Poster: Dynamic Multi-Fuzzer Scheduling via Reinforcement Learning

Naomasa Matsubayashi, Yuichi Sugiyama
Ricerca Security, Inc.
Email: {naomasam, yuichis}@ricsec.co.jp

***Abstract*—In this paper, we focus on macro-level optimization that dynamically switches among multiple fuzzers and propose an automated switching approach using reinforcement learning (RL). By controlling fuzzers with different exploration characteristics on the same CPU core, a RL agent sequentially decides which fuzzer to use, using the number of newly discovered seeds as a reward. We applied our approach to real-world programs and confirmed that exploration efficiency improved when training and evaluation were performed on the same target program. Furthermore, we observed partial transferability in a specific case when applying the trained agent to other programs.**

I. Introduction

Fuzzing is widely used as a software defect detection technique, leveraging automated input generation and runtime feedback to efficiently discover previously unknown vulnerabilities. Previous studies have primarily focused on *micro-level optimization* within individual fuzzers, such as dynamically optimizing seed selection and mutation strategies [1]. Meanwhile, *macro-level optimization*, in which multiple fuzzers are used either concurrently or sequentially, has also recently attracted attention [2], [3], further expanding the exploration space by sharing seeds generated by each fuzzer.

However, many existing methods simply run fuzzers in parallel in a fixed manner, offering insufficient mechanisms for flexibly switching among fuzzers according to the dynamic execution status or internal structure of the program under test (PUT). Therefore, in this study, we propose an approach that uses reinforcement learning (RL) to *dynamically optimize* switching among fuzzers. Specifically, a RL agent, with the number of newly saved seeds as its reward, determines which fuzzer to execute at each point in time.

II. Background and Motivation

Among various software defect detection methods, fuzzing stands out for its ability to efficiently uncover unknown vulnerabilities by iteratively generating and mutating inputs based on runtime feedback [1]. A single fuzzer comprises multiple components (e.g., seed selection and mutation), each with a range of parameters. Much research has focused on dynamically optimizing these parameters to improve exploration efficiency [4]–[6]. We refer to this type of optimization within a single fuzzer as *micro-level optimization*.

Meanwhile, *macro-level optimization*, which harnesses multiple fuzzers either in parallel or in stages to explore diverse execution paths, has garnered attention [2], [3], [7]. For example, EnFuzz [2] runs heterogeneous fuzzers in parallel and shares seeds among them, thereby discovering paths and bugs that a single fuzzer might miss. However, many existing methods only execute fuzzers in a *fixed* parallel manner, providing insufficient flexibility to reallocate resources as time progresses. In reality, the optimal fuzzer can shift depending on both the characteristics of the PUT and the current stage of exploration—some fuzzers excel in initial exploration, while others are specialized for particular input formats.

Recent work such as autofz [7] dynamically switches fuzzers at runtime by monitoring progress, indicating that adaptive resource allocation tailored to the exploration process can yield further efficiency gains. Building upon these insights, this study proposes a macro-level optimization employing RL for *dynamic fuzzer switching*, providing more flexible and effective resource allocation than existing approaches.

III. Prototype Implementation

We implement an adaptive scheduler that dynamically switches between AFL++ (mutation-oriented) and Angora (constraint-guided), leveraging their complementary exploration strengths. A Python script orchestrates these fuzzers, leveraging the asynchronous advantage actor-critic (A3C) algorithm from the Ray RL library to guide switching decisions. Specifically, the agent learns to choose between two actions —running AFL++ for 10 iterations or running Angora for 10 iterations—based on observed execution metrics.

We provide the agent with simple statistics such as the number of times each fuzzer has been selected and the total number of discovered seeds, which constitute its state. The reward is defined as the number of newly saved seeds in each iteration. By monitoring the number of newly discovered seeds at each iteration, the A3C agent learns which fuzzer is more effective at different stages of execution, ultimately enabling it to automatically derive an optimal fuzzer-switching policy.

IV. Evaluation

In this section, we address two research questions (RQ) to evaluate the effectiveness and generalization capability of our macro-level optimization approach based on fuzzer switching:

RQ1: Does performance improve when the model is trained and evaluated on the PUT?

RQ2: Can a model trained on one PUT also yield performance improvements on a different PUT?

We compared our RL-based fuzzer switching method with a baseline that randomly switches between the two fuzzers. In both cases, we measured the progression of coverage over time. For the RL-based method, the policy was learned offline

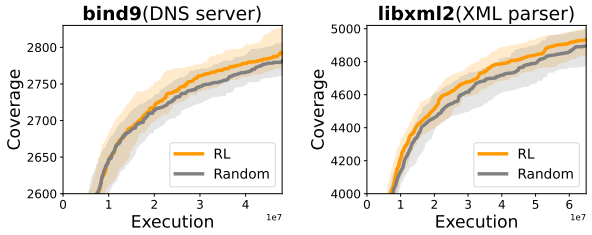


Fig. 1: Coverage over time when trained and evaluated on the same PUT (average of 80 runs).

for 10,000 training cycles on each target PUT, following an offline RL (also known as batch RL) approach. This training phase utilized the A3C algorithm and the reward mechanism detailed in Section III. During the subsequent evaluation runs, this fixed pre-learned policy was applied consistently to make scheduling decisions. Our evaluation experiments were conducted on several PUTs selected from FuzzBench [8].

A. RQ1: Same-PUT Performance

To assess the effectiveness of RL in the fuzzer-selection task, we examined whether performance improves when the model is evaluated on the same PUT used for training. We first trained and evaluated on `bind9` (DNS server parsing network protocols). We then performed a similar experiment training and evaluating on `libxml2` (XML parser library).

Figure 1 shows the results for both experiments. For `bind9`, our approach shows almost no performance difference from random switching in the early stages, but achieves higher final coverage. For `libxml2`, it exhibits a faster initial coverage growth and similarly obtains a higher final coverage. These findings indicate that macro-level optimization through RL is effective at least for the same PUT trained on.

B. RQ2: Cross-PUT Generalization

Next, we evaluated the generalization capability by testing whether the model trained on `bind9` improved performance when applied to different PUTs. We applied the `bind9`-trained model to `libxml2`, as well as to `harfbuzz` (text shaping engine handling font files), `sqlite3` (database engine processing SQL), and `libpcap` (packet capture library).

Figure 2 shows the results. The `bind9`-trained model only improved performance significantly on `libpcap`, showing little difference for `libxml2`, `sqlite3`, or `harfbuzz`. This limited transfer might stem specifically from shared network packet processing logic between `bind9` and `libpcap`. More generally, effective generalization seems to require specific similarities in PUTs' input structure or execution behavior.

V. Conclusion and Future Direction

In this paper, we proposed a prototype system that uses RL to achieve macro-level optimization by switching among multiple fuzzers, and evaluated it on AFL++ and Angora. Experimental results showed that when training and evaluation were conducted on the same PUT, our approach achieved higher coverage than random switching. However, applying the

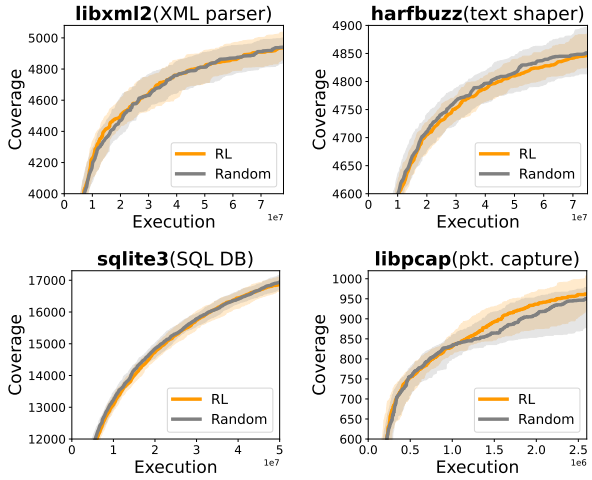


Fig. 2: Coverage over time when applying the model trained on `bind9` to different PUTs.

trained model to a different PUT often yielded minimal performance improvements. This indicates that a PUT's specific characteristics, particularly those related to its input structure and execution behavior, significantly impact the effectiveness and transferability of learned fuzzer-switching policies.

In future work, we plan to explore methods that learn more general fuzzer-switching policies by considering PUTs' unique structural and behavioral properties. Understanding these properties via static [9] and potentially dynamic analysis could reveal transferable features, improving cross-PUT strategies.

Acknowledgment

This work was supported by the Acquisition, Technology & Logistics Agency (ATLA) under the Innovative Science and Technology Initiative for Security 2020 (JPJ004596).

References

[1] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, 2022.

[2] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *USENIX Security Symposium*, N. Heninger and P. Traynor, Eds., 2019, pp. 1967–1983.

[3] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *The Annual Computer Security Applications Conference (ACSAC)*, 2020, pp. 360–372.

[4] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 2194–2211.

[5] Y. Koike, H. Katsura, H. Yakura, and Y. Kurogome, "Slopt: Bandit optimization framework for mutation-based fuzzing," in *The Annual Computer Security Applications Conference (ACSAC)*. New York, NY, USA: Association for Computing Machinery, 2022, p. 519 – 533.

[6] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A. Sadeghi, "DARWIN: survival of the fittest fuzzing mutators," in *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[7] Y.-F. Fu, J. Lee, and T. Kim, "autofz: automated fuzzer composition at runtime," in *USENIX Security Symposium*, 2023, pp. 1901–1918.

[8] "Fuzzbench: An open fuzzer benchmarking platform and service," https://fuzzbench.com.

[9] D. Zhang, A. Fioraldi, and D. Balzarotti, "On understanding and forecasting fuzzers performance with static analysis," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 3973–3987.

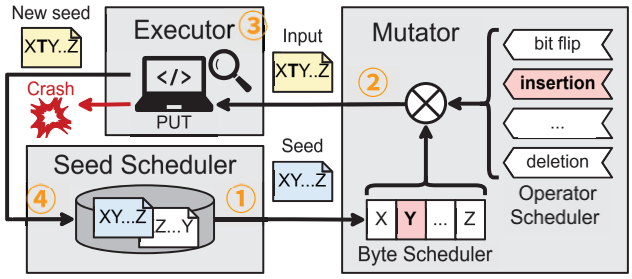
Dynamic Multi-Fuzzer Scheduling via Reinforcement Learning

Naomasa Matsubayashi, Yuichi Sugiyama
Ricerca Security, Inc.

RICERCA SECURITY

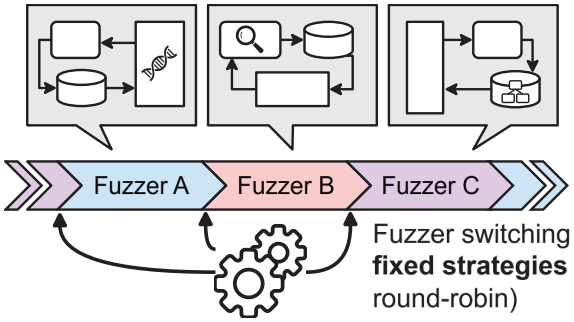
Introduction

① Select a seed to be mutated.
② Generate a new input to test the PUT.
③ Execute the input and collect the coverage.
④ Save inputs with increased coverage.



Each fuzzer has strengths and weaknesses on the PUT and the current exploration status.

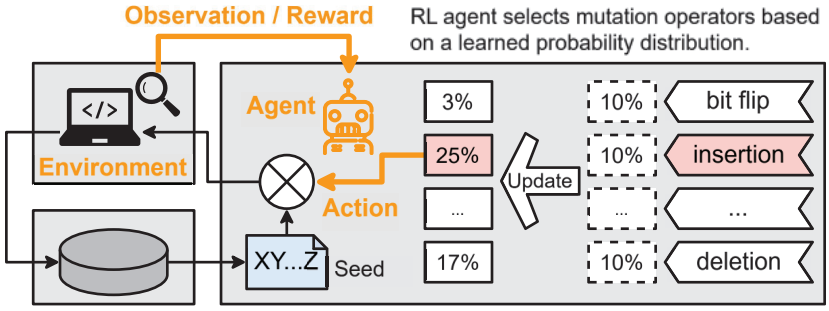
➡ ***Multi-Fuzzer Scheduling***: Switching fuzzers boosts coverage by combining their strengths.



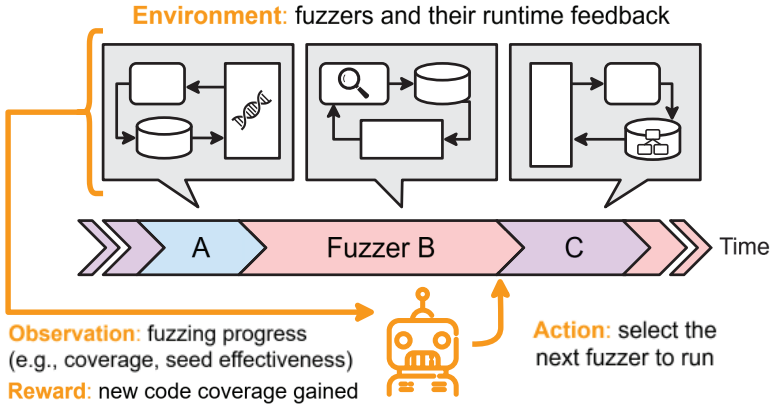
Challenge: Cannot change strategies at runtime based on various feedbacks (e.g., coverage statistics and PUT characteristics).

RL-based "Micro-level" Optimization

Reinforcement Learning (RL) is often applied to **part of the fuzzing components** (e.g., seed scheduler[1,2] or mutator[3,4]) to guide action selection based on runtime feedback.



Proposal: RL-based Multi-Fuzzer Scheduling



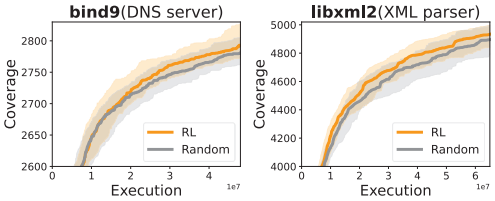
Contribution: This approach addresses the limitation of static strategies by applying RL to global multi-fuzzer scheduling, extending beyond local component optimization.

Implementation & Evaluation

- Switches between **AFL++ (mutation)** and **Angora (constraint-solving)**, leveraging complementary strengths.
- A policy learned via **offline RL** (optimizing for new seeds) selects the next fuzzer based on current fuzzing state.

RQ1: Same-PUT Performance

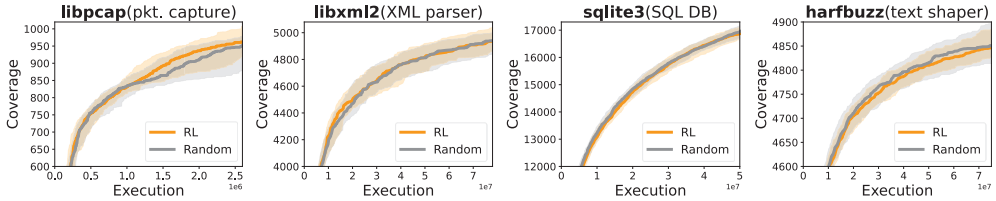
Training & Evaluation: Same PUT



RQ1: Improves same-PUT coverage.

RQ2: Cross-PUT Generalization

Training: bind9 (DNS server) / Evaluation: Other PUTs



RQ2: Limited transfer; possible for similar PUTs (e.g., bind9→libpcap).

Conclusion and Future Work

- RL-based scheduling improves same-PUT coverage.
- Cross-PUT transfer limited; possible among similar PUTs.
- Analyzing structural & behavioral similarities may improve transfer.

Reference

[1] Yue, Tai, et al. "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit." USENIX Security 2020.
[2] Wang, Jinghan, et al. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing." NDSS 2021.
[3] Wu, Mingyuan, et al. "One Fuzzing Strategy to Rule Them All." ICSE 2022.
[4] Koike, Yuki et al. "SLOPT: Bandit Optimization Framework for Mutation-Based Fuzzing." ACSAC 2022.