ARMy Fuzzing - DRAFT

Andrew York
Clemson University

Kathryn Smith
Clemson University

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

Due to the success of fuzzing as a method of identifying bugs in software, many fuzzers have been developed. Unfortunately, selecting a fuzzer is a difficult task. The following challenges prevent researchers from quickly selecting a fuzzer using benchmarking: no fuzzer consistently outperforms the others, the efficacy of a fuzzer is inconsistent throughout its execution, and fuzzing results are not reproducible.

In response to these problems, Autofz was developed. Autofz leverages multiple fuzzers by dynamically allocating resources to fuzzers using runtime data. This methodology empowers Autofz to eliminate the fuzzer selection problem and outperform individual fuzzers.

In spite of these successes, Autofz is not perfect. While there is no consensus on the best way to assess fuzzers, Autofz oversimplifies this process. Autofz relies on a singular metric, bitmap coverage, to rank its fuzzers and allocate resources. Autofz may perform even better if it employs additional metrics when assigning resources to its fuzzers.

Moreover, Autofz prides itself on streamlining the fuzzer selection process for the novel user, but it was designed for and tested on an AMD Ryzen 9 3900 with 24 cores and 32 GB memory. This equipment is not representative of the standard user, and as a result, Autofz needs to be adapted for use in additional environments. [2]

In this effort, we seek to improve Autofz and answer the following research questions:

- **RQ1.** Can we successfully reproduce a working tool chain from Fu et al. [2] on portable ARM64 computing devices?
- **RQ2.** Can we reproduce the results (e.g. number of defects found) of fuzzing at least two targets from Fu et al. [2] on our ARM64-based tool chain?
- **RQ3.** Autofz uses a single metric (i.e. AFL bitmap coverage) for allocation of resources to fuzzers. Can we identify at least one additional, alternate metric that will improve this allocation?

2 Related Work

2.1 Fuzzing

In 1990, Miller et al. proposed fuzz testing as a tool to identify bugs in Unix utilities and other applications. Their fuzzer generated, deployed and monitored the execution of random strings on a target program. The application could succeed, crash, or hang. The application was successful when program terminated normally, and the input did not identify a bug. Alternatively, if the input caused the program to crash or hang, a defect was discovered. Using this simple methodology, Miller et al. were able to uncover bugs within Unix utilities that had not been previously identified using formal testing procedures. [4]

Since Miller et al. introduced fuzzing as an effective means of identifying bugs, over thirty more sophisticated fuzzers have been proposed. Most fuzzers conform to the basic methodology introduced by Miller et al; they generate test cases and deploy them on a target while monitoring its state.

Despite their similarities, the test case generation phase varies between fuzzers. Fuzzers can be divided as generation-based or mutation-based fuzzers. Generation-based fuzzers generate inputs that conform with grammars while mutation-based fuzzers modify existing inputs.

Fuzzers can also be divided into blackbox, whitebox, and greybox. Blackbox fuzzers only monitor the state of the target application before and after the execution of an input while whitebox fuzzers document the status of the target throughout execution, and greybox fuzzers monitor specific execution attributes. [**?**]

// add more detail about how seeds are generated

2.2 Fuzzing Assessment Methods

Despite the abundance of fuzzers and their differences in implementation and performance, a consensus has not emerged on how to compare them.

2.2.1 Bitmap Coverage

Bitmap coverage is a one means of capturing a fuzzer's performance; it quantifies the number of branches explored by the fuzzer. To calculate bitmap coverage, a large bitmap is generated. Each bit represents some combination of the target's entrance and exit points. When an input is injected, the bit that represents the input's entrance and exit locations is incremented. A non zero bit indicates that that branch has been explored. After execution, the number of non-zero bits represents the bitmap coverage achieved by the fuzzer. [**?**]

2.2.2 Additional Metrics

Ecezia et al. asserted that the methodology for testing new and evaluating existing fuzzers needs to be standardized. In their proposed protocol, they recommended using three categories of metrics. They are bugs, coverage, and performance.

- **Bugs** Ecezia et al. recommends using the total number of bugs discovered as the best means of assessing the general performance of a fuzzer. They argue that since number of bugs discovered encompasses all other bug metrics, it is the superior bug metrics. While total number of bugs identified may encompass unique bugs discovered, unique bugs discovered is likely a better means of measuring a fuzzers general performance. Total number of bugs discovered includes duplicate bugs. This may result in a fuzzer that discovers the few bugs many times appearing to perform better than another fuzzer that discovers more unique bugs once. Moreover, once a bug is known, it does not need to be discovered again.

- **Coverage** Coverage metrics communicate the performance of a fuzzer's exploratory processes. While Ecezia et al. do not explicitly identify bitmap coverage in their proposed method, they recommend a combination of line and branch coverage for evaluating the coverage achieved by a fuzzer. Despite this recommendation, they maintain that complete line or branch coverage does not guarantee complete bug discovery. Instead, they report that if the fuzzer does not provide the target bug triggering input, an error may remain undiscovered even after the buggy line of or path through code is fuzzed.

- **Performance** Performance metrics reflect the efficiency of the fuzzer. Ecezia et al. reports that density is the best means of measuring a fuzzer's efficiency. Density communicates the ratio of number of tests completed to bugs discovered. A denser fuzzer will need few runs to identify a bug, and consequently, identify more bugs faster than a fuzzer with a lower density.

2.2.3 Experimental Conditions

In addition to establishing the evaluation metrics, Ecezia et al. established criteria for fuzzer test conditions. They asserted that in order to compare fuzzers, the fuzzers must be executed against the same applications, at least 15 times, and for the same period of time. These conditions are aimed at eliminating inconsistencies associated with evaluating fuzzers.

For instance, fuzzers can only be compared if they are run on the same target application because the performance of a fuzzer is directly linked to the system under test. When a more buggy program is fuzzed, it will yield more bugs than a well-designed system regardless of the fuzzer.

Moreover, when evaluating a fuzzer, it must be executed at least 15 times because the random input generated by a fuzzers will vary between runs.

The final condition that must be constant for each test is execution time; a fuzzer executed for 10 hours cannot be compared to a fuzzer that has been running for 24 hours; the fuzzer that ran for 24 hours will likely discover more bugs than the fuzzer that has run for 10 hours. The 24 hour fuzzer is not necessarily a better fuzzer, but it had more opportunities to discover unwanted behavior. [1]

2.3 Autofz

While Ecezia et al. looked to simplify the difficult task of identifying the best fuzzer by establishing evaluation criteria, Fu et al. aimed to eliminate the task. They proposed a new fuzzer that would make the task of evaluating and selecting an individual fuzzer obsolete. Autofz is a dynamic fuzzer that dynamically deploys a set of fuzzers. Autofz accomplishes this by dividing its workload into two phases; they are a preparation and a focus phase.

During the preparation phase, Autofz captures the runtime trends of multiple fuzzers on the target application. First, Autofz deploys the fuzzers with the same seeds. Each fuzzers execute for the an equal portion of the preparation phase. Next, Autofz measures bitmap coverage achieved by each individual fuzzers. This process is repeated until the end of the preparation phase is reached or a strong trend emerges. A strong trend occurs when the difference in bitmap coverage of the best and worst performing fuzzer exceeds a predetermined threshold.

Based on the data collected in the preparation phase, Autofz deploys the set of fuzzers with the potential to maximize the focus phase's performance. More specifically, Autofz allocates resources to the fuzzers proportionally to their performance during the preparation phase. For example, the fuzzers that demonstrated the greatest bitmap coverage during the preparation phase have the most resources allocated to them during the focus phase, and the fuzzers that preformed poorly will have less available. This may result in one fuzzer receiving all available resources or no resources. After assigning resources to the individual fuzzers, the focus phase is executed. In the focus phase, each fuzzer is executed with its respective resources.

During the focus phase, Autofz achieved greater bitmap coverage than individual fuzzers that executed under the same conditions. [2]

3 Methodology

In order to confirm the findings presented in Fu et.al. [2], we deployed our own instance of their autofz environment. We replicated their testing environment by deploying their docker image on our own hardware. Moreover, we tested our recreation of Fu et al.'s autofz against the targets: exiv2, mp3gain, mojs, and tcpdump. Our initial findings validate the results presented in Fu et al.; autofz is consistently a top performing against the tested targets.

We have also been able to produce a working tool chain on ARM64, with some significant variations from Fu et al. [2]. Our ARM64 environment implements 7 fuzzing algorithms in Ubuntu 22.04 LTS, running in a virtual machine with the UTM app on host laptops running macOS 14.3.1 on Apple Silicon (ARM64) architecture **(RQ1)**. Key differences are described in Table 1.

	Original [2]	ARM64	Reason
Ubuntu Version	16.04	22.04	1
aflforkserver.so	x86_64	aarch64	2
AFL (original)	Yes	Yes	
AFLFast	Yes	Yes	
Angora	Yes	No	4
Fairfuzz	Yes	Yes	
LAF-Intel	Yes	Yes	
LearnAFL	Yes	No	3
LibFuzzer	Yes	Yes	
QSYM	Yes	No	4
Radamsa	Yes	Yes	
RedQueen	Yes	Yes	

Table 1: Key differences between Original AMD64 and ARM64 Environments

There are a number of reasons for the differences between Fu et al., and our ARM64 environment. They correspond to the reason numbers in Table 1 and are:

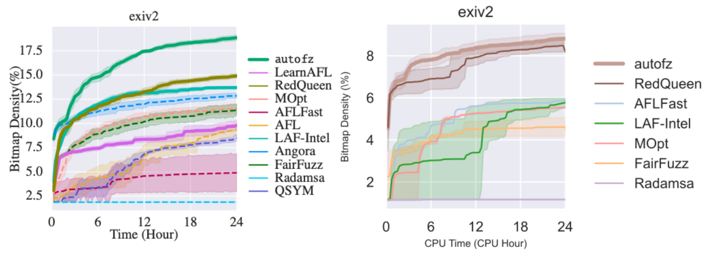

Figure 1: A comparison of bitmap density covered in the original [2] and our coverage during initial fuzzing of exiv2

1. Ubuntu 16.04 LTS does not support the ARM64 architecture (also called aarch64).
2. Aflforkserver.so comes from the quickcov package, developed to support the Cupid research project [3]. We had to recompile it for aarch64.
3. Learn AFL does not properly compile instrumented binaries for aarch64.
4. Angora and QSYM are written such that they are dependent upon the x86_64 ISA. They do not support other architectures.

To compare the modified autofz with the proposed autofz, we tested it against the targets: exiv2, nm, mojs, and tcpdump. Initial test indicate that the ARM64 compatible autofz performs similarly to the unmodified autofz. Figure 1 displays bitmap density covered of the individual algorithms in our ARM64 implementation, as compared to a similar plot from Fu et. al [2].

Bitmap density coverage of our initial fuzzing campaign against tcpdump is plotted in figure **??**, shown as a comparison with results from Fu et al. [2].

In addition to producing an ARM64 compatible autofz, we identified a potential improvement for autofz. Existing versions of autofz use bitmap coverage to rank fuzzers during the preparation, but bitmap coverage may not be the best metric for identifying the most effective fuzzers for a target. While bitmap coverage indicates the portion of a target that was fuzzed, it does not guarantee bug discovery. To resolve these problems, we plan to modify autofz, so fuzzers that discover more bugs during the preparation phase are rewarded with more resources during the focus phase because discovering more bugs across a smaller area of code decreases the attack surface by more entry points than discovering fewer bugs over a greater area of the code.

References

[1] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Improving fuzzing assessment methods through the analysis

of metrics and experimental conditions. *Computers & Security*, 124:102946, 2023.

[2] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. autofz: Automated fuzzer composition at runtime. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1901–1918, Anaheim, CA, August 2023. USENIX Association.

[3] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference*, pages 360–372, 2020.

[4] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.