

II. METHODOLOGY

A. Overall Framework of SNIFFER

The SNIFFER continuously collects FIO and OCP metrics during runtime, which serve as the RL agent’s observations. The agent then selects a new FIO configuration as action, which is executed on the SSD, and the resulting performance metrics are used to calculate the reward. This loop enables the agent to learn a policy that generates latency spike-inducing command sequences (Figure 1). The key novelty of our approach lies in how the environment is defined and managed, especially under non-stationary and partially observable real-world conditions, where SSD behavior can vary due to internal factors invisible to the agent [12, 13].

B. RL Problem Formulation

We formulate the task as a Markov Decision Process defined by the tuple (S, A, R, P, γ) [14], where:

- S is the set of states, constructed from external metrics such as latency and write amplification factor (WAF).
- A is the set of actions, each corresponding to a unique I/O workload configuration.
- R is the reward function, designed to promote conditions that induce latency anomalies.
- P denotes the transition dynamics of the environment, which are unknown and non-stationary in real SSDs.
- γ is the discount factor that balances short-term and long-term objectives.

1) **State Definition:** The state $s_t \in S$ is constructed using a set of externally measurable performance indicators, ensuring a fully black-box approach that facilitates vendor-agnostic generalization. The state vector is defined as follows:

$$s_t = [\text{UserWrites}_t, \text{NANDWrites}_t, \text{WAF}_t, \text{Latency}_t]$$

The first three features— UserWrites_t , NANDWrites_t , and WAF_t —are extracted from the OCP telemetry specification. These metrics provide critical insights into the I/O workload behavior and internal write amplification (WA) trends, both of which are known precursors to GC and latency anomalies.

The final component, Latency_t , is derived from the FIO’s `clat_ns` output and captures statistical latency descriptors—including minimum, maximum, mean, and standard deviation—for write, read, and trim operations. This combination of throughput-related indicators and temporal latency statistics provides the agent with a holistic snapshot of device behavior, enabling it to reason about stress accumulation and emerging latency spikes.

Design Rationale: FIO-derived latency metrics provide real-time signals of transient performance degradation, while OCP metrics like WAF and NANDWrites capture longer-term stress build-up. This dual perspective enables the agent to correlate observable performance symptoms with their potential underlying causes. Furthermore, the exclusive use of standardized, externally visible metrics ensures that the SNIFFER is applicable across diverse SSD models without requiring internal firmware access.

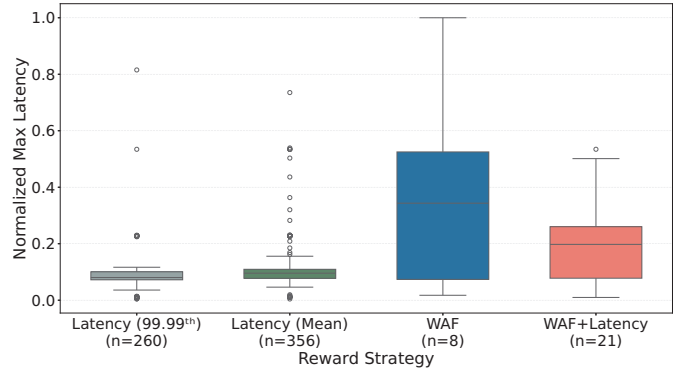


Fig. 2. Normalized maximum write latency by reward strategy. Sample sizes (n) are shown in labels. Despite a smaller sample size ($n=8$), WAF reached higher maximum latency compared to other reward strategies tested over hundreds of trials, highlighting its efficiency for anomaly induction.

2) **Action Definition:** Each action $a_t \in A$ corresponds to a pre-defined I/O workload configuration implemented via parameterized FIO scripts. The action space is structured around three I/O types (Write, Read, and Trim) and further diversified by varying key parameters such as I/O block size (Small, Medium, and Large) and the logical address range that discretized into eight segments representing hot and cold regions. To ensure a fair and stable learning process, the total I/O volume is normalized across all actions to equalize their execution duration. By equalizing action durations, we ensure the RL agent learns based on workload characteristics rather than timing artifacts. Total parallelism (`iodepth` \times `numjobs`) is fixed at 256 to ensure steady stress levels across actions.

Design Rationale: The overall action space is designed to balance expressiveness with tractability. By structuring the space with well-established I/O semantics and locality principles, the agent can efficiently explore diverse workload behaviors. The normalization of workload duration is particularly critical, as it ensures temporal consistency and fair reward attribution, which are essential for stable policy optimization.

3) **Reward Definition:** We define the reward function based on the WAF, a critical metric reflecting the ratio of NAND writes to host writes, which indicates the level of internal device activity [15].

$$r_t = \begin{cases} \alpha \cdot (\text{WAF}_t - 1), & \text{if } \text{WAF}_t > 1 \\ \beta \cdot (\text{WAF}_t - 1), & \text{otherwise} \end{cases}$$

This formulation incentivizes the agent to explore I/O patterns that increase WAF. The coefficient α amplifies the reward when WAF exceeds 1, reflecting scenarios where internal WA may contribute to long-latency behaviors. Conversely, β is used when WAF is below or equal to 1, potentially penalizing low amplification scenarios that are less likely to stress the device. By tuning α and β , the agent can be guided toward I/O patterns that increase internal stress and induce performance anomalies.

Design Rationale: This design choice was empirically determined by reward shaping experiments (Figure 2) where direct latency-based rewards (mean or 99.99th percentile latency) resulted in unstable training. On high-performance SSDs, latency spikes are rare, leading to sparse and inconsistent reward signals that prevented effective learning. In other cases, the agent prematurely converged on policies that exploited only moderate delays. Similarly, the combination of latency and WAF reward, which provided threshold-based reinforcement for high write latency in addition to WAF, also proved unstable due to the inherently bursty and rare nature of high-latency events. In contrast, using WAF alone offered a more stable and informative feedback signal for anomaly induction than latency-based rewards. It captures internal stress conditions that often precede latency anomalies and provides consistent measurability across SSDs of varying capabilities, enabling more stable training dynamics.

Statistical analysis further confirmed this finding: a one-way ANOVA revealed a significant effect of reward design on performance ($F = 37.88$, $p = 3.86 \times 10^{-20}$), and post-hoc Tukey HSD tests showed that WAF yielded significantly higher performance than all other reward strategies (all adjusted $p < 0.001$).

4) **Algorithm Selection:** We employ the on-policy RL algorithm Proximal Policy Optimization (PPO) [16] as the agent-training component in SNIFFER, selected for its robustness in noisy real-device environments and its compatibility with discrete action spaces. PPO optimizes a clipped surrogate objective to stabilize policy updates, improving training resilience in noisy environments [16]. Its minimal sensitivity to hyperparameters make it a practical choice for real device training where experimental cost is high and reset overhead is non-trivial.

Selection Rationale: We compared our PPO based framework with other representative methods, including RL algorithms designed for discrete action spaces such as Deep Q-Networks (DQN) [17] and Advantage Actor-Critic (A2C) [18], as well as a traditional optimization method, GA. All algorithms were evaluated under comparable conditions for the latency spike inducing task (Figure 3).

GA generally operates effectively in environments where training can proceed over full episodes and stable state transitions are guaranteed. However, in our real device setup, a single episode requires several hours, making conventional GA training infeasible. To ensure a fair comparison, we adjusted GA’s parameters such that the total number of exploration steps approximated the $n_steps = 64$ used in PPO training. Specifically, we set the chromosome count to 4, population size to 3, and number of generations to 5, resulting in 60 exploration steps followed by 4 execution steps.

DQN is known to be less effective in large discrete action spaces [19], which explains its poor performance under our 241 action setup. GA and A2C achieved comparable performance, with maximum latencies in the 460 ms range. Although A2C employed similar hyperparameters to PPO, it suffered from premature convergence, limiting sufficient

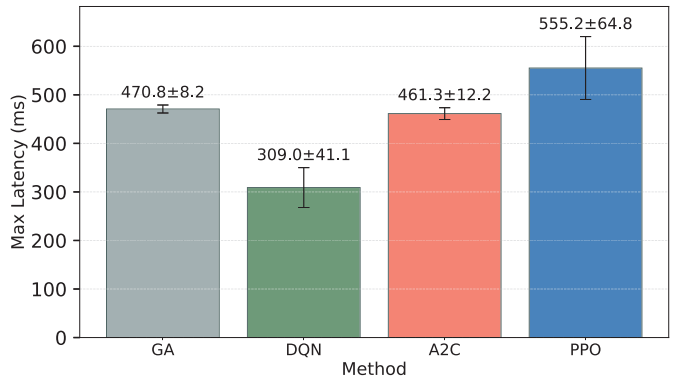


Fig. 3. Comparison of maximum latency induced by different algorithms (GA, DQN, A2C, PPO). Shown is the mean \pm s.d. of 5 runs with different random seeds.

exploration. In contrast, PPO consistently outperformed the other methods, achieving the highest average and maximum latency. To verify the statistical significance of differences among algorithms, we conducted a one-way ANOVA. The results indicated that algorithm choice had a significant effect on performance ($F = 34.45$, $p = 3.26 \times 10^{-7}$). Subsequent post-hoc Tukey HSD tests confirmed that PPO significantly outperformed A2C ($p = 0.0077$), DQN ($p < 0.001$), and GA ($p = 0.0167$).

Based on these results, we confirm that PPO is the most effective algorithm for inducing SSD latency spikes, and we adopt it as the primary agent-training method in all subsequent experiments.

C. Training and Inference Strategy

Training is conducted entirely online—directly interacting with real SSDs—without relying on offline datasets or simulated feedback. This allows that the agent learns from actual device behavior, capturing realistic latency dynamics and stress patterns. Each episode runs up to a fixed horizon of 20,000 steps, or is terminated early if a pre-defined latency threshold is exceeded. After training, we evaluate the learned policy by performing inference runs on the same devices used during training. These inference runs are conducted using a stochastic policy to assess the robustness and diversity of the generated TCs. The key objective is to determine whether the learned policy can induce long-latency conditions more efficiently—e.g., with fewer steps—than during training. In many cases, the stochastic nature of the policy not only preserves but even enhances stress-inducing behavior, occasionally triggering higher latency spikes or discovering alternative stress paths. This indicates that the policy has captured transferable and meaningful patterns, rather than merely memorizing specific sequences.

D. Experimental Setup

To reflect practical validation workflows, we decoupled the RL agent training and SSD evaluation environments. The test hosts ran CentOS 9 with kernel version 5.9.1. To minimize

TABLE I
COMPARISON OF MAX LATENCY, STEPS, EXECUTION TIME, AND ENTROPY BETWEEN SNIFFER AND RANDOM BASELINE ACROSS SSD PRODUCTS

Product	Max Latency (ms)	Δ Latency (%)	Steps	Δ Steps (%)	Time (hour)	Δ Time (%)	Entropy (nats)	Δ Entropy (%)
	Ours vs. Random	(%)	Ours vs. Random	(%)	Ours vs. Random	(%)	Ours vs. Random	(%)
A	656.2 vs. 375.5	+74.7%	2,448 vs. 3,301	-25.8%	0.6 vs. 3.3	-81.8%	2.43 vs. 5.38	-54.8%
B	302.7 vs. 210.7	+43.7%	1,896 vs. 9,851	-80.7%	1.6 vs. 10.5	-84.8%	4.04 vs. 5.47	-26.2%
C	430.6 vs. 382.6	+12.5%	1,665 vs. 10,973	-84.8%	2.4 vs. 15.5	-84.5%	0.71 vs. 5.48	-87.1%
D	79.6 vs. 62.4	+27.6%	12,302 vs. 18,061	-31.9%	12.7 vs. 14.0	-9.3%	4.25 vs. 5.48	-22.3%

thermal variability, we attached an active cooling fan to each SSD and maintained the ambient temperature of 25°C. This setup ensured consistent performance and prevented TT during repeated training and inference episodes. Moreover, we conducted all evaluations on a real PCIe Gen5 NVMe SSD using FIO 3.36, training the RL agent online through direct interaction with the device.

Target Devices. We evaluated SNIFFER on four commercial SSD products, each from a different vendor, denoted as A, B, C, and D. To evaluate them fairly under consistent experimental conditions, where each of them has different NAND characteristics and firmware architecture, all tests were conducted in a temperature-controlled environment. Furthermore, to ensure hardware isolation and avoid resource contention across devices, we assigned a dedicated experimental host to each SSD.

RL Configuration and Workload. We trained the agent using the PPO algorithm from scratch, maintaining a uniform training configuration across products. The training horizon was set to 20,000 steps per device, with key hyperparameters including Adam optimizer (learning rate: 3×10^{-4}), clipping range $\epsilon = 0.2$, rollout length $n_steps = 64$, discount factor $\gamma = 0.99$, GAE parameter $\lambda = 0.95$, and a 1024-1024-1024 policy network. Other settings follow Stable Baselines3 defaults [20].

To evaluate the performance of the SNIFFER in a vendor-agnostic and reproducible manner, we used a random policy as the baseline. The random policy samples uniformly from our carefully designed FIO action space, which includes diverse yet bounded configurations across I/O type, block size, and address range. This design choice is motivated by two key factors: (1) heuristic or rule-based baselines often depend on proprietary knowledge, which violates the black-box assumption and varies widely across vendors; and (2) manually optimized FIO scripts are difficult to standardize and reproduce, making fair comparisons non-trivial. In contrast, our random policy provides a consistent and unbiased reference point within the same operational constraints as the RL agent.

Importantly, our contribution lies not in manually designing effective FIO workloads, but in enabling the agent to *automatically discover high-stress sequences* through trial-and-error. This approach eliminates the need for SSD-specific expertise, significantly reducing the engineering effort required to uncover edge-case behaviors, while also ensuring that learned TCs are adaptable across different products.

III. EXPERIMENTAL RESULTS

A. Latency Spike Induction Results

To assess the capability of SNIFFER to induce latency spikes, we conducted comparative experiments against a random baseline using the previously defined SSD products (A–D). The comparison focuses on two key metrics: the maximum latency observed and the number of steps required to reach it.

As summarized in Table I, the SNIFFER consistently induced a higher maximum latency than random exploration. For example, on product A, the SNIFFER achieved a maximum latency of **656.2 ms**, which is **74.7%** higher than that observed under random behavior (375.5 ms). Additionally, SNIFFER required substantially fewer steps to induce latency spikes, with reduction rates ranging from 25.8% to over 84%. Notably, on product C, the step count dropped from 10,973 to 1,665, an **84.8% reduction**, and the action entropy decreased from 5.48 to 0.71, a **87.1% drop**. This reduction in both steps and exploration uncertainty suggests that SNIFFER learns a more focused and deterministic policy, enabling rapid convergence toward high-latency states.

To complement the step count metric, we also measured the actual wall-clock execution time per episode. On product B, SNIFFER required only **1.6 hours** to reach the target latency, compared to **10.5 hours** under the random baseline, achieving a **84.8% reduction** in total validation time. While the learned policy consistently reduced the number of steps required to reach the target latency across products, the corresponding reduction in wall-clock time varied. For example, on Product D, the policy achieved a **31.9%** step reduction (from 18,061 to 12,302), yet the wall-clock execution time was reduced by only **9.3%**. This highlights that step efficiency does not always translate directly to time efficiency, particularly when I/O operations differ in execution cost across states.

These results demonstrate that SNIFFER is capable of inducing more severe latency behaviors while requiring fewer steps to discover them, suggesting effective navigation of the input space to trigger worst-case scenarios, which are critical for rigorous SSD validation. The significantly lower entropy across products (0.71–4.25 vs. 5.47 under random policy) further confirms that the behavior is not due to chance, but the result of a trained and optimized policy.

To further investigate how the SNIFFER differs from the random baseline, we visualized the step-by-step action selections and corresponding write latency trends on Product B, as shown in Figure 4. In the random baseline cases (Figures 4a

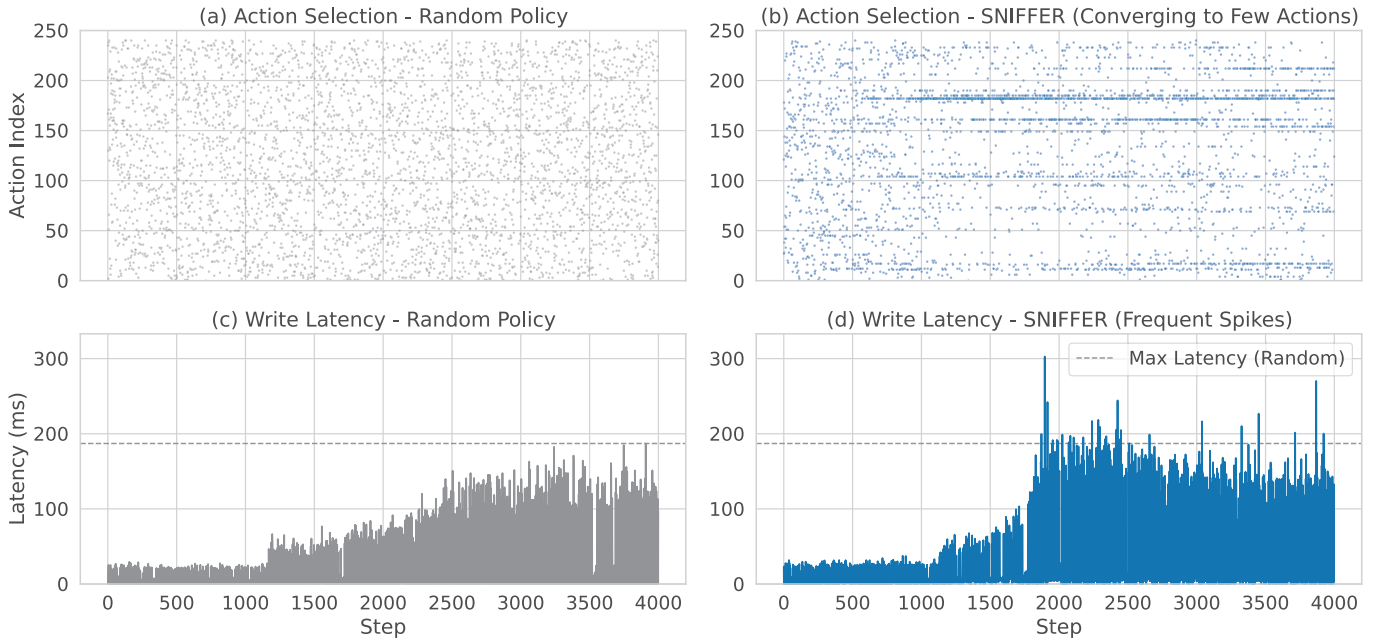


Fig. 4. **Step-wise Visualization of Action Selection and Write Latency Trends on Product B.** Each dot represents the action taken at a given step, and the line shows the corresponding write latency in milliseconds. Subfigures (a) and (c) illustrate the random baseline (gray), where actions are selected without strategy and latency spikes appear infrequently and without clear patterns. In contrast, subfigure (b) shows that SNIFFER (blue) gradually converges to a small subset of effective actions, and subfigure (d) demonstrates that this policy induces multiple latency spikes that consistently exceed those observed under the random baseline.

and 4c), actions are selected without any discernible strategy, resulting in widely scattered patterns and latency spikes that appear late and inconsistently. This behavior highlights the inherent inefficiency of unguided test generation. By contrast, the SNIFFER (Figure 4b) exhibits a noticeable shift toward more focused action selection. After an initial exploration phase, the agent tends to favor a narrower set of effective I/O patterns, though occasional variations still appear. This reflects partial convergence and a more guided search compared to the random baseline. As a result, it causes multiple long-latency events earlier and more reliably, as shown in Figure 4d, where spike magnitudes exceed those observed under random policies. This behavior confirms that the agent learns to focus on high-impact test sequences that stress the SSD more aggressively and consistently.

This behavioral shift confirms that the learned policy not only improves spike induction performance numerically but also operates in a deterministic and targeted manner, focusing on effective high-stress I/O patterns. The visualization strengthens our claim that the RL agent identifies meaningful patterns correlated with worst-case firmware behavior, making it a practical and interpretable tool for real-world SSD validation.

B. Diverse and Accelerated Spike Induction

To evaluate the practical effectiveness of our learned policy, we analyzed the inference behavior on Product A by comparing latency spike characteristics with those observed during the training phase. In particular, we focused on two key aspects:

(1) whether the policy could induce a latency spike earlier than in training, and (2) whether the maximum latency increased.

The reproduction of specific TCs on SSDs is inherently challenging due to the non-deterministic and opaque nature of internal behaviors such as GC, WL, and background maintenance. These processes are governed by history-dependent triggers that are not externally observable or controllable, making it difficult to consistently recreate identical outcomes—even under fixed I/O workloads. Among these, latency spikes are particularly critical yet notoriously hard to reproduce, as they often emerge from complex and timing-sensitive interactions between internal mechanisms and external stimuli.

We define a latency spike as any write latency exceeding 500 ms. This threshold is not arbitrary; it is derived from the empirical distribution of write latencies observed during the training phase, where the agent was actively learning through interaction with the device. The 99.99th percentile latency reached approximately 491 ms—just 8.75 ms below our 500 ms cutoff—indicating the rarity and severity of such spikes even under learned policies.

Importantly, this threshold is stricter than those typically used in production environments, as our training data already exhibits higher stress than conventional workloads. Under this challenging baseline, our learned policy successfully caused a latency spike of 501 ms at step 8,693, demonstrating its ability not only to intensify worst-case behavior but also to accelerate its occurrence.

Despite these challenges, the SNIFFER demonstrates strong reproducibility and stress-inducing capability. As shown

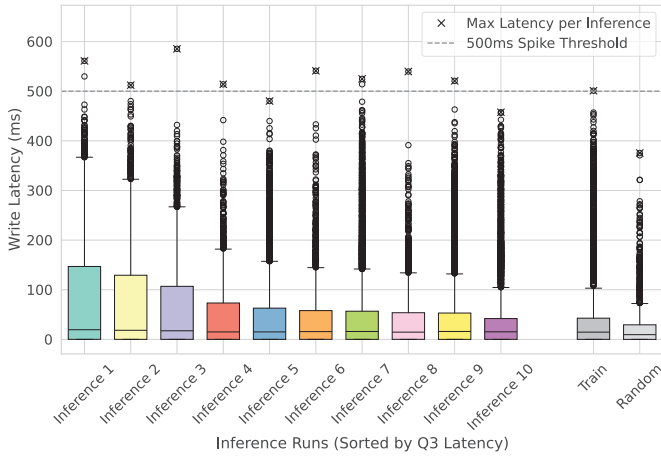


Fig. 5. Write latency distributions from 10 inference runs, sorted by Q3 (75th percentile). All inference runs exceed the Train Q3, and 8 out of 10 exceed its maximum latency.

in Figure 5, all 10 inference runs produce 75th percentile (Q3) write latencies that exceed the training distribution, indicating a consistent elevation in baseline latency behavior. Notably, the maximum latency (excluding outliers) follows the same order as Q3, suggesting that the entire latency distribution has uniformly shifted toward more stressful conditions. Furthermore, in 8 out of 10 runs, the maximum latency surpasses that of the training phase, confirming that the learned policy not only reproduces but also amplifies worst-case latency scenarios. These findings validate the effectiveness of our approach in consistently generating long-latency TCs on real hardware without relying on internal device instrumentation.

To assess whether SNIFFER could induce latency spikes earlier than during training, we measured the first occurrence of a write latency exceeding 500 ms for each inference run. As illustrated in Figure 6, most policies triggered a spike before the training baseline (dashed line), demonstrating an improvement in stress induction timing. However, Inference 3 and Inference 7 reached the 500 ms threshold later than the training run. Despite this delay, both runs exhibited higher maximum latencies, as shown in Figure 5, suggesting that the learned policies can intensify stress even when not accelerating its onset. Notably, Inference 5, Inference 10, and the random baseline failed to reach the 500 ms threshold within the allowed steps, further underscoring the reliability of the trained policies in causing severe latency behaviors.

Such consistent performance under randomness highlights the generalization capability of the learned policy. It reliably causes worst-case behaviors without overfitting to specific I/O patterns observed during training. Moreover, its stochastic nature enables the generation of diverse, high-impact test sequences, broadening coverage across corner cases and significantly enhancing the effectiveness of validation efforts. These properties make the SNIFFER approach both efficient and practical for uncovering elusive performance anomalies in real SSD products.

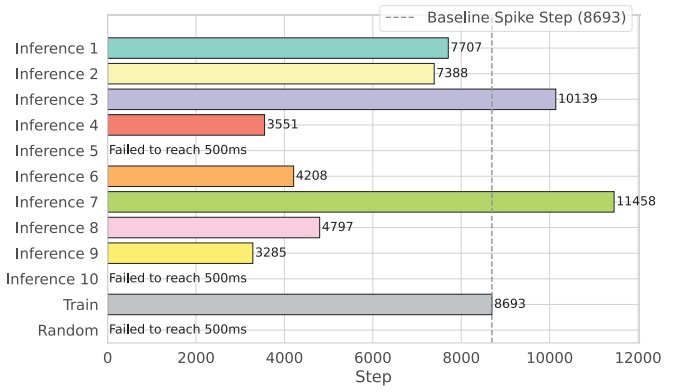


Fig. 6. Step-wise comparison of the first 500 ms latency spike across inference runs. Bars are sorted identically to Figure 5, with the dashed line indicating the Train spike step.

IV. DISCUSSION AND CONCLUSION

This work tackles the challenge of automated TC generation for SSD validation in a realistic vendor-agnostic, black-box, non-stationary setting.

Why On-Policy RL Works: Our results strongly suggest that the sequential and non-stationary nature of the SSD environment is a critical factor. Unlike GA, which performs stateless optimization, RL methods leverage the history of interactions. Among RL algorithms, the on-policy nature of PPO proved decisive. By exclusively using the most recent interactions for policy updates, it avoids being misled by stale data from the SSD’s past internal states, a pitfall that degrades the performance of off-policy methods.

Engineering Utility: SNIFFER offers significant practical value for validation engineers. The learned policies are reusable TCs that can be integrated into regression testing to quickly validate firmware patches. By automatically discovering complex, high-stress I/O sequences, it reduces substantial manual effort and uncovers corner-case behaviors that rule-based tests might miss.

Conclusion: We introduced SNIFFER, an RL-based framework that automates the generation of latency spike-inducing TCs on real SSDs without internal access. By relying exclusively on external metrics from FIO and OCP, our method ensures confidentiality and broad applicability. Experiments confirmed that our approach consistently achieves latency spikes efficiently and with high repeatability across multiple commercial SSD models. While the trained policies are inherently specific to each device, our framework is reproducible and portable across diverse SSDs, enabling broad applicability across vendors. Additionally, the high reproducibility and reduced inference steps of RL-generated TCs substantially accelerate fault diagnosis, enabling engineers to swiftly identify root causes and implement corrective actions. This work makes a key contribution to industrial SSD validation by providing a scalable RL methodology that works across vendors while preserving confidentiality, thereby setting a robust engineering baseline for future automated firmware validation frameworks.

REFERENCES

- [1] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and {SSD} deployments," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 263–276.
- [2] J. Kim, K. Choi, W. Lee, and J. Kim, "Performance modeling and practical use cases for black-box ssds," *ACM Trans. Storage*, vol. 17, no. 2, Jun. 2021.
- [3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
- [4] D. Skourtis, D. Achlioptas, C. Maltzahn, and S. Brandt, "High performance & low latency in solid-state drives through redundancy," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013, pp. 1–9.
- [5] Y. H. Lee, Y. Oh, G. Jeong, M. Pi, H. Kwon, H. Lim, E. Kim, S. Lee, and B. Hwang, "Graft: Graph-assisted reinforcement learning for automated ssd firmware testing," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.
- [6] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 469–481.
- [7] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "{LinnOS}: Predictability on unpredictable flash storage with a light neural network," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 173–190.
- [8] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [9] S. Shirode, "Identifying latency outliers in workload testing," <https://www.micron.com/about/blog/storage/insights/identifying-latency-outliers-in-workload-testing>, August 2023, micron Technology Blog.
- [10] J. Axboe, "fio: Flexible i/o tester, version 3.36," <https://github.com/axboe/fio>, 2023, accessed: 2025-09-12.
- [11] "Ocp 101: What is ocp?" Open Compute Project Foundation, Tech. Rep., July 2018, accessed: 2025-09-12. [Online]. Available: <https://www.opencompute.org/files/OCP-Basics-July-2018.pdf>
- [12] Q. Liu, S. Liu, Y. Wu, S. Zou, L. Koster, P. Cui, R. Cai, and X. Liu, "When is partially observable reinforcement learning not scary?" in *International Conference on Machine Learning (ICML)*, 2022, pp. 12 949–12 970.
- [13] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.
- [14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.
- [15] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: Association for Computing Machinery, 2009.
- [16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
- [17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*, pp. 1928–1937, 2016.
- [19] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," *arXiv preprint arXiv:1512.07679*, 2015.
- [20] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of machine learning research*, vol. 22, no. 268, pp. 1–8, 2021.