, J., Hu, H., Solodukhin, D., Pagan, D., Lee, K. H., Kim, T., 2019. Fuzzification: Anti-fuzzing techniques.
Klees, G., Ruef, A., Hicks, M., 2018. Evaluating fuzz testing. CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.

Index, of /sources/win32. http://xmlsoft.org/sources/win32/.
libjpeg-turbo, - browse /1.5.2 at SourceForge.net. https://sourceforge.net/projects/libjpeg-turbo/files/1.5.2/.
LIBPNG, : PNG reference library. https://sourceforge.net/projects/libpng/files/libpng12/older-releases/1.2.45/.
LibTIFF, -4.0.9. https://www.linuxfromscratch.org/blfs/view/8.3/general/libtiff.html.
Deepfuzzer, 2021. Deepfuzzer: accelerated deep greybox fuzzing. IEEE Trans Dependable Secure Comput 18, 2675–2688. doi:10.1109/TDSC.2019.2961339.
Lemieux, C., Sen, K., 2017. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage1709.07101, doi:10.1145/3238147.3238176.
Li, D., Chen, H., 2019. FastSyzkaller: improving fuzz efficiency for linux kernel fuzzing. J. Phys. Conf. Ser. 1176 (2). doi:10.1088/1742-6596/1176/2/022013.
Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering Part F1301, 627–637. doi:10.1145/3106237.3106295.
Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., Lu, K., Wang, T., 2020. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers.
Li, Y., Ji, S., Lv, C., Chen, Y., Chen, J., Gu, Q., Wu, C., 2019. V-fuzz: Vulnerability-oriented evolutionary fuzzing
Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J., 2018. Fuzzing: state of the art. IEEE Trans. Reliab. PP, 1–20. doi:10.1109/TR.2018.2834476.
Liang, J., Jiang, Y., Chen, Y., Wang, M., Zhou, C., Sun, J., 2018. PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode. ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering 809–814. doi:10.1145/3236024.3275525.
Lin, Y.D., Liao, F.Z., Huang, S.K., Lai, Y.C., 2016. Browser fuzzing by scheduled mutation and generation of document object models. Proceedings - International Carnahan Conference on Security Technology 2015-Janua, 1–6. doi:10.1109/CCST.2015.7389677.
e. c. L.L.C,. IEC 61508 compliant module testing: Part 2. https://www.exida.com/blog/IEC-61508-Compliant-Module-Testing-Part-II.
Lopez, P.,. kermitt2/xpdf-4.00. Original-date: 2018-02-22T19:05:35Z https://github.com/kermitt2/xpdf-4.00.
Luo, Z., Zuo, F., Shen, Y., Jiao, X., Chang, W., Jiang, Y., 2020. ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation. Technical Report.
Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R., 2019. MOPT: Optimized mutation scheduling for fuzzers. Proceedings of the 28th USENIX Security Symposium 1949–1966.
MP3gain, - browse /mp3gain/1.5.2 at SourceForge.net. https://sourceforge.net/projects/mp3gain/files/mp3gain/1.5.2/.
McNally, R., Yiu, K., Grove, D., Gerhardy, D., 2012. Fuzzing: the state of the art. Defence Science and Technology Organisation Edinburgh (Australia) 43.
Metzman, J., Arya, A., Szekeres, L., 2020. FuzzBench: Fuzzer Benchmarking as a Service.
Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of UNIX utilities. Commun ACM 33 (12), 32–44. doi:10.1145/96267.96279.
Molnar, D., Li, X.C., Wagner, D., 2009. Dynamic test generation to find integer bugs in x86 binary linux programs. Proceedings of the 18th conference on USENIX security symposium 1–24. http://dl.acm.org/citation.cfm?id=1855768.1855773.
Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D., 2018. What you corrupt is not what you crash: challenges in fuzzing embedded devices. Network and Distributed Systems Security (NDSS) Symposium 2018 doi:10.14722/ndss.2018.23176.
Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A., 2017. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. doi:10.1145/3167132.3167289.
PCR, E2. https://www.pcre.org/.
Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-Fuzz: fuzzing by program transformation. IEEE Symposium on Security and Privacy (SP) 697–710. doi:10.1109/SP.2018.00056.
Pham, V.-t., Böhme, M., 2016. Model-Based whitebox fuzzing for program binaries. ASE 2016: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering 552–562.
Rash, M.,. afl-cov - AFL fuzzing code coveragehttps://github.com/mrash/afl-cov.
Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. VUzzer: Application-aware evolutionary fuzzing. NDDS Symp (March) doi:10.14722/ndss.2017.23404.
Rebert, A., Avgerinos, T., Cha, S.K., Foote, J., Warren, D., Grieco, G., Brumley, D., 2003. Optimizing seed selection for fuzzing. USENIX Association, p. 256.
SQLite, source 3.30.1. https://www.npackd.org/p/sqlite-source/3.30.1.
Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T., 2017. kafl: Hardware-assisted feedback fuzzing for os kernels.
Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: augmenting fuzzing through selective symbolic execution. NDSS 16 (2016), 1–16.
Tcpdump-libpcap, public repository. https://www.tcpdump.org/old_releases.html.
Takanen, A., Demott, J.D., Miller, C., 2008. Fuzzing for software security testing and quality assurance. Artech House.
IFuzzer, 2016. : An evolutionary interpreter fuzzer using genetic programming. Springer Verlag, pp. 581–601. doi:10.1007/978-3-319-45744-4_29.
Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire : data-Driven seed generation for fuzzing. IEEE Symposium on Security and Privacy.
Wang, J., Chen, B., Wei, L., Liu, Y., 2019. Superion: grammar-aware greybox fuzzing. Proceedings - International Conference on Software Engineering 2019-May, 724–735. doi:10.1109/ICSE.2019.00081.
Wang, T., Wei, T., Gu, G., Zou, W., 2010. TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. Proceedings - IEEE Symposium on Security and Privacy 497–512. doi:10.1109/SP.2010.37.
Wang, Y., Cartmell, M., 1997. KLEE: An overtaking model and the determination of safe passing sight distance for cars and trucks, 209–224.
Woo, M., Cha, S.K., Gottlieb, S., Brumley, D., 2013. Scheduling black-box mutational fuzzing. Proceedings of the ACM Conference on Computer and Communications Security 511–522. doi:10.1145/2508859.2516736.
Xie, X., See, S., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis 146–157. doi:10.1145/3293882.3330579.
Yan, G., Lu, J., 2017. ExploitMeter : Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability. 10.1109/PAC.2017.10
You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B., 2019. SLF: fuzzing without valid seed inputs. Proceedings - International Conference on Software Engineering 2019-May, 712–723. doi:10.1109/ICSE.2019.00080.
Yue, T., Tang, Y., Yu, B., Wang, P., Wang, E., 2019. LearnAFL: greybox fuzzing with knowledge enhancement. IEEE Access 7, 117029–117043. doi:10.1109/ACCESS.2019.2936235.
Zhao, J., Wen, Y., Zhao, G., 2011. H-fuzzing: A new heuristic method for fuzzing data generation. 32–43,10.1007/978-3-642-24403-2_3

Maialen Eceiza received the degree of electronic and automatic industrial engineering in 2017 and the master of embedded systems in 2019 from the University of Basque Country. She holds a Ph.D. from Mondragon Unibertsitatea in 2022, where she worked on fuzzing for embedded systems. She is currently working as researcher at Ikerlan Technology Research Center in the Cybersecurity in Embedded System team.

Jose Luis Flores is a researcher at Ikerlan Technology Research Center and he is currently part of the Cybersecurity in Embedded System team. He holds a M.Sc. in Robotics and Advanced Control from the University of the Basque Country. His main interest is related to Artificial Intelligence and Cybersecurity. As such, the main lines he works on in each organization are Embedded System security at Ikerlan, Machine Learning and Optimization at the university.

Mikel Iturbe is a lecturer and researcher at Mondragon Unibertsitatea and he is currently part of the Data Analysis and Cybersecurity research group. He holds a Ph.D. from Mondragon Unibertsitatea, where he worked on data-driven intrusion detection in industrial networks and a MSc in ICT Security from the Open University of Catalonia. His main research interest is related to cybersecurity, primarily in the industrial sector. As such, the main lines he works on are Industrial Control System security, Embedded Security and Software Security. He also works in exploring novel data-driven applications for cybersecurity.