Towards Effective Performance Fuzzing

Yiqun Chen, Matthew Bradbury and Neeraj Suri
School of Computing and Communications, Lancaster University, UK
{y.chen101, m.s.bradbury, neeraj.suri}@lancaster.ac.uk

***Abstract*—Fuzzing is an automated testing technique that utilizes injection of random inputs in a target program to help uncover vulnerabilities. Performance fuzzing extends the classic fuzzing approach and generates inputs that trigger poor performance. During our evaluation of performance fuzzing tools, we have identified certain conventionally used assumptions that do not always hold true. Our research (re)evaluates PERFFUZZ [1] in order to identify the limitations of current techniques, and guide the direction of future work for improvements to performance fuzzing. Our experimental results highlight two specific limitations. Firstly, we identify the assumption that the length of execution paths correlate to program performance is not always the case, and thus cannot reflect the quality of test cases generated by performance fuzzing. Secondly, the default testing parameters by the fuzzing process (timeouts and size limits) overly confine the input search space. Based on these observations, we suggest further investigation on performance fuzzing guidance, as well as controlled fuzzing and testing parameters.**

***Index Terms*—performance fuzzing, input selection, metrics**

I. INTRODUCTION

Typically, performance measures a program's processing capability. Performance issues obviously waste computational resources, however, they can also lead to safety, liveness and security violations. An attacker could provide inputs which cause Denial-of-Service (DoS) attacks against running programs by exploiting the worst algorithmic cases [2]. For example, a hash table is usually implemented with amortized constant complexity [3]. If the hash algorithm produces many hash collisions, then the hash table's performance degrades to performing a linear search for the requested element.

There exist many performance diagnostic tools [4], which provide runtime information such as function call graphs annotated with performance overheads plus hardware and software events. However, these approaches are typically unable to identify causalities of performance issues without suitable test cases. Therefore, performance fuzzing techniques [1], [5] aim to generate test cases that trigger worst algorithmic cases.

1) The Basic Fuzzing Process: A typical fuzzing process searches for vulnerabilities by providing randomly generated inputs to a target software under test (SUT). On execution, the fuzzer records all inputs that crash or hang, and selects interesting inputs. The process of input selection is known as *guidance*, which is typically performed based on an input's code coverage. Finally, the selected inputs will be randomly *mutated* to yield *input mutants*, the fuzzing process is repeated with these mutants as the inputs for subsequent iterations.

2) Performance Fuzzing: In performance fuzzing, the process is guided by *performance values* and *program components*. Typically, the program components can be visualized as the edges of a control flow graph (CFG) [6], and the number of traversals on each edge represent the performance values [1], [5]. The *path length* is the number of hits on CFG edges[1]. Small inputs that maximize the performance impacts are favoured (as is the case with traditional fuzzing), because such test cases with small inputs are not expected to yield bad performance. PERFFUZZ [1] extends `AFL` by adding path length guidance in addition to code coverage guidance. SLOWFUZZ [5] guides the performance fuzzing solely by path length. Both works focus on the path length yielded by an input, but neither measured pertinent performance metrics, i.e., the execution time.

3) Observation 1: A longer path length means the SUT executes more steps which implies a longer execution time. However, inputs with the same path length will have different execution times when the performance overhead of CFG nodes differ. Therefore, the efficacy of a performance fuzzer should be measured by the performance overhead of its test cases.

4) Observation 2: Default performance fuzzing parameters limit the search space in order to constrain the time spent evaluating inputs. However, a short timeout prevents the fuzzer from searching for potentially interesting test cases that cause slow execution. In recent works, timeout inputs are ignored when evaluating input quality [1], [5]. Similarly, the default limit on the input size (1 MiB by `AFL` and PERFFUZZ) also restricts the search space. As we will demonstrate, such limits could restrict the search space for interesting test cases with larger input sizes that lead to larger performance overheads.

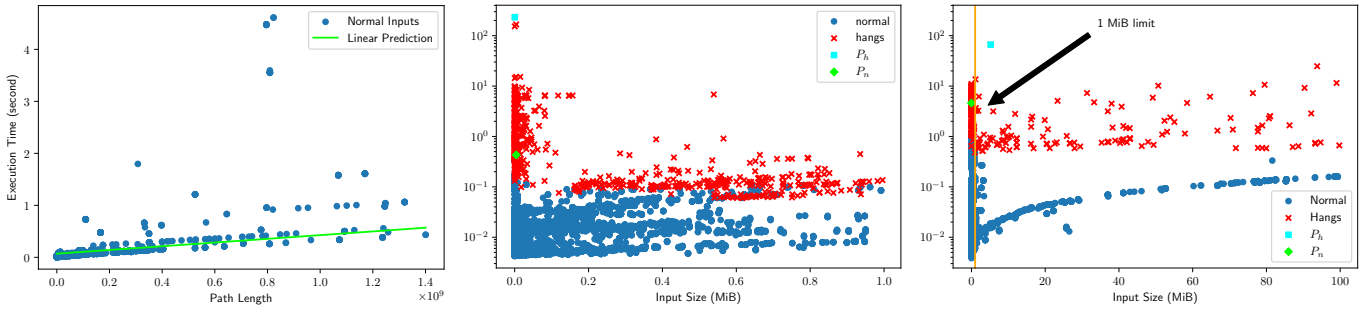
Our research investigates two performance fuzzing problems using `AFL` based on PERFFUZZ [1] as the research target.

1) Are path lengths a good indicator for performance?
2) Do default parameters overly limit the input search space?

II. EXPERIMENTAL SETUP AND INITIAL ANALYSIS

The SUT is `libjpeg`, which is a widely used library and was also used by previous evaluations of PERFFUZZ [1]. The experiments are carried out on a separate virtual machine with 8 Intel® Xeon® Gold 6248R CPUs (3.00 GHz) with 8 GiB memory, running Ubuntu Linux 20.04.3 LTS with the Linux kernel 5.4.0. We spawn 8 fuzzer instances running in parallel, and measure the execution time in seconds of generated inputs with `time`. Each measurement of the execution time is repeated 100 times to mitigate the impacts of external noises. We carried out fuzzing with 2 setups and ran experiments on generated test cases with a time budget of 18 hours. $\mathbb{S}1$ is the setup

[1]In PERFFUZZ, the path length is the number of hits on CFG nodes. This is equivalent to using CFG edges as the performance values will be accumulated.



(a) Performance and path length (b) Performance and input size (S1) (c) Performance and input size (S2)

with default fuzzing parameters (1 MiB and 1 s) and S2 is with increased file size limits of 100 MiB and the timeout of 10 s. `AFL` selects timeout dynamically and the timeout parameter is the upper bound of the actual timeout.

1) Problem 1: Fig. 1a shows the correlation between execution time and path length. Each blue dot is the median execution time, and we focus on normal inputs for simplicity. The green line is the linear regression, with a Pearson's correlation coefficient of 0.64 and mean squared error of 0.025. The execution time is generally correlated with the path length, but not all path lengths result in a similar execution time. For example, the input with the highest execution time (4.61 s) has a path length of 0.8×10^9, while the input with the longest path length (1.4×10^9) has an execution time of 0.43 s. Therefore, the quality of generated test cases should be evaluated using execution time, as some inputs with a long path are not necessarily interesting for performance diagnostics. Further research should explore more relevant performance fuzzing guidance, e.g., memory allocation [1].

2) Problem 2: In our experiments, S1 has generated 17 912 normal inputs and 671 hanging inputs, while S2 produced 13 119 normal inputs and 336 hanging inputs. The difference in the number of generated inputs is expected as the increased input file size and timeout value slow the fuzzing process. Figs. 1b and 1c show the relationship between input size and the execution time, where blue dots are normal test cases, red crosses hang, and the orange vertical line in Fig. 1c is the default 1 MiB input size limit. Two interesting inputs are highlighted for each setup, which are the slowest normal test case (denoted P_n) and the slowest hanging test case (denoted P_h). The default fuzzing parameters (S1) limit the capability of performance fuzzing to detect interesting test cases, because the slowest normal test case finishes in 0.43 s. S2 can result in test cases which are slower and have a smaller input sizes. For example, the slowest S2 test case runs for 4.61 s, with one tenth the input size of the slowest S1 test case (436 B for P_n in S2 and 4496 B for P_n in S1). When analysing performance, hanging cases are interesting as they yield longer execution time than normal test cases. For example, S1 has a timeout input with an execution time of 233.14 s and a small input size (304 B), while other inputs larger than 1 MiB can be processed within a second. Here, the input likely triggered slow code in `libjpeg` which has the potential to be optimized. However, hanging inputs are not fully explored by fuzzers and whether a fuzzing process can produce interesting inputs which hang is not deterministic. There exists a trade-off between a larger search space and the cost to explore it. Larger input sizes and timeouts allows for additional interesting parameters to be found, but this incurs a time and computational cost to explore.

3) Open Issues: We have observed that testing parameters have large impacts on performance fuzzing, which both overly constrains the search space and insufficiently limits it. Secondly, though the path length is partially correlated to execution time, we find that it cannot replace performance metrics in terms of selecting interesting inputs. Further research on performance fuzzing should: 1) use performance metrics (e.g., execution time or throughput) to select inputs for performance diagnostics, 2) identify suitable parameters for optimal input search space of a SUT, and 3) explore further fuzzing guidance besides path length, i.e., memory allocations and number of instructions [1].

Acknowledgment

This work was supported, in part, by EC H2020 CONCORDIA GA No. 830927 and by the UKRI Trustworthy Autonomous Systems Node in Security [EPSRC grant EP/V026763/1].

References

[1] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *Proc. of the 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2018, pp. 254–265. DOI: 10.1145/3213846.3213874.

[2] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *12th USENIX Security Symposium*, USENIX Association, 2003.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang, *XRay: A Function Call Tracing System*, Online: https://research.google/pubs/pub45287, 2016.

[5] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *Proc. of the 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2155–2168. DOI: 10.1145/3133956.3134073.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006, ISBN: 0321486811.