“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.

Due to the success of this technique, many fuzzers have been developed. The effectiveness of individual fuzzers vary greatly between target programs. As a result, productive fuzzing relies on selecting the best fuzzer for the target. Unfortunately, one fuzzer does not consistently outperform the others, so selecting a fuzzer is a difficult task.

In the past, researchers solved this problem through offline analysis and benchmarking. Despite their efforts, the following challenges prevented researchers from consistently selecting the best fuzzer: no fuzzer outperforms the others, efficacy of each fuzzer is inconsistent throughout execution, equal resource allocation is inefficient, and fuzzing results are not reproducible. Autofz aims to solve these problems by leveraging multiple fuzzers and dynamically allocating resources to the fuzzers using runtime data.

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 for its analysis. Potential metrics for evaluation include the number of bugs discovered and the number of run cycles executed during the preprocessing phase.

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. [3]

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 el 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 el 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 application 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 el 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. Moveover, 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 Plan, Timeline

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] 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.