Seed Scheduling in Fuzz Testing as a Markov Decision Process

Rafael F. Cunha[1,*], Luca Müller[2], Thomas Rooijakkers[3], Puck de Haan[3] and Fatih Turkmen[1]

[1]University of Groningen, The Netherlands
[2]Fraunhofer Institute for Secure Information Technology SIT, Germany
[3]TNO, The Netherlands

Abstract

Coverage-guided Greybox Fuzzing (CGF) is an effective method for discovering software vulnerabilities. Traditional fuzzers, such as American Fuzzy Lop (AFL), rely on heuristics for critical tasks like seed scheduling, which often lack adaptability and may not optimally balance exploration with exploitation. This paper presents a novel approach to enhance seed scheduling in CGF by formalizing it as a Markov Decision Process (MDP). We detail the design of this MDP, including the state representation derived from fuzzer and coverage data, the action space encompassing seed selection and power assignment, and a reward function geared towards maximizing coverage and bug discovery. A Proximal Policy Optimization (PPO) agent is then trained to learn a scheduling policy from this MDP within the AFL++ fuzzer. Our investigation into this Deep Reinforcement Learning (DRL) based approach reveals that while the MDP formulation provides a structured framework, practical application faces significant challenges, including high computational demands for training and intensive hyperparameter tuning. The key contributions of this work are: (1) a concrete MDP formulation for the complex task of fuzzer seed scheduling, (2) an analysis of the inherent difficulties and trade-offs in applying DRL to this specific domain, and (3) insights gained from the agent's learning process (or lack thereof), which inform the discussion on the suitability of DRL for this type of optimization problem in fuzzing. This research provides a foundational exploration of DRL for seed scheduling and highlights critical considerations for future advancements in intelligent fuzzing agents.

1. Introduction

Coverage-guided Greybox Fuzzers (CGFs), exemplified by tools like American Fuzzy Lop (AFL) [1], are highly effective for uncovering software vulnerabilities. They operate by mutating a corpus of 'seed' inputs to generate new test-cases, retaining those that achieve new program coverage to guide further exploration. While the general idea is straightforward, CGFs are often compute-intensive, and their efficiency heavily relies on making intelligent decisions throughout the fuzzing process [2].

A critical stage amenable to optimization is *seed scheduling*. This encompasses two interdependent decisions: (i) selecting the next seed input from the corpus for mutation, and (ii) determining the amount of fuzzing effort (or 'energy') to allocate to the chosen seed. This task inherently involves a trade-off between *exploration* (testing diverse seeds to find new program areas) and *exploitation* (focusing on seeds that have historically been productive). Current fuzzers predominantly employ heuristics for these decisions, which, while often effective, may lack adaptability and can be suboptimal for specific programs or evolving fuzzing campaign states.

To address these limitations, this work investigates the application of Deep Reinforcement Learning (DRL) to optimize seed scheduling in CGFs. While machine learning (ML) has increasingly found applications in the fuzzing domain [3, 4, 5, 6], we specifically propose to formalize the seed scheduling problem as a Markov Decision Process (MDP) [7]. An MDP provides a principled framework for sequential decision-making, allowing an agent to learn a policy that maximizes a cumulative reward. DRL, which combines Reinforcement Learning (RL) [8] with artificial neural networks (ANNs), offers the potential to learn complex policies in environments with large state and action spaces, such as those encountered in fuzzing.

SPAIML'25: International Workshop on Security and Privacy-Preserving AI/ML, October 26, 2025, Bologna, Italy
*Corresponding author.
r.f.cunha@rug.nl (R. F. Cunha)

This paper details our approach to modeling seed scheduling as an MDP and the subsequent attempt to train a DRL agent, specifically using Proximal Policy Optimization (PPO), to learn an effective scheduling policy. The primary contributions of this work are: First, we present a novel MDP formulation tailored for CGF seed scheduling, elaborating on the design of the state space (derived from fuzzer telemetry and coverage information), the action space (representing choices of seeds and their mutation energy), and the reward function (aimed at encouraging new coverage and bug discovery). Second, we provide an in-depth analysis of the significant challenges encountered when applying DRL to this specific fuzzing problem. These challenges include the design of a Markovian state representation, the non-stationarity of the fuzzing environment, the vast action space if selecting seeds directly, the practical difficulties of online training within a live fuzzer, and the intensive hyperparameter tuning required for DRL agents. Third, based on our experiences and the performance of the trained agent, we offer insights into the suitability and current limitations of DRL for tackling the seed scheduling task in its current common CGF implementations. This research contributes to the understanding of how DRL can be applied to complex decision-making problems in software security and highlights key areas for future research in developing more intelligent and adaptive fuzzing systems.

2. Background and Related Work

This section provides essential background on Coverage-Guided Fuzzing (CGF) and Markov Decision Processes (MDPs). It then reviews related applications of Reinforcement Learning in fuzzing to contextualize the novelty of our approach to seed scheduling.

2.1. Coverage-Guided Greybox Fuzzing (CGF)

Fuzzing is a dynamic software testing technique renowned for its effectiveness in finding bugs and vulnerabilities [9]. Coverage-guided greybox fuzzers, such as AFL++ [10], represent a particularly successful class. CGFs iteratively mutate a set of input files, known as 'seeds', to generate new test-cases. These test-cases are executed against the Program Under Test (PUT), and feedback, typically in the form of code coverage (e.g., edge coverage in a Control-Flow Graph (CFG)), is used to guide the fuzzing process. Inputs that discover new coverage are added to the seed corpus for further mutation, aiming to progressively explore more program states [1]. The selection of which seed to fuzz next and for how long (seed scheduling) is a critical heuristic that significantly impacts CGF performance. Our work focuses specifically on modeling this seed scheduling aspect within a CGF like AFL++.

2.2. Markov Decision Processes (MDPs) and Reinforcement Learning (RL)

Reinforcement Learning (RL) is a paradigm where an agent learns to make optimal sequential decisions by interacting with an environment and receiving scalar reward signals [8]. Many RL problems are formalized using a Markov Decision Process (MDP) [7]. An MDP is typically defined by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{P}(s'|s,a)$ is the state transition probability function, $\mathcal{R}(s,a,s')$ is the reward function, and $\gamma \in [0,1]$ is a discount factor. The agent's goal is to learn a policy $\pi(a|s)$ that maximizes the expected cumulative discounted reward. Deep Reinforcement Learning (DRL) utilizes deep neural networks to scale RL to complex problems. Our work employs DRL to learn a seed scheduling policy by modeling the process as an MDP, with a key aspect being the definition of the program's coverage map as the state $s \in \mathcal{S}$.

2.3. Reinforcement Learning in Fuzzing

The application of Reinforcement Learning (RL) to enhance fuzzing effectiveness is an active research area [3]. Seed scheduling, in particular, significantly impacts fuzzer performance [11, 12, 13, 14, 15], making it a prime candidate for RL-based optimization.

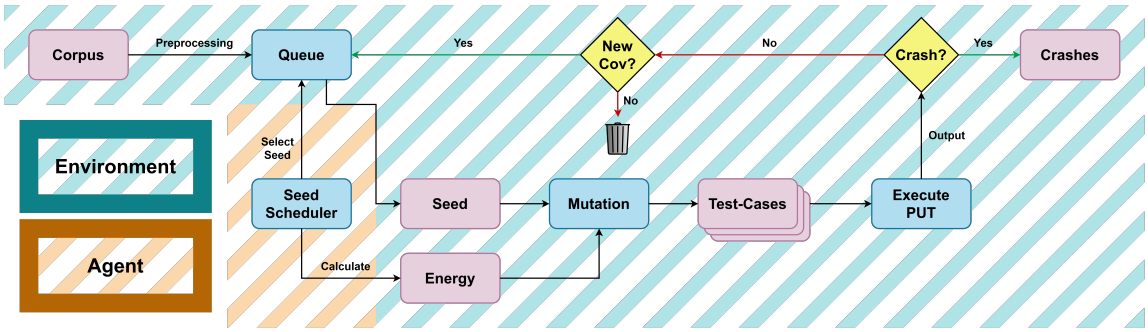


Figure 1: Agent-Environment boundary in our MDP formulation, where the agent is the seed scheduler.

Existing ML-based approaches to *seed scheduling* have predominantly utilized Multi-Armed Bandit (MAB) models [16]. These include T-Scheduler [17] using Thompson sampling [18], AFL-HIER [15] employing UCB1 [19], EcoFuzz [20] using adversarial bandits, LinFuzz [21] with contextual bandits [22], and SLIME [23] utilizing UCB-V [24]. While effective, MABs typically optimize for immediate rewards and do not inherently model the full state transitions of a fuzzing campaign.

Deep Reinforcement Learning (DRL) has also been applied, primarily focusing on the *mutation strategy*. For example, Böttinger et al. [25] utilized DRL to select mutation operators, and similar DRL-based mutation optimization was explored by Zhang et al. [26] and Shao et al. [27]. These works address a different aspect of fuzzing than seed scheduling.

To our knowledge, Choi et al. [28] presented the only prior attempt to formalize seed scheduling for CGF as an MDP and apply RL. Their approach modeled individual test-cases as states and used Temporal-Difference TD(0) [8] to learn state values, guiding seed selection and energy assignment. However, their formulation presents several challenges regarding standard MDP assumptions. For instance, the action space size (number of seeds in the queue) can vary over time, potentially violating the requirement for a fixed action set or a consistent mapping for policy approximation. Furthermore, their reward mechanism and state transition definition (where multiple new inputs from one seed are considered next states) could lead to non-Markovian dynamics and difficulties in ensuring convergence of value estimates.

This paper builds upon the promise of RL for fuzzing but specifically addresses the seed scheduling task by proposing a novel MDP formulation where the global coverage map constitutes the state. This aims to provide a more holistic and Markovian representation of the fuzzing campaign's progress for DRL-based policy learning, distinct from prior MAB approaches and addressing the formulation challenges noted in previous MDP attempts for scheduling.

3. Our Approach: MDP for Fuzzing Seed Selection

To address the limitations of heuristic-based seed scheduling in CGFs, we propose a novel approach based on DRL. This involves formalizing the seed scheduling task—encompassing both seed selection and mutation energy assignment—as an MDP. Our core contribution lies in this formulation, particularly the use of the global coverage map as the primary state representation. We implement this within AFL++ [10] and train a DRL agent using the Proximal Policy Optimization (PPO) algorithm [29] to learn an optimal scheduling policy.

Central to our MDP formulation is the definition of the agent and its environment. As depicted in Figure 1, the DRL **agent** embodies the seed scheduler. Its role is to observe the current state of the fuzzing campaign and decide which seed to fuzz next and with what energy. The **environment** comprises all other components of the fuzzer (e.g., mutation engine, AFL++'s core loop) and the Program Under Test (PUT). The agent learns by receiving (state, reward) feedback from this environment.

3.1. MDP Formulation Details

Our MDP is defined by the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$ each of which we summarize below.

3.1.1. State Space ($\mathcal{S}$)

A critical design choice is the state representation $s \in \mathcal{S}$. The primary component is AFL++'s global **coverage bitmap** (specifically, the `virgin_bits` array). This bitmap indicates which program edges have been covered and, for each edge, which hit-count bins (e.g., hit 1x, 2x, 3x, 4-7x, etc. [1]) have been achieved. Each edge's coverage status is typically represented by a byte, providing a fine-grained view of exploration. Figure 2 visualizes this structure. Details of the hit-count bins are provided in Appendix C. This global coverage map, of size N_{edges} (the number of instrumented edges in the PUT), forms a fixed-size vector $[0\text{x}00, 0\text{xFF}]^{N_{edges}}$ and serves as the core observation for the agent.

To allow the agent to potentially adapt its strategy over the duration of a fuzzing campaign, we make the MDP **time-aware** [30] by including the *remaining allocated fuzzing time* (in seconds) as an additional scalar component in the state s. This aims to provide context about the urgency or remaining opportunity for exploration. The combined state (coverage map + remaining time) is designed to be Markovian, providing sufficient information for decision-making.

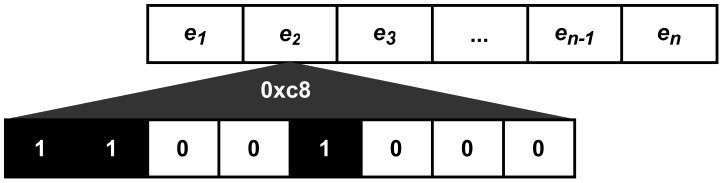


Figure 2: Coverage map state structure.

3.1.2. Action Space ($\mathcal{A}$)

The agent's action a_t at each time step t is a tuple (e_t, p_t), controlling two aspects of seed scheduling:

- **Edge Selection (e_t):** The agent selects an edge e_t from the set of currently *covered* edges in the program's Control-Flow Graph (CFG), as indicated by the state's coverage map. This selected edge e_t is then used as an index into AFL++'s `top_rated` list to retrieve the actual seed input to be fuzzed. This indirect selection mechanism ensures that the action component for seed choice is bounded by N_{edges}, the size of the coverage map, rather than the potentially unbounded and dynamic size of the seed queue. Invalid action masking [31] is applied to the policy network's output to ensure that only currently covered edges can be selected.
- **Mutation Energy (p_t):** The agent also selects a mutation energy p_t to be assigned to the chosen seed. This energy value, determining the number of mutations to generate, is chosen from a predefined discrete set, typically ranging from 10% to 300% of AFL++'s default energy allocation for a seed.

The full action space for a given state s is the Cartesian product of the available covered edges and the set of discrete energy levels:

$$\mathcal{A}(s) \doteq \mathcal{A}_{\text{edge}}(s) \times \mathcal{A}_{\text{power}}(s), \text{ for all } s \in \mathcal{S}. \tag{1}$$

For some experiments, the agent was configured to only perform edge selection (e_t), while relying on AFL++'s default power scheduling mechanisms for p_t.

3.1.3. Reward Function ($\mathcal{R}$)

The reward function r_t is designed to encourage the agent to select seeds and energy levels that efficiently lead to the discovery of new program behaviors. After the fuzzer completes a round of mutations on the seed chosen via action $a_t = (e_t, p_t)$, the reward is calculated as the ratio of generated test-cases that were marked "interesting" (i.e., triggered new edge coverage or hit a new hit-count bin) to the total number of test-cases generated in that round:

$$r_t = \frac{|\{x \in X'_t \mid x \text{ is interesting}\}|}{|X'_t|}, \tag{2}$$

where X'_t is the set of all test-cases generated from the seed associated with a_t. A reward closer to 1 indicates a highly efficient scheduling decision for that step.

3.1.4. Transition Probabilities ($\mathcal{P}$) and Discount Factor (γ)

The state transition probability function $\mathcal{P}(s_{t+1}|s_t, a_t)$ is implicitly defined by the complex interactions within the CGF (mutation, execution, coverage update) and the PUT. As is common in DRL applied to complex systems, $\mathcal{P}$ is not explicitly modeled; the agent learns a policy in a model-free manner. We use a discount factor $\gamma = 0.99$ to value future rewards.

3.1.5. Episode Definition and MDP Dynamics

Theoretically, our MDP has a finite horizon, as an episode would terminate upon achieving complete coverage of the PUT, which is a finite set of edges. However, in practical fuzzing scenarios, complete coverage is rarely attained, leading to what is effectively a single, very long episode for the entire fuzzing campaign. This presents a significant challenge for DRL algorithms, which typically learn from experience aggregated over many shorter episodes.

Furthermore, a key characteristic of our state representation (the global coverage map) is that it is largely non-decreasing. Newly discovered coverage updates the bitmap, but existing coverage is persistent. Consequently, the agent generally cannot revisit exact past states that represented a lesser degree of coverage. This results in a predominantly unidirectional exploration of the state space, as illustrated in Figure 3.

To facilitate DRL training by providing more frequent learning updates and varied experiences in the face of a potentially single, long fuzzing episode, we implemented a **multi-queue strategy**. In this setup, each initial seed (or a small group of initial seeds) forms the basis of an independent "fuzzing group." Each group maintains its own coverage map context relative to its initial seed(s), and the DRL agent cycles through these groups (e.g., via round-robin scheduling). Interactions within each group are then treated as part of a shorter, distinct episode for learning updates. New inputs found to be interesting within a group's context are added to that group's queue. Furthermore, with a small probability, a particularly fruitful input might be used to initialize an entirely new group. This approach aims to provide the necessary episodic structure crucial for many DRL algorithms. A schematic of this multi-queue mechanism is provided in Appendix E (Figure 8).

3.2. DRL Agent Design and System Integration

3.2.1. Algorithm Choice: PPO

We selected Proximal Policy Optimization (PPO) [29] as our DRL algorithm. PPO is a policy gradient method well-regarded for its balance of sample efficiency, ease of implementation, and stable performance across a range of tasks. It works by optimizing a clipped surrogate objective function designed to prevent excessively large policy updates, thereby promoting more stable learning. PPO typically incorporates Generalized Advantage Estimation (GAE) [32] for lower variance advantage estimates, a value function loss to learn a state-value baseline, and an entropy bonus to encourage exploration. (The detailed PPO objective functions are provided in Appendix A).

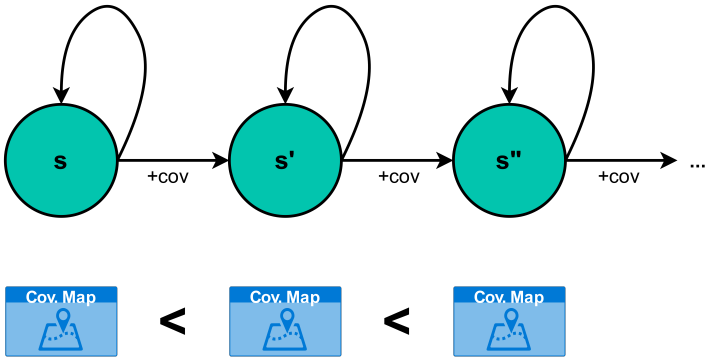


Figure 3: State exploration dynamic.

3.2.2. Network Architecture and Training

The policy (actor) network, which outputs probabilities for edge selection and a distribution for energy choice, and the value (critic) network, which estimates the state value, are both implemented as Multi-Layer Perceptrons (MLPs). Each network consists of two hidden layers with 64 units per layer, utilizing the hyperbolic tangent (tanh) as the activation function. The Adam optimizer [33] is employed for updating the network parameters. Specific hyperparameters used for training, such as learning rates, batch sizes, and PPO-specific coefficients, are detailed in Appendix B.

3.2.3. Integration with AFL++ via a Custom Gym Environment

To enable interaction between our Python-based DRL agent—built using the Proximal Policy Optimization (PPO) implementation from the Stable-Baselines3 library [34]—and the AFL++ fuzzer (written in C), we developed a custom environment compatible with the Gymnasium API [35]. This "Gym wrapper" acts as an intermediary, managing Inter-Process Communication (IPC) through message queues. At each step:

1. AFL++ pauses its main loop and sends the current state (coverage bitmap and remaining time) to the Gym environment via IPC.
2. The Gym environment passes this state to the DRL agent.
3. The agent selects an action (chosen edge and energy level) based on its current policy.
4. This action is sent back to AFL++ via IPC.
5. AFL++ executes the fuzzing round with the agent-specified seed and energy, calculates the reward based on the outcome, and determines the next state.
6. The new state and reward are sent back to the agent for learning.

This architecture, illustrated in Figure 4, allows the DRL agent to control AFL++'s seed scheduling. A detailed description can be found in Figure 7 in the Appendix.

4. Experimental Evaluation

This section details the experimental setup used to evaluate our DRL-based seed scheduling approach and presents the results. The primary goal was to assess whether the PPO agent, guided by our MDP formulation, could learn an effective seed scheduling policy and how its performance compared to baseline AFL++ heuristics.

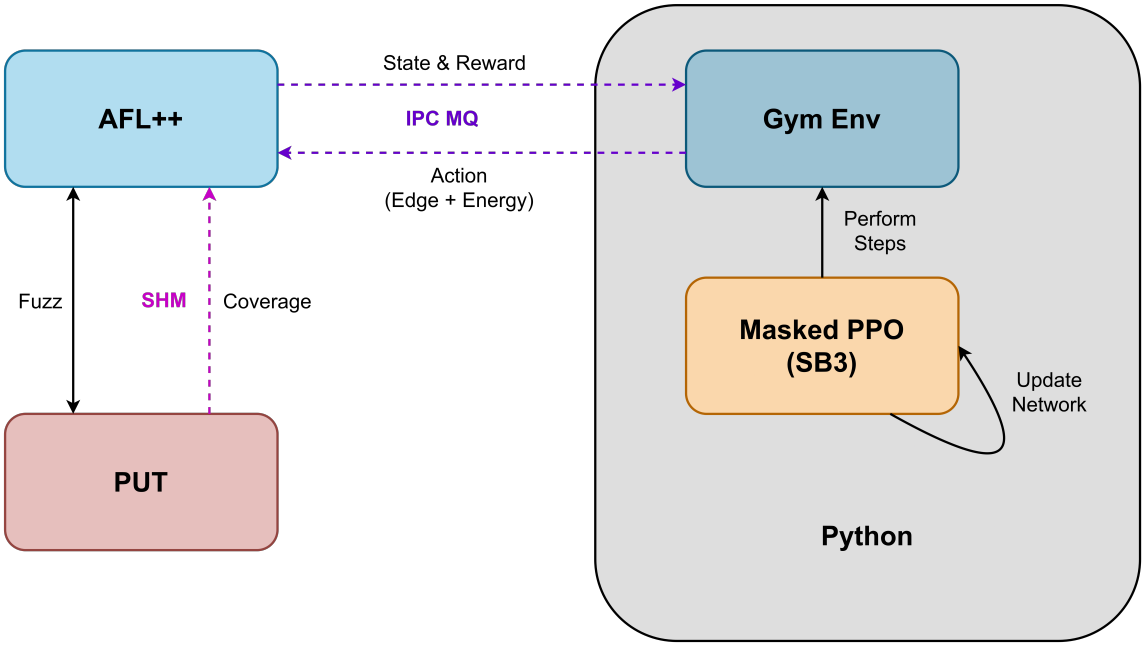


Figure 4: System architecture: DRL agent, Gym environment, IPC, and AFL++.

4.1. Experimental Setup

Target Program and Fuzzer. All experiments were conducted by fuzzing the `pdftotext` utility (version 3.02 from the xpdf suite), a real-world program that parses PDF files and extracts text. We used AFL++ version 4.10c [10] as the base fuzzer, modified to integrate our DRL agent as described in Section 3.2.

DRL Agent Configuration. We used PPO as the seed scheduling agent. The neural network architecture and key PPO hyperparameters are detailed in Section 3.2 and Appendix B, respectively. Training utilized the multi-queue strategy (described in Section 3.1.5 and visualized Figure 8 in Appendix E) to generate multiple learning episodes from a single fuzzing campaign.

Experimental Variants. We evaluated the following configurations:

- **AFL++ Default**: Standard AFL++ using its default `explore` power schedule.
- **DRL-Full**: Our PPO agent controlling both seed selection (edge choice) and mutation energy assignment.
- **DRL-Explore**: Our PPO agent controlling only seed selection (edge choice), while mutation energy was determined by AFL++'s `explore` power schedule.

Metrics and Procedure. Each fuzzing campaign was run for eight hours. Performance was primarily evaluated based on edge coverage achieved over time. We also monitored executions per second, the number of test-cases saved to the queue, and the maximum depth (number of ancestors) of test-cases.

4.2. Results and Analysis

4.2.1. DRL Agent Training Performance

The training dynamics of the PPO agent were monitored throughout the eight hour fuzzing campaigns. Figure 5 presents typical training metrics (value loss, policy loss, and explained variance) for four experimental runs (two for DRL-Full, two for DRL-Explore) using the multi-queue setup with a horizon of 32 steps and a learning rate of 0.001.

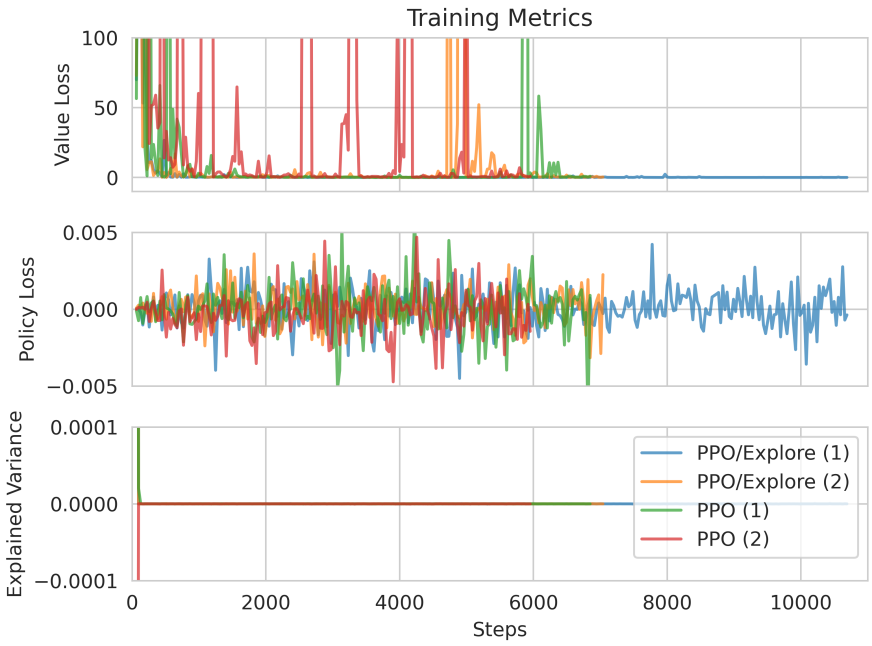


Figure 5: Training metrics for PPO agents fuzzing `pdftotext`. PPO/Explore refers to DRL-Explore. There is no clear convergence towards improvement.

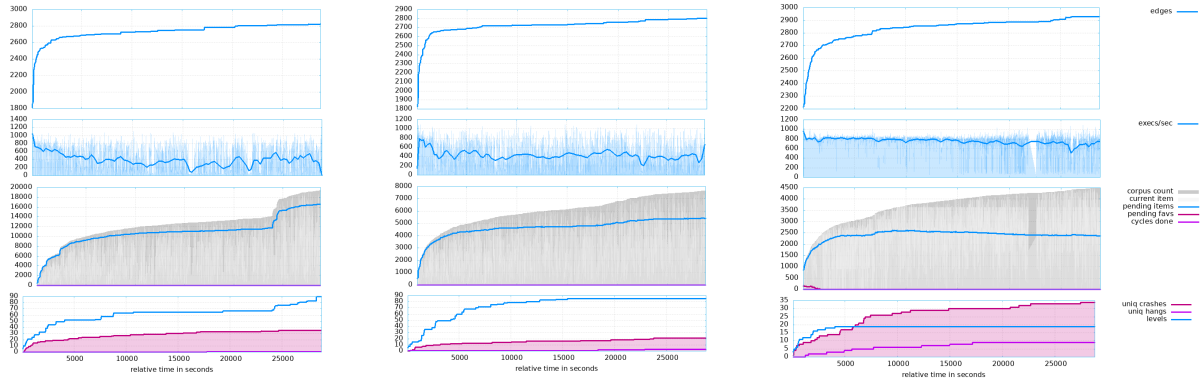
As shown in Figure 5, while the state value loss (L^{VF}) sometimes exhibited temporary decreases, the policy's surrogate loss (L^{CLIP}) did not show clear convergence towards improvement. The explained variance remained near-zero throughout the fuzzing campaigns for all DRL agent configurations. This indicates that the value function was not accurately predicting returns, and consequently, the agent struggled to learn a consistently better policy than random exploration or simple heuristics within the allocated time and data.

4.2.2. Comparative Fuzzing Performance

Figure 6 compares key performance metrics—edge coverage over time, executions per second, corpus, crash, and depth information—for the DRL variants against the AFL++ default over an eight hour period on `pdftotext`.

As shown in the first row of Figure 6, the default AFL++ configuration achieved approximately 4% more edge coverage after eight hours compared to both DRL variants. The executions per second (second row of Figure 6) were, as expected, higher for the default AFL++ due to the overhead introduced by the DRL agent interaction and IPC.

The DRL variants saved significantly more test-cases (approx. 4x for DRL-Full, 2x for DRL-Explore) compared to default AFL++. This is primarily an artifact of the multi-queue training strategy: since each "group" in the multi-queue maintains its own coverage context, inputs are deemed "new" or "interesting" relative to that smaller context, leading to more frequent additions to the respective group's queue. Similarly, the maximum depth of test-case ancestry was higher for the DRL variants. However, these effects on corpus statistics did not translate into superior overall coverage or bug finding performance within the eight hour timeframe.



(a) Full DRL seed scheduling (seed selection and power scheduling).

(b) DRL seed selection with AFL++'s `explore` power schedule.

(c) AFL++ with default settings (`explore` power schedule).

Figure 6: Experimental results for three fuzzing configurations on `pdftotext` for eight hours each. The figure is organized into a 4-row, 3-column grid. The columns correspond to different configurations: (a) Full DRL scheduling, (b) DRL seed selection with AFL++'s `explore` power schedule, and (c) AFL++ with default settings (`explore` power schedule). Each column (from top to bottom) displays four plots representing: (i) edge coverage over time, (ii) executions per second, (iii) corpus information, and (iv) crash & depth information.

5. Discussion and Conclusion

This paper explored the formalization of coverage-guided fuzzer seed scheduling as a Markov Decision Process (MDP) and the application of Deep Reinforcement Learning (DRL) to learn an optimal scheduling policy. Our primary contribution lies in the proposed MDP formulation, utilizing the global coverage map as state, and the subsequent analysis of applying DRL to this complex, dynamic environment.

Modeling seed scheduling as an MDP presents inherent challenges, including defining a Markovian state representation that captures sufficient history for reward calculation, and designing an action space that is both expressive and bounded for neural network agents. Our approach addressed these by using the coverage map and remaining time as state, and by mapping edge selections to seeds via AFL++'s `top_rated` list. However, training the DRL (PPO) agent proved difficult. Key reasons include: (1) the inability to pre-train policies due to the program-specific nature of the coverage map state; (2) the effectively single, long episode of a typical fuzzing campaign, requiring the agent to generalize rapidly; (3) the slow execution of actions (a full fuzzing round per seed), limiting data collection for learning; and (4) the extensive computational resources required for effective hyperparameter tuning in such a noisy, high-variance environment. Consequently, our implemented agent did not demonstrate significant policy improvement over baseline heuristics.

These challenges highlight a contrast with DRL applications for fuzzing mutation strategies, where the action space is often smaller and knowledge might be more transferable. Simpler Reinforcement Learning models, such as Multi-Armed Bandits (MABs) [17, 23, 21, 20, 15], offer less modeling complexity for seed scheduling by focusing on immediate rewards. While they forego explicit long-term planning, the difficulty in defining predictable long-term consequences in the chaotic fuzzing environment makes it debatable whether the complexity of a full MDP for scheduling is always warranted without further advances in state representation or learning algorithms.

Conclusion and Future Work. Our exploration suggests that while formalizing seed scheduling as an MDP is feasible, effectively training current DRL agents on this formulation within practical CGF setups faces substantial hurdles related to sample efficiency, generalization, and training stability. The negative results underscore the difficulty of applying DRL to this specific fuzzing sub-problem with current methods. Future work should focus on several avenues. Investigating more sample-efficient DRL algorithms or model-based RL could be beneficial. For MAB-based approaches, systematic comparisons of different bandit algorithms (including non-stationary and contextual variants [36]) and reward

definitions are needed. Exploring hybrid approaches that combine simpler learning models with more expressive state features also holds promise. Ultimately, developing truly adaptive and intelligent seed schedulers may require novel learning frameworks tailored to the unique characteristics of the fuzzing process.

Declaration on Generative AI

We used generative AI tools (e.g., Gemini 2.5 Pro) to assist with language refinement and formatting tasks such as table arrangement and phrasing improvements. All technical content and results were developed by the authors.

References

[1] M. Zalewski, American fuzzy lop (afl) fuzzer, URl: https://lcamtuf. coredump. cx/afl (2017).

[2] A. Fioraldi, Fuzzing in the 2020s: novel approaches and solutions, Ph.D. thesis, Sorbonne Université, 2023.

[3] A. Abo-eleneen, A. Palliyali, C. Catal, The role of reinforcement learning in software testing, 2023. doi:10.1016/j.infsof.2023.107325.

[4] S. Li, X. Xie, Y. Lin, Y. Li, R. Feng, X. Li, W. Ge, J. S. Dong, Deep learning for coverage-guided fuzzing: How far are we?, IEEE Transactions on Dependable and Secure Computing (2022). doi:10.1109/TDSC.2022.3200525.

[5] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, S. Jana, Neuzz: Efficient fuzzing with neural program smoothing, in: Proceedings - IEEE Symposium on Security and Privacy, volume 2019-May, 2019. doi:10.1109/SP.2019.00052.

[6] Y. Wang, P. Jia, L. Liu, C. Huang, Z. Liu, A systematic review of fuzzing based on machine learning techniques, 2020. doi:10.1371/journal.pone.0237749.

[7] M. L. Puterman, Markov decision processes: Discrete stochastic dynamic programming, Wiley-Interscience, 2008. doi:10.1002/9780470316887.

[8] R. Sutton, A. Barto, Reinforcement learning: An introduction, IEEE Transactions on Neural Networks 9 (2005). doi:10.1109/tnn.1998.712192.

[9] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, M. Woo, The art, science, and engineering of fuzzing: A survey, IEEE Transactions on Software Engineering 47 (2021). doi:10.1109/TSE.2019.2946563.

[10] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, AFL++: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Association, 2020.

[11] D. She, A. Shah, S. Jana, Effective seed scheduling for fuzzing with graph centrality analysis, in: Proceedings - IEEE Symposium on Security and Privacy, volume 2022-May, 2022. doi:10.1109/SP46214.2022.9833761.

[12] M. Böhme, V. T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as markov chain, IEEE Transactions on Software Engineering 45 (2019). doi:10.1109/TSE.2017.2785841.

[13] M. Böhme, V. J. Manès, S. K. Cha, Boosting fuzzer efficiency: An information theoretic perspective, in: ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. doi:10.1145/3368089.3409748.

[14] Y. Chen, M. Ahmadi, R. M. Farkhani, B. Wang, L. Lu, Meuzz: Smart seed scheduling for hybrid fuzzing, in: RAID 2020 Proceedings - 23rd International Symposium on Research in Attacks, Intrusions and Defenses, 2020.

[15] J. Wang, C. Song, H. Yin, Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing, in: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, 2021. doi:10.14722/ndss.2021.24486.

[16] Q. Zhao, Multi-armed bandits: Theory and applications to online learning in networks, Springer Nature, 2022.
[17] S. Luo, A. Herrera, P. Quirk, M. Chase, D. C. Ranasinghe, S. S. Kanhere, Make out like a (multi-armed) bandit: Improving the odds of fuzzer seed scheduling with t-scheduler, in: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, 2024, pp. 1463–1479.
[18] W. R. Thompson, On the likelihood that one unknown probability exceeds another in view of the evidence of two samples, Biometrika 25 (1933). doi:10.2307/2332286.
[19] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multiarmed bandit problem, Machine learning 47 (2002) 235–256.
[20] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, X. Zhou, Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit, in: Proceedings of the 29th USENIX Security Symposium, 2020.
[21] Y. Su, D. Xiong, Y. Wan, C. Shi, Q. Zeng, Linfuzz: Program-sensitive seed scheduling greybox fuzzing based on linucb algorithm, IEEE Access (2024).
[22] A. Beygelzimer, J. Langford, L. Li, L. Reyzin, R. Schapire, Contextual bandit algorithms with supervised learning guarantees, in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 2011, pp. 19–26.
[23] C. Lyu, H. Liang, S. Ji, X. Zhang, B. Zhao, M. Han, Y. Li, Z. Wang, W. Wang, R. Beyah, Slime: Program-sensitive energy allocation for fuzzing, in: ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022. doi:10.1145/3533767.3534385.
[24] J.-Y. Audibert, R. Munos, C. Szepesvári, Exploration–exploitation tradeoff using variance estimates in multi-armed bandits, Theoretical Computer Science 410 (2009) 1876–1902.
[25] K. Böttinger, P. Godefroid, R. Singh, Deep reinforcement fuzzing, in: Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018, 2018. doi:10.1109/SPW.2018.00026.
[26] Z. Zhang, B. Cui, C. Chen, Reinforcement learning-based fuzzing technology, in: Advances in Intelligent Systems and Computing, volume 1195 AISC, 2021. doi:10.1007/978-3-030-50399-4_24.
[27] J. Shao, Y. Zhou, G. Liu, D. Zheng, Optimized mutation of grey-box fuzzing: A deep rl-based approach, in: Proceedings of 2023 IEEE 12th Data Driven Control and Learning Systems Conference, DDCLS 2023, 2023. doi:10.1109/DDCLS58216.2023.10166955.
[28] G. Choi, S. Jeon, J. Cho, J. Moon, A seed scheduling method with a reinforcement learning for a coverage guided fuzzing, IEEE Access 11 (2023). doi:10.1109/ACCESS.2022.3233875.
[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347 (2017).
[30] F. Pardo, A. Tavakoli, V. Levdik, P. Kormushev, Time limits in reinforcement learning, in: 35th International Conference on Machine Learning, ICML 2018, volume 9, 2018.
[31] S. Huang, S. Ontañón, A closer look at invalid action masking in policy gradient algorithms, in: Proceedings of the International Florida Artificial Intelligence Research Society Conference, FLAIRS, volume 35, 2022. doi:10.32473/flairs.v35i.130584.
[32] J. Schulman, P. Moritz, S. Levine, M. Jordan, P. Abbeel, High-dimensional continuous control using generalized advantage estimation, arXiv preprint arXiv:1506.02438 (2015).
[33] D. P. Kingma, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
[34] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-baselines3: Reliable reinforcement learning implementations, Journal of Machine Learning Research 22 (2021) 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.
[35] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, O. G. Younis, Gymnasium, 2023. URL: https://zenodo.org/record/8127025. doi:10.5281/zenodo.8127026.
[36] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, N. Abu-Ghazaleh, Syzvegas: Beating kernel fuzzing odds with reinforcement learning, in: Proceedings of the 30th USENIX Security Symposium, 2021.

Appendices

A. PPO Objective Functions

The Proximal Policy Optimization (PPO) algorithm [29] aims to maximize a clipped surrogate objective. The Conservative Policy Iteration (CPI) loss is:

$$L_t^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right], \tag{3}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. PPO uses a clipped version:

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]. \tag{4}$$

The full loss includes a value function term $L_t^{VF}(\phi) = (V_\phi(s_t) - V_t^{\text{targ}})^2$ and an entropy bonus $S[\pi_\theta](s_t)$:

$$L^{CLIP+VF+S}(\theta, \phi) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\phi) + c_2 S[\pi_\theta](s_t) \right]. \tag{5}$$

B. PPO Hyperparameters

Table 1
Hyperparameters used by PPO.

Hyperparameter	Value
Horizon (T)	{8, 16, 32}
Episodes	1, multi
Batch size	{2, 4, 8}
Learning Rate (α)	{0.003, 0.001, 0.0003}
Discount (γ)	0.99
GAE param. (λ)	0.95
Entropy coeff. (c_2)	0.0
Clip range (ϵ)	0.2

C. AFL Hit-Count Bins

Table 2
Hit-count bins used by AFL [1].

Bit	Counts
1	1
2	2
3	3
4	4-7
5	8-15
6	16-31
7	32-127
8	128+

D. Detailed DRL Process Overview

Figure 7 provides a detailed flowchart of the DRL seed scheduling process as integrated into AFL++, including the multi-queue mechanism and agent interaction points.

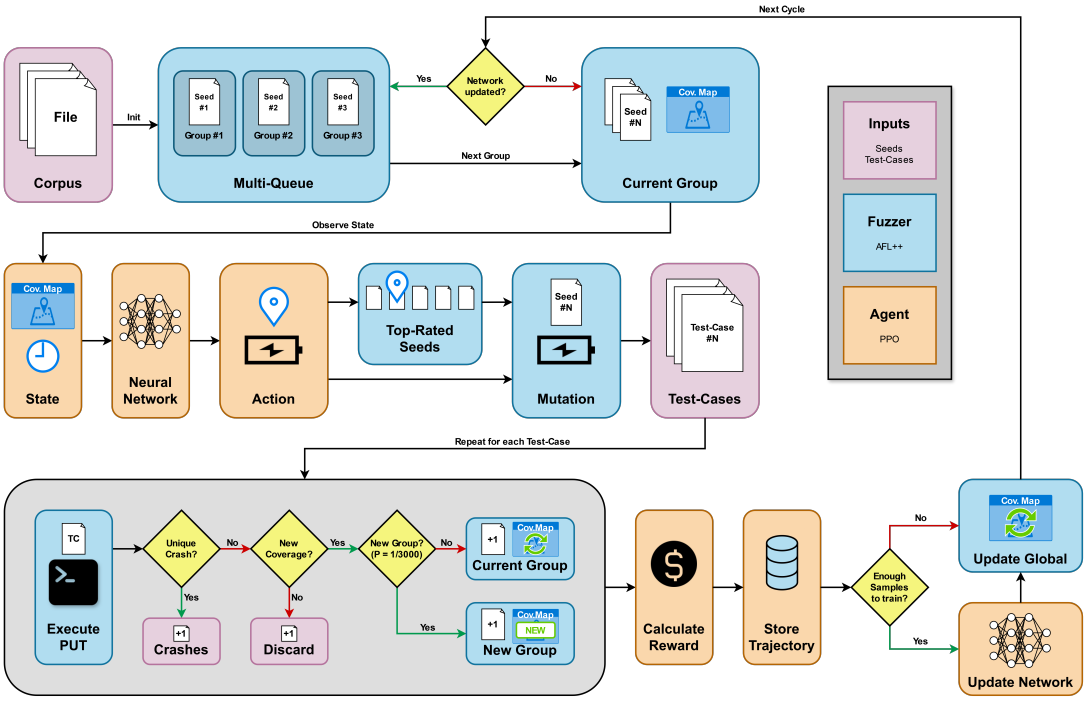


Figure 7: Overview of the DRL seed scheduling process, integrated into AFL++.

E. Multi-Queue Strategy for Episode Generation

To address the challenge of DRL training with typically long fuzzing campaigns, we employed a multi-queue strategy. This approach, illustrated in Figure 8, divides the fuzzing process into shorter, pseudo-independent episodes by managing separate "fuzzing groups," each with its own coverage context.

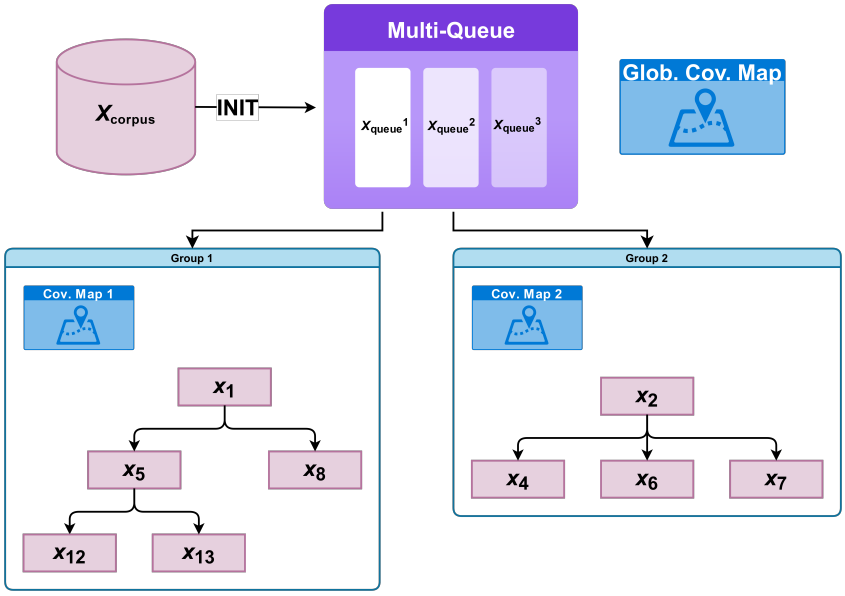


Figure 8: Schematic of the multi-queue strategy used for generating multiple training episodes. Each group maintains its own coverage context.