Program Feature-based Fuzzing Benchmarking

Miao Miao
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA
Email: mmiao@utdallas.edu

***Abstract*—Fuzzing is a powerful software testing technique renowned for its effectiveness in identifying software vulnerabilities. Traditional fuzzing evaluations typically focus on overall fuzzer performance across a set of target programs, yet few benchmarks consider how fine-grained program features influence fuzzing effectiveness. To bridge this gap, we introduce a novel benchmark designed to generate programs with configurable, fine-grained program features to enhance fuzzing evaluations. We reviewed 25 recent grey-box fuzzing studies, extracting 7 program features related to control-flow and data-flow that can impact fuzzer performance. Using these features, we generated a benchmark consisting of 153 programs controlled by 10 fine-grained configurable parameters. We evaluated 11 popular fuzzers using this benchmark. The results indicate that fuzzer performance varies significantly based on the program features and their strengths, highlighting the importance of incorporating program characteristics into fuzzing evaluations.**

***Index Terms*—fuzzing benchmarking, program features.**

I. Research Problem and Motivation

The evaluation of fuzz testing is usually conducted on a set of target programs, focusing on the overall performance (e.g., bug finding capability and code coverage) after fuzzers run a preset period of time. We observe that such evaluations often reveal that different fuzzers tend to favor specific programs. For instance, fuzzers' performance often varies across different target programs in the evaluations that use FuzzBench [1]. One of the reasons for such variation lies in the design of any given fuzzer. For example, EcoFuzz's advantage in reducing energy wastage and maximizing path coverage [2] may become more pronounced as program complexity increases. However, current evaluations do not consider program features or analyze performance deviations in relation to those features. Therefore, the research community has yet to establish a link between fuzzing performance and program features; without this link, it remains unknown if the hypotheses and claims made in these fuzzers hold, making it hard to assess and further improve them. To close this gap, we propose to develop a feature-based fuzzing benchmark that systematically controls the syntactic structure of programs. Our approach provides fine-grained, configurable parameters to construct benchmark programs that represent specific features of control-flow and data-flow complexity, offering deeper insights into fuzzing performance in terms of different program features.

II. Background and Related Work

There are several important fuzzing benchmarks and are widely used in fuzzing evaluation. FuzzBench [3] provides an infrastructure to evaluate fuzzers in terms of code coverage and vulnerability exposure. Magma [4] provides a benchmark of 138 ported bugs in 9 open source programs along with the lightweight oracle (ground truth) that reports the bug when triggered. There exist other benchmarks focus on vulnerabilities in the programs (e.g., FixReverter [5], LAVA-M [6], and CGC [7]). These benchmarks focus on bugs or vulnerabilities in the programs but do not identify program properties that can influence the performance of the fuzzer. UNIFUZZ [8] proposes a collection of pragmatic performance metrics to evaluate fuzzers from six complementary perspectives. GreenBench [9] focuses on energy consumption of fuzzing evaluations. Although these approaches bring new dimensions to understand fuzzer performance, they still overlook the influence of specific characteristics in target programs that can impact fuzzer effectiveness. Wolff et al. [10] and Zhu et al. [11] evaluated fuzzers based on program properties. However, the program properties they proposed do not focus on the systematic generation of programs using configurable program features.

III. Approach

A. Feature Extraction

We reviewed 25 grey-box fuzzing papers that are published within last three years, as well as the most cited fuzzers from earlier years, and summarized the common hypotheses or claims of improvements on fuzzing performance. Note that It is not our goal to cover all published fuzzing papers. Instead, we reviewed these popular fuzzing papers as a representative set to extract important program features to construct the benchmark. In total, we extracted 7 program features from two aspects: *control-flow complexity* and *data-flow complexity*.

1) Control-Flow Complexity (15 papers): Defined by four program features: *number of conditional branches*, *execution probability of conditional branches*, *loops and recursions* and *loops and recursions with data constraints*.

2) Data-Flow Complexity (10 papers): Defined by three program features: *magic bytes*, *checksum tests*, and *nested magic bytes and checksum tests*.

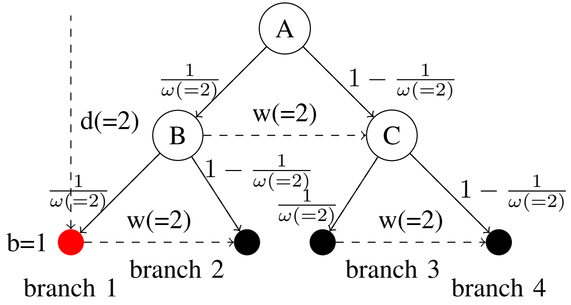
B. Benchmark Generation

We generate synthetic programs emphasizing specific features with varying levels of strength to assess fuzzer performance. We adjust control- and data-flow complexity by stacking template blocks, and use fine-grained configurable

```

void COMP_W2_D2_ω2_B1(unsigned hash)
{if (hash < 2) {  // A (nesting n=1)
     if (hash < 1) {  // B (nesting n=2)
         PRINTF("This is branch 1\n");
         ... // Inserting a bug here
     } else {
         PRINTF("This is branch 2\n");
     }
 } else {
     if (hash < 3) {  // C (nesting n=2)
         PRINTF("This is branch 3\n");
     } else {
         PRINTF("This is branch 4\n");
     ... // some code
}
```

(a) Program template



(b) Control-flow graph

Fig. 1: Illustrative example of control-flow complexity parameters. `Width`, `Depth`, `Weight`, and `BBranch` are denoted as w, d, ω, and b respectively.

parameters to control each feature's strength. We crafted 10 configurable parameters and generated a total of 153 programs, targeting 7 distinct program features.

1) Control-Flow Complexity.: We define six parameters to manipulate the control-flow complexity of the programs. `Width`, `Depth`, `Weight`, `BBranch`, are used to control the number of conditional branches and the execution probability of the buggy branch (as illustrated in Figure 1). `Iteration` and `Has_Data_Constraint`, are used to control the generation of programs with bugs reside in a deep loop or recursive call.

2) Data-Flow Complexity: We use four parameters to define the data-flow complexity of the programs: `Start`, `Length`, `Depth`, and `Count`, which defines the starting index of the magic bytes, the number of involved magic characters, the nesting level of conditions, and the number of checksum tests, respectively.

IV. Results and Contributions

We evaluate the performance of 11 fuzzers on our feature-based benchmark suite. We report the *completion rate*, which calculates the successfully completed programs within the timeout to show how effectively each fuzzer supports specific feature parameters. We also calculate the *Spearman's rank correlation coefficient* of each feature parameter and the fuzzing runtime to analyze the impact of the strength of each parameter on the performance of different fuzzers. Table I shows the partial results of *control-flow complexity* features parameters.

TABLE I: **COMD**, **COMW**, and **COMWE** stand for Depth, Width, and Weight of control-flow complexity. Spearman correlation (**corr**) and completion rate (**comp**). RedQ stands for RedQueen, Mem-S and -H stand for two variants (Stack and Heap) of Memlock, and Tort-B and -L stand for two variants (Basic Block and Loop) of TortoiseFuzz, Hongg stands for Honggfuzz. Statically significant correlations are denoted with an asterisk (*). Weak correlations (between -0.3 and 0.3) with a 100% completion rate are highlighted in bold, and a hyphen (-) indicates unavailable correlations due to insufficient data.

Fuzzer	COMD		COMW		COMWE	
	corr	comp	corr	comp	corr	comp
EcoFuzz	**0.287***	1.00	**-0.024**	1.00	**-0.237***	1.00
MOpt	0.662*	1.00	**0.106**	1.00	**-0.192***	1.00
AFLFast	0.517*	1.00	**0.010**	1.00	-0.307*	1.00
Fairfuzz	0.513*	1.00	**0.141***	1.00	-0.364*	1.00
RedQ	0.878*	1.00	-	0.06	-0.452*	1.00
Laf-intel	0.853*	0.75	0.333*	0.38	**-0.291***	1.00
Mem-S	0.534*	1.00	**0.154***	1.00	**-0.228***	1.00
Mem-H	0.500*	1.00	**0.177***	1.00	**0.061**	1.00
Tort-B	0.839*	1.00	0.253*	0.94	-0.474*	1.00
Tort-L	0.735*	0.88	0.044	0.50	-0.410*	1.00
AFL	0.640*	1.00	**0.255***	1.00	-0.315*	1.00
AFL++	0.894*	1.00	0.872*	1.00	-0.507*	1.00
Hongg	0.366*	1.00	0.013	0.94	**-0.093**	1.00

We made several observations on different features. For example, `Depth` of control-flow complexity has a stronger impact on the fuzzing performance than `Width` of control-flow complexity. AFL++ is most sensitive to the increase of control-flow complexity, while EcoFuzz is the least sensitive. *Our findings indicate that fuzzer performance varies significantly based on program features and their strength. Establishing a link between fuzzing performance and program features can help developers assess and improve these fuzzers for better effectiveness.* Moving forward, we plan to perform static analysis to extract additional program features from real-world programs, and expand our benchmark to include features representing a broader range of real-world scenarios.

Overall, we made the following contributions in this work:

- A literature review of 25 recent grey-box fuzzing papers to extract 7 fine-grained program features from their claimed improvements.
- A feature-based fuzzing benchmark with 153 programs systematically generated using 10 configurable parameters for the extracted program features.
- Evaluate 11 popular fuzzers on our feature-based benchmark to understand fuzzer behaviors and the impact of each program parameter on their performance.

V. Acknowledgment

This work was partly supported by NSF grants CCF-2008905 and CCF-2047682.

REFERENCES

[1] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.

[2] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[3] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[4] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[5] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3699–3715. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[6] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[7] D. CGC, "Darpa cyber grand challenge (cgc)," https://github.com/CyberGrandChallenge/, 2018.

[8] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "{UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2777–2794.

[9] J. Ounjai, V. Wüstholz, and M. Christakis, "Green fuzzer benchmarking," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1396–1406.

[10] D. Wolff, M. Böhme, and A. Roychoudhury, "Explainable fuzzer evaluation," *arXiv preprint arXiv:2212.09519*, 2022.

[11] X. Zhu, X. Feng, T. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 658–663.