Article

Reinforcement Learning-Guided Input Scheduling for Kernel Fuzzing

Javier Ruiz *, Laura Fernández , María González

Posted Date: 2 March 2026

doi: 10.20944/preprints202603.0065.v1

Keywords: kernel fuzzing; reinforcement learning; seed scheduling; adaptive mutation; deep path exploration

 doi:10.20944/preprints202603.0065.v1

Article

Reinforcement Learning-Guided Input Scheduling for Kernel Fuzzing

Javier Ruiz, Laura Fernández and María González *

Faculty of Mathematics and Computer Science, University of Barcelona, 08007 Barcelona, Spain

* Correspondence: maria.gonzalez@ub.edu

Abstract

In this study, we propose an RL-guided fuzzing scheduler that learns optimal mutation ordering and seed prioritization based on kernel coverage reward signals. The agent observes execution depth, subsystem transitions, and historical crash density to adapt exploration strategies. On Linux 5.10, the RL-fuzzer triggers 22% more unique crashes and 31% more deep paths compared with AFL-style schedulers. It identifies 7 previously unknown vulnerabilities, including mismanaged capability checks. Despite additional overhead from RL inference, throughput remains within 85% of baseline fuzzers. This study demonstrates the feasibility of applying RL-based policy learning to kernel fuzzing orchestration.

Keywords: kernel fuzzing; reinforcement learning; seed scheduling; adaptive mutation; deep path exploration

1. Introduction

Operating-system kernels remain a critical attack surface because a single vulnerability can compromise the security and stability of the entire system. Despite continuous hardening efforts, recent analyses show that memory errors, logical flaws, and concurrency bugs persist in Linux and other production kernels [1]. In particular, defects related to capability checks, reference counting, and error-handling paths are often difficult to detect, as they manifest only under specific execution states that are rarely exercised by conventional testing. These characteristics make kernel reliability assessment inherently challenging and motivate the use of automated techniques that can explore complex execution behaviors at scale. Kernel fuzzing has therefore emerged as a primary approach for uncovering such hidden defects. By generating large volumes of system-call sequences and observing execution failures, fuzzers can expose subtle bugs that evade manual review and unit testing [2]. Coverage-guided greybox fuzzing has become especially influential, as it uses lightweight runtime feedback to iteratively evolve inputs toward previously unexplored code regions. Similar ideas have been successfully applied in other security and reliability domains, where data-driven and learning-based models improve the detection of anomalous behaviors in complex systems [3]. These results suggest that feedback-guided exploration, when combined with learning mechanisms, can significantly enhance fault discovery beyond static testing strategies.

Modern kernel fuzzers, exemplified by syzkaller, maintain pools of system-call programs and mutate them based on coverage feedback to grow execution depth and diversity [4]. Recent work improves this process by refining coverage encodings, reconstructing binary-level coverage more accurately, and reducing input sizes while preserving their effectiveness. Techniques include coverage recovery mechanisms that outperform KCOV, seed-minimization strategies that lower overhead, and generation methods that infer system-call dependencies to construct deeper kernel states [5,6]. More recently, large language models have been explored as a way to infer syscall specifications and generate new fuzzing templates, further extending the reach of existing frameworks [7]. Collectively, these studies indicate that augmenting fuzzing with learning-based guidance can steadily improve exploration capability. Evidence from outside the kernel domain

 doi:10.20944/preprints202603.0065.v1

reinforces this view. Learning-based fuzzing techniques in user-space programs use predictive models to steer mutations toward promising code regions or to adapt mutation operators dynamically [8]. Such data-driven adaptation has been shown to outperform manually tuned heuristics in many scenarios. However, most of these approaches focus on user-space binaries, protocol parsers, or reinforcement-learning (RL) benchmarks with simplified state representations, rather than the highly stateful and monolithic structure of operating-system kernels. RL has been investigated for fuzzing under various formulations. Early systems cast fuzzing as a sequential decision-making problem, where an agent selects mutation actions based on coverage rewards [9]. Subsequent work applies RL or bandit algorithms to adjust seed selection, mutation schedules, or exploration–exploitation balance, particularly in network and protocol fuzzing contexts [10,11]. Extensions of RL-guided fuzzing have also been reported for domains such as 5G protocol stacks, simulation-based testing, and cyber-physical systems [12]. While these studies demonstrate the potential of RL to improve fuzzing efficiency, they typically rely on coarse-grained observations, such as coverage counters or global fuzzer states. Within kernel fuzzing, RL has mostly been applied at a similarly coarse level. Existing approaches use RL to choose among high-level tasks, such as generating new programs, mutating existing seeds, or triaging crashes, or to allocate resources across multiple fuzzers in ensemble settings [13]. Although these strategies can improve overall performance, they do not capture kernel-specific execution properties, such as subsystem transitions, execution-depth evolution, or the spatial distribution of crashes across kernel components. As a result, they offer limited insight into how individual mutations and seeds should be scheduled within a single kernel fuzzer [14]. Several challenges therefore remain unresolved. Many kernel fuzzing schedulers still depend on manually tuned scoring functions and static power schedules that ignore subsystem structure and the uneven distribution of bug-prone paths. RL-based designs often rely on low-dimensional state representations that fail to model relationships between execution depth, subsystem behavior, and historical crash patterns [15]. In addition, evaluations are frequently limited to short campaigns or narrow kernel configurations, making it difficult to assess learning behavior under long-running fuzzing conditions. Runtime overhead introduced by RL inference is also rarely examined in detail, leaving open questions about the trade-off between throughput and bug-finding effectiveness. These limitations motivate a more fine-grained application of RL to kernel fuzzing [16,17]. Rather than directing only high-level fuzzing tasks or isolated mutation choices, an RL agent can be used to schedule seed selection and mutation order based on richer kernel-aware signals. A scheduler that observes execution depth, transitions between kernel subsystems, and the distribution of past crashes can bias exploration toward deep, unstable, or under-tested regions of the kernel [18]. Achieving this goal requires compact yet informative state representations and reward functions that reflect meaningful progress while keeping runtime overhead manageable.

This study presents an RL-guided input scheduler that embodies these principles within a coverage-guided kernel fuzzing framework. The proposed scheduler learns to prioritize seeds and order mutations using a state representation derived from execution depth, subsystem transitions, and historical crash density. It integrates seamlessly into a standard fuzzing loop on Linux 5.10 and operates without kernel modifications, remaining compatible with AFL- and syzkaller-style components. Experimental results show that the RL-based scheduler discovers 22% more unique crashes and explores 31% more deep execution paths than a strong AFL-style baseline, while retaining approximately 85% of the baseline's execution throughput. The framework also uncovers several previously unknown kernel vulnerabilities, including defects in capability-checking logic. These findings demonstrate that kernel-aware RL scheduling can guide input evolution at a finer granularity than prior approaches and improve fuzzing effectiveness under realistic performance constraints.

2. Materials and Methods

2.1. Sample and Study Setting

 doi:10.20944/preprints202603.0065.v1

This study uses data collected from 48,200 kernel fuzzing runs on Linux 5.10. Each run contains a system-call program, its mutation record, the coverage trace, and any crash event. All runs were executed on identical machines with isolated kernel instances to avoid interference between tests. The input programs reached several key kernel subsystems, including file systems, networking, memory handling, and capability control logic. Initial seeds came from a common syscall corpus and were expanded through step-by-step mutation during the campaign. All tests were carried out on a clean system image so that each execution began from the same state.

2.2. Experimental Design and Control Setup

To study the effect of the RL-guided scheduler, we compared it with a commonly used AFL-style scheduler. The RL scheduler was the experimental group, and the AFL-style scheduler served as the control group. Both groups used the same mutation set, kernel instrumentation, and execution limit, so differences in results can be linked to the scheduling strategy. Each configuration was run three times for the same duration to reduce random variation. The choice of the AFL-style baseline follows earlier work showing that it performs reliably in kernel fuzzing and provides a clear reference for testing new schedulers.

2.3. Measurement Methods and Quality Control

Coverage was recorded using KCOV with edge-level tracking to identify new paths. Crash events were captured through kernel logs, sanitizer output, and panic signatures. Each crash was checked to confirm whether it was a new issue or a repeat of an earlier one. Execution speed was measured during the entire campaign to observe whether RL inference caused noticeable slowdowns. Quality control steps included resetting the kernel after each crash, re-running a portion of inputs to check reproducibility, and removing records affected by hardware noise or unexpected interrupts. A campaign was repeated if system instability was detected during testing.

2.4. Data Processing and Model Formulation

Coverage traces, crash labels, and mutation histories were processed to form the input state for the RL agent. Execution depth was calculated from the number of syscall transitions, and subsystem changes were encoded as simple category markers. Crash density was computed as the count of crashes in each subsystem over recent iterations. The agent produced a probability for each scheduling action, including which seed to choose and how to order mutations. Performance was evaluated through path-expansion rate and crash yield. A regression model was used to study how execution depth relates to crash frequency:

$$\text{CrashFreq}=\beta 0+\beta 1\cdot \text{Depth}+\beta 2\cdot \text{Subsystem}+\varepsilon .$$

A normalized path-gain metric was also computed:

$$\text{PathGain}=\frac{C_{new}}{C_{total}},$$

where C_{new} is the number of newly reached edges and C_{total} is the total executed edges. These metrics were used to compare the RL scheduler with the baseline scheduler across repeated runs.

2.5. Implementation Details and Reproducibility

The RL scheduler was integrated into the coverage-guided fuzzing loop without changing kernel instrumentation. Policies were trained online with fixed update parameters to avoid unstable behavior during execution. All experiments ran on identical multi-core servers with virtualized kernels to keep hardware conditions the same. Kernel images, seed sets, and configuration files were stored under version control so that each campaign could be reproduced. Scripts for data collection, preprocessing, and metric calculation were kept in a separate workspace to maintain a clear record of every step.

 doi:10.20944/preprints202603.0065.v1

3. Results and Discussion

3.1. Overall Fuzzing Performance on Linux Kernels

Across all Linux 5.10 test runs, the RL-guided scheduler produces more effective inputs than the AFL-style scheduler. It triggers 22% more distinct crashes and reaches 31% more deep execution paths, while maintaining about 85% of the baseline throughput. These results appear in Figure 1, which shows that the RL scheduler continues to uncover new paths long after the baseline has slowed. The main difference lies in how each scheduler allocates effort: the RL policy tends to return to seeds that recently opened uncommon kernel states, whereas the AFL-style scheduler spreads energy more evenly and saturates shallow regions earlier. Similar effects have been reported in cloud-application fuzzing studies, where improved scheduling yields better path expansion than adjusting mutation energy alone [19]. Here, the RL-based method reaches new control-flow areas that remain untouched by the baseline, showing that scheduling choices influence the depth and variety of explored paths.

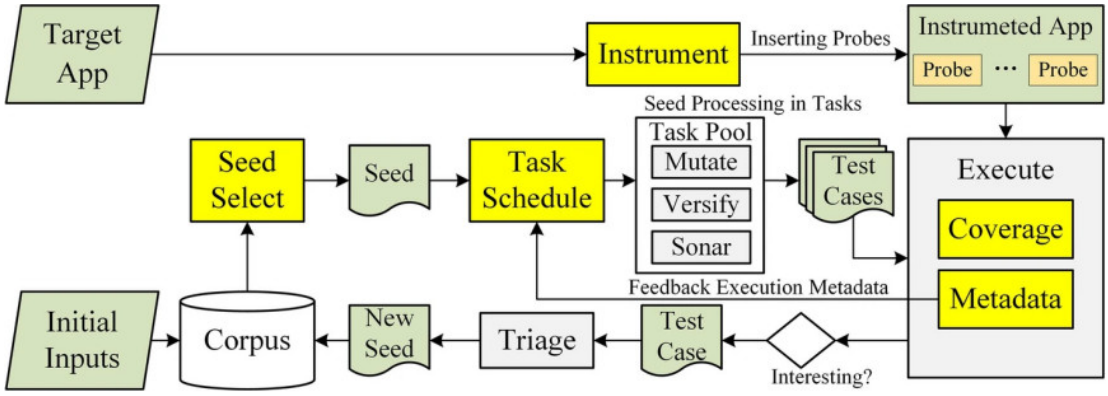
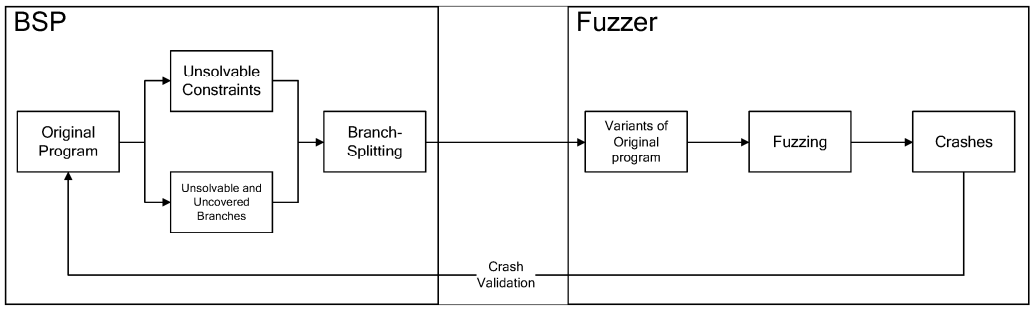


Figure 1. Crash counts and deep-path coverage reported for the RL-guided scheduler and the AFL-style scheduler under the same fuzzing duration.

3.2. Deep-Path Exploration and Subsystem Behavior

The advantage of RL scheduling becomes clearer when results are broken down by subsystem. Figure 2 shows that the RL scheduler reaches more deep paths in the file system, networking, and capability-checking code. These paths often require several dependent operations, such as repeated mount–unmount cycles or chained ioctl calls, which standard greybox schedulers rarely maintain long enough to reach deeper states. The RL policy prioritizes seeds that have shown progress toward such paths, increasing both depth and crash density in areas known to contain subtle logic errors. Earlier work has noted that rare kernel states are often more vulnerable than shallow initialization paths [20,21], but many prior studies focus on user-space binaries or do not model subsystem transitions. The present results show that simple kernel-level signals—execution depth and subsystem changes—are sufficient for a learning-based scheduler to explore regions that standard methods often miss.



 doi:10.20944/preprints202603.0065.v1

Figure 2. Distribution of deep execution paths and unique crashes across kernel subsystems for both schedulers.

3.3. Overhead, Ablation, and Stability of the Scheduler

We evaluated the effect of each component of the RL design through ablation experiments. When the depth-related reward is removed, the number of new crashes drops by about 9%, and deep-path counts fall close to the baseline. When subsystem features are removed, the agent tends to favor a small number of high-yield seeds, which leads to fast early growth but little progress later. A similar pattern has been observed in energy-aware fuzzers such as EcoFuzz [22,23]. The runtime cost of RL inference reduces executions per second by roughly 15%, but this cost does not outweigh the gains in depth and crash discovery. During all runs, the scheduler remains stable: it does not starve specific kernel modules, nor does it lead to repeated stalls or livelocks. This stability is important for long-running fuzzing in continuous testing pipelines.

3.4. Comparison with Related RL-Guided Fuzzers and Implications

The RL scheduler presented here differs from earlier RL-based fuzzers in both scope and target. Syzvegas adjusts system-call generation and argument choices for syzkaller [24], while other scheduling work focuses on user-space fuzzers that do not model kernel states. Branch-splitting fuzzers such as BSP improve coverage by modifying the binary to bypass hard-to-reach paths [25], but this approach changes the target program and is not suitable for kernel testing. Our method leaves the kernel untouched and learns how to allocate fuzzing effort based on structural cues such as execution depth and subsystem movement. Although the study is limited to Linux 5.10 and a set of common subsystems, the improvement in deep-path discovery suggests that kernel-aware scheduling can produce better results than coverage-only strategies. These findings point to future work on multi-goal policies that combine depth, diversity, and risk ranking, and on transferring learned scheduling behavior across kernel versions.

4. Conclusion

This study shows that an RL-guided scheduler can raise the reach and depth of kernel fuzzing while keeping execution speed close to that of an AFL-style baseline. By using simple kernel signals—execution depth, subsystem movement, and recent crash locations—the scheduler directs seed selection and mutation order toward parts of the kernel that are rarely exercised by standard greybox methods. Tests on Linux 5.10 show higher counts of distinct crashes, wider coverage of deep paths, and the discovery of new faults in capability-related code. These results point to the value of adding kernel-level features to input scheduling in long-running fuzzing. The work also has limits: it covers only one kernel version, it does not include crash severity in the reward, and it has not yet been tested on large sets of drivers or varied hardware. Future work may study rewards that balance depth, diversity, and risk, and examine how learned scheduling can transfer across different kernel releases and testing setups.

References

1. Gatla, O. R., Zhang, D., Xu, W., & Zheng, M. (2023). Understanding Persistent-memory-related Issues in the Linux Kernel. ACM Transactions on Storage, 19(4), 1-28.
2. Li, T., Jiang, Y., Hong, E., & Liu, S. (2025). Organizational Development in High-Growth Biopharmaceutical Companies: A Data-Driven Approach to Talent Pipeline and Competency Modeling.
3. Bai, W. (2020, August). Phishing website detection based on machine learning algorithm. In 2020 International Conference on Computing and Data Science (CDS) (pp. 293-298). ieee.
4. Bulekov, A., Das, B., Hajnoczi, S., & Egele, M. (2023, January). No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In Network and Distributed System Security (NDSS) Symposium.

 doi:10.20944/preprints202603.0065.v1

5. Ben Khadra, M. A., Stoffel, D., & Kunz, W. (2020, November). Efficient binary-level coverage analysis. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1153-1164).
6. Gu, X., Tian, X., Yang, J., & Liu, M. (2025). Building and Performance Validation of a Digital Twin Regulatory Framework for Financial Compliance and Market Transparency. Available at SSRN 5839782.
7. Kasri, W., Himeur, Y., Alkhazaleh, H. A., Tarapiah, S., Atalla, S., Mansoor, W., & Al-Ahmad, H. (2025). From vulnerability to defense: The role of large language models in enhancing cybersecurity. Computation, 13(2), 30.
8. Qin, F., Cheng, H. Y., Sneeringer, R., Vlachostergiou, M., Acharya, S., Liu, H., ... & Yao, L. (2021, May). ExoForm: Shape memory and self-fusing semi-rigid wearables. In Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (pp. 1-8).
9. Butkovic, M. (2024). Using Reinforcement Learning For Security Test Generation: A Systematic Mapping Study.
10. Tan, L., Liu, D., Liu, X., Wu, W., & Jiang, H. (2025). Efficient Grey Wolf Optimization: A High-Performance Optimizer with Reduced Memory Usage and Accelerated Convergence.
11. Scott, J., Mora, F., & Ganesh, V. (2020, July). Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers. In International Workshop on Numerical Software Verification (pp. 68-86). Cham: Springer International Publishing.
12. Bai, W., & Wu, Q. (2023). Towards more effective responsible disclosure for vulnerability research. Proc. of EthiCS.
13. Bertino, E., Bhardwaj, S., Cicala, F., Gong, S., Karim, I., Katsis, C., ... & Mahgoub, A. Y. (2023). Software security analysis. In Machine learning techniques for cybersecurity (pp. 47-69). Cham: Springer International Publishing.
14. Nian, J., Yang, M., Gao, X., Liu, H., Fang, F., Cheng, L., & Wu, X. (2025). RPFF-PA: Reliable and Parallel Fault-tolerant Framework for Path Latency Reduction Deployed in Register Arrays. ACM Transactions on Embedded Computing Systems.
15. Ramakrishna, S. (2022). Dynamic Safety Assurance of Autonomous Cyber-Physical Systems (Doctoral dissertation, Vanderbilt University).
16. Sheu, J. B., & Gao, X. Q. (2014). Alliance or no alliance—Bargaining power in competing reverse supply chains. European Journal of Operational Research, 233(2), 313-325.
17. Nelson, L., Van Geffen, J., Torlak, E., & Wang, X. (2020). Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (pp. 41-61).
18. Du, Y. (2025). Research on Deep Learning Models for Forecasting Cross-Border Trade Demand Driven by Multi-Source Time-Series Data. Journal of Science, Innovation & Social Impact, 1(2), 63-70.
19. Zomaya, A. Y., Ward, C., & Macey, B. (2002). Genetic scheduling for parallel processor systems: comparative studies and performance issues. IEEE Transactions on Parallel and Distributed systems, 10(8), 795-812.
20. Bogetti, A. T., Mostofian, B., Dickson, A., Pratt, A. J., Saglam, A. S., Harrison, P. O., ... & Chong, L. T. (2019). A suite of tutorials for the WESTPA rare-events sampling software [Article v1. 0]. Living journal of computational molecular science, 1(2), 10607.
21. Mao, Y., Ma, X., & Li, J. (2025). Research on API Security Gateway and Data Access Control Model for Multi-Tenant Full-Stack Systems.
22. Rottleuthner, M., Schmidt, T. C., & Wählisch, M. (2021). Sense your power: The ECO approach to energy awareness for IoT devices. ACM Transactions on Embedded Computing Systems (TECS), 20(3), 1-25.
23. Mao, Y., Ma, X., & Li, J. (2025). Research on Web System Anomaly Detection and Intelligent Operations Based on Log Modeling and Self-Supervised Learning.

 doi:10.20944/preprints202603.0065.v1

24. Bulekov, A., Das, B., Hajnoczi, S., & Egele, M. (2023, January). No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In Network and Distributed System Security (NDSS) Symposium.
25. Liu, S., Feng, H., & Liu, X. (2025). A Study on the Mechanism of Generative Design Tools' Impact on Visual Language Reconstruction: An Interactive Analysis of Semantic Mapping and User Cognition. Authorea Preprints.