BandFuzz: A Practical Framework for Collaborative Fuzzing with Reinforcement Learning

Wenxuan Shi
wenxuan.shi@northwestern.edu
Northwestern University
Evanston, Illinois, USA

Hongwei Li
li4909@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Jiahao Yu
jiahao.yu@northwestern.edu
Northwestern University
Evanston, Illinois, USA

Wenbo Guo
henrygwb@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Xinyu Xing
xinyu.xing@northwestern.edu
Northwestern University
Evanston, Illinois, USA

ABSTRACT

In recent years, the technique of collaborative fuzzing has gained prominence as an efficient method for identifying software vulnerabilities. This paper introduces BandFuzz, a distinctive collaborative fuzzing framework designed to intelligently coordinate the use of multiple fuzzers. Unlike previous tools, our approach employs reinforcement learning to enhance both the efficiency and effectiveness of fuzz testing.

ACM Reference Format:
Wenxuan Shi, Hongwei Li, Jiahao Yu, Wenbo Guo, and Xinyu Xing. 2024. BandFuzz: A Practical Framework for Collaborative Fuzzing with Reinforcement Learning. In *2024 ACM/IEEE International Workshop on Search-Based and Fuzz Testing (SBFT '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3643659.3648563

1 INTRODUCTION

In recent years, collaborative fuzzing has emerged as a powerful and effective approach to uncovering software vulnerabilities. This collaborative approach leverages the strengths of multiple fuzzers to enhance the overall efficacy of the fuzzing process. Notable examples [1–4] of this approach include projects such as autofz [3], which incorporates the collective efforts of 11 different fuzzers, and Pastis [1], a combination of three fuzzers, showcased in the SBFT23 competition. These initiatives have demonstrated the remarkable potential of collaborative fuzzing to unearth vulnerabilities in various software applications.

The principle of collaborative fuzzing is simple: by combining different fuzzing techniques, we can create a more robust and comprehensive fuzzer. Each fuzzer contributes its unique strengths, such as different mutation operations, or seed selection strategies. The synergy achieved by amalgamating these distinct approaches significantly increases the likelihood of discovering previously elusive software vulnerabilities, even transforming seemingly inefficient mutation strategies into highly effective ones when the normal mutation search space becomes exhausted. Consequently, the scheduling of different fuzzing strategies becomes the pivotal question.

In this paper, we introduce BandFuzz, a novel collaborative fuzzing framework designed to intelligently orchestrate the use of multiple fuzzers. Unlike previous tools, our method is powered by reinforcement learning. In BandFuzz, our reinforcement learning algorithm dynamically allocates fuzzing resources to the most effective fuzzer for a given target program. Through the algorithm, BandFuzz could adapt and adjust its fuzzing strategy in response to changes in the environment. This allows BandFuzz to optimize performance in real-time instead of relying on predetermined strategies based on offline evaluations. To the best of our knowledge, this is the first work that leverages a reinforcement learning algorithm to optimize fuzzing resource allocation for better performance.

2 METHODS

Fuzzing is an iterative process of discovering program states. Various fuzzers employ different strategies to enhance their performance throughout the entire process, but this performance can vary greatly between iterations. Due to limited resources in the competition (only one CPU core and a 23-hour time frame), it becomes crucial to make informed decisions about which fuzzer to use at each iteration. To tackle this challenge, BandFuzz incorporates three key components aimed at optimizing fuzzing efficiency.

The first component is the **Fuzzer Performance Monitor**. It acts as a real-time assessment tool, constantly evaluating the performance of different fuzzers. It does this by analyzing metrics like code coverage and execution time for each fuzzer. This monitoring mechanism offers valuable insights into how each fuzzer performs in the current environment.

The second component is a **Reinforcement Learning Algorithm**. This algorithm uses performance information collected by the Fuzzer Performance Monitor to predict which fuzzer will likely produce the best results in the next iteration. By analyzing past performance data, the algorithm aims to make smart decisions about selecting fuzzers and improve the fuzzing process over time.

The third component is the **Fuzzing Task Manager**. It allocates time and computing resources to selected fuzzers based on forecasts generated by the reinforcement learning algorithm. Additionally, it possesses the capability to continuously monitor the health of fuzzers, promptly identifying crashes or infinite loops, and taking

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SBFT '24, April 14, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0562-5/24/04
https://doi.org/10.1145/3643659.3648563

Table 1: Mutation score by different fuzzers for various targets

Benchmark	bandfuzz (%)	fox (%)	libafl (%)	libfuzzer (%)	mystique (%)	pastis (%)	tunefuzz (%)
freetype2_ftfuzzer	100.00	87.00	91.00	75.00	87.00	90.00	87.00
jsoncpp_jsoncpp_fuzzer	100.00	57.00	55.00	59.00	57.00	60.00	58.00
lcms_cms_transform_fuzzer	100.00	89.00	91.00	89.00	92.00	91.00	85.00
libpcap_fuzz_both	86.00	100.00	74.00	67.00	83.00	76.00	86.00
libxml2_xml	100.00	95.00	85.00	84.00	86.00	86.00	91.00
re2_fuzzer	100.00	55.00	55.00	55.00	55.00	55.00	90.00
stb_stbi_read_fuzzer	97.00	94.00	95.00	89.00	97.00	100.00	100.00
zlib_zlib_uncompress_fuzzer	100.00	98.00	90.00	98.00	90.00	99.00	92.00

corrective actions such as rebooting to maintain their functionality. Ultimately, this leads to better program state discovery within resource constraints.

In summary, our approach BandFuzz combines real-time performance monitoring, reinforcement learning-based forecasting, and resource allocation to optimize fuzzing iterations in situations with limited resources. This framework is designed to enhance the efficiency and effectiveness of the fuzzing process, ultimately leading to improved program state discovery outcomes.

3 IMPLEMENTATION

BandFuzz is comprised of 3.2K lines of Go code and includes various code patches for fuzzer integration. To implement the Fuzzer Performance Monitor, we also utilize a customized runtime shared object along with LLVM coverage sanitizer. Additionally, our framework is implemented as a Docker instance, enabling its inclusion in benchmarks and competitions.

4 EVALUATION

Based on the final evaluation results of SBFT 2024, BandFuzz achieved first place in mutation analysis benchmarks and also demonstrated excellent performance in coverage benchmarks.

Figure 1 shows that BandFuzz achieved the highest number of mutant kills in total.

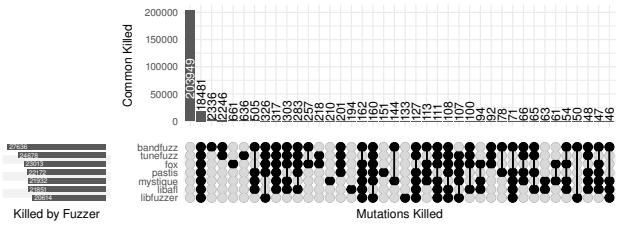

Figure 1: Upset figure of killed mutants for different fuzzers

As depicted in Figure 2, BandFuzz successfully covers and eliminates a significant number of mutants across various types. This demonstrates the robustness of BandFuzz in bug hunting.

Table 1 presents the effectiveness of different fuzzers in eliminating mutants during mutation-based testing for various benchmarks. The values in the table show the mutation score of each fuzzer. The

Figure 2: Percentage of mutants covered or killed

highest value in each row is considered 100%, and other percentages are relative to this maximum. BandFuzz ranked 1st on 6 of 8 benchmarks and achieved the highest average score.

REFERENCES

[1] 2023. PASTIS: Collaborative Fuzzing Framework. Quarkslab.
[2] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proc. of USENIX Security*.
[3] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. 2023. autofz: Automated Fuzzer Composition at Runtime. In *Proc. of USENIX Security*.
[4] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Proc. of ACSAC*.