# "ARMy Fuzzing:" Metrics for comparably Efficient Fuzzing on Commodity, Portable [ARM64] Devices

Andrew York
*Clemson University*

Kathryn Smith
*Clemson University*

## 1 Problem

Software is constantly evolving and growing in complexity. As a result, new methodologies develop to identify vulnerabilities in these programs. Fuzzing is one technique that has emerged. Fuzzing identifies unwanted behavior by generating and deploying random inputs on a target program. [1]

Due to the success of this technique, 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 the fuzzers using runtime data. Autofz eliminates the fuzzer selection problem and consistently outperforms individual fuzzers.

In spite of its successes, Autofz is not perfect. While there is no consensus on the best way to assess fuzzers, Autofz oversimplifies this process. Autofz relies on one metric, path coverage, to rank its fuzzers and allocate resources. Autofz could be improved by incorporating additional metrics during its analysis.

Moreover, Autofz prides itself on streamlining the fuzzer selection process for the novel user, but it was designed and tested on Ubuntu 20.04 equipped with AMD Ryzen 9 3900 having 24 cores and 32 GB memory. This equipment is not representative of the standard user, and as a result, Autofz needs to be rebuilt and tested in additional environments. [2]

## 2 Related Work

### 2.1 Fuzzing

In 1990, Miller et al. proposed fuzz testing as a tool to test Unix utilities and other applications. Their fuzzer generated random number and character strings. Next, they executed a target program with the random input. Finally, the fuzzer would record how the tested application responded to the random input. The application could succeed, crash, or hang; succeed meant that the program executed normally, a crash indicated that the program terminated abnormally while a hang implied that the application entered an infinite loop. Furthermore, a crash or hang indicated unwanted behavior within the target application. Using this simple methodology, Miller et al. discovered a wealth of bugs within Unix utilities that had not been identified using formal testing procedures. [4]

### 2.2 Fuzzing Assessment Methods

Since Miller et al. introduced fuzzing as an inexpensive means of identifying bugs and increasing overall system reliability, many researchers have developed more complex fuzzers. Despite this abundance of fuzzers, a consensus has not emerged on how to compare them. Moreover, testing conditions are not consistent for new fuzzers. Ecezia et al. aimed to resolve this issue by proposing a universal method for testing new fuzzers and evaluating existing fuzzers. They presented evaluation metrics that could be divided into three categories: bug detection, coverage, and performance. Metrics defined in the bug detention group quantified any undesirable behavior identified by the fuzzer while metrics categorized under coverage measured the percentage of the code that had been executed by the fuzzer. Metrics in the performance category are measurements not directly related to bug detection or performance; they include data such as number of runs executed during a set time frame and time needed to identify the first bug.

After establishing the metrics that need to be collected during testing, Ecezia et al. set the criteria for fuzzer test conditions. They asserted that in order to compare multiple fuzzers, the fuzzers must be tested against the same target 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 applica-

tion because the results of fuzzers are 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, in order to evaluate 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 run 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 collaborative fuzzer that dynamically deploys a set of fuzzers. Autofz accomplishes this by dividing its workload into 2 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. These fuzzers are allowed to execute for the same period of time. Next, Autofz measures the number of unique paths explored by the individual fuzzers. This process is repeated until the phase times out or a strong trend emerges. A strong trend occurs when the difference between the code 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 fuzzers with the potential to maximize the focus phase's performance. Specifically, Autofz allocates resources to the fuzzers proportionally to their performance during the preparation phase. For example, the fuzzers that demonstrated the greatest code coverage during the preparation phase have the most resources allocated to them during the focus phase. Moreover, one fuzzer can be allocated 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. Autofz also outperformed collaborative fuzzers that equally allocated resources to individual fuzzers. [2]

## 3 Remaining Plan, Timeline

In this effort, we seek to 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?

Our plan for the remaining time in this term is as follows:

1. In **March**, we will implement at least one metric proposed in February and compare its effectiveness against the existing version of Autofz (i.e. AFL bitmap).
2. In **April**, we will compile our results and document our findings.

## 4 Progress to Date

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, nm, mojs, and tcpdump. Our initial findings validate the results presented in Fu et al.; autofz is consistently a top preforming fuzzers 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.

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:

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. Original AFL does not compile instrumented binaries for aarch64. Instead, we implement AFL++.
4. Angora and QSYM are written such that they are dependent upon the x86_64 ISA. They do not support other architectures.

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

Table 1: Key differences between Original AMD64 and ARM64 Environments

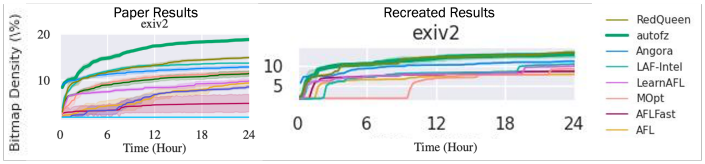


Figure 1: A comparison of bitmap density covered in the original [2] and our coverage during initial fuzzing of exiv2

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 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 2, 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.

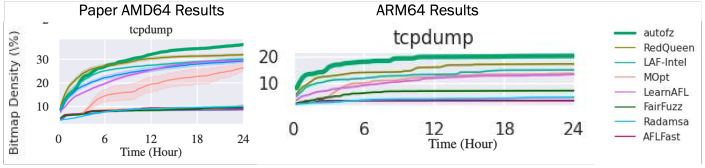


Figure 2: A comparison of bitmap density covered in the original [2] and our coverage during initial fuzzing of tcpdump

## 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.