When Legal Completion Reordering Breaks QoS: Risk Cliffs in NVMe Scheduling

Tihomir Thomas Bicanic
University of Trier
Trier, Rheinland-Pfalz, Germany
bicanic@uni-trier.de

Abstract

Modern NVMe stacks permit legal completion reordering, yet the QoS impact of this within-spec scheduling freedom is rarely evaluated systematically. We present NVMe-lite, a deterministic model of an NVMe-like host–device protocol that isolates completion scheduling from protocol state to enable reproducible exploration. We treat the schedule as a second input and parameterize reordering freedom with a bound k, sweeping k across FIFO, Random, Batched, and Adversarial policies, and considering fault modes (NONE, TIMEOUT, RESET). Our results show risk cliffs, beyond policy-specific thresholds, small increases in k trigger abrupt transitions into a high-tail state, while other policies remain stable. We quantify realized reordering via a normalized inversion metric (RD) and report p95 latency in scheduler steps. Finally, we extract top-K poison schedules as deterministic traces for replay and regression testing, and validate a subset differentially against an independent C implementation. NVMe-lite supports systematic QoS robustness validation of compliant host–device stacks.

1 Introduction

Operating Systems often interact with fast I/O devices via *asynchronous host-device protocols* with queue-based data structures in shared memory. With NVMe, the host provides commands in *Submission Queues* and the controller returns completions as entries in *Completion Queues*. The visible completion order need not be the same as the submission order, because commands may complete *out of order* [14]. A structurally similar pattern is found in VirtIO, where descriptor indices are exchanged between driver and device via *Available*- and *Used*-rings [16]. Such queue protocols enable high parallelism, but shift complexity into the *schedule*. The runtime behavior arises from interleavings of host actions, device progress, notifications, and recovery events.

This schedule variability is not merely noise around a mean, but often affects *tail latency*. NVMe, for example, allows *interrupt coalescing*, i.e., interrupts can be bundled or delayed to reduce host overhead, which directly influences latency distributions [14]. In large and parallelized systems, tail effects are particularly critical. Rare delays of individual components can dominate the end-to-end tail (Tail at Scale) [5]. Common evaluations, however, often treat reordering and schedule effects only implicitly (e.g. as random variation), instead of systematically investigating *reordering freedom as a controllable parameter*.

This challenge is exacerbated by the necessity to also handle *error and recovery paths* robustly. In practice, I/O timeouts exist. Under Linux, for example, the boot parameter `nvme_core.io_timeout` can be used (documentation states a default of 30 s) [1]. Additionally, controller resets are part of the realistic operating picture of NVMe-based systems [15]. Precisely such paths are difficult to test, because they occur rarely, but in the event of an error must function correctly and performantly.

This work addresses the gap between protocol-close complexity and reproducible, systematic schedule evaluation. Real NVMe stacks are often too variant-rich for the targeted investigation of large schedule spaces (firmware/hardware differences, instrumentation effort), so we use a lightweight but protocol-close model *NVMe-lite*[1]. We investigate the device side along two axes: First *reordering freedom* via a bound k (*bounded reordering*) and second *schedule policies* (FIFO, Random, Adversarial, Batched). Additionally, we consider fault modes (None, Reset and Timeout). To make the search space manageable, we follow the bounding idea from systematic interleaving exploration (e.g., CHESS) [13]. The focus is on a stress setting with high parallelism, in which robust and fragile policies clearly separate. Our results show *risk cliffs*: with increasing reordering freedom, certain policies exhibit transitions into a high-tail regime, while other policies remain stable. Beyond aggregates, we provide curated *poison schedules* as reproducible traces that are directly suitable for deterministic replay and regression testing in driver/runtime workflows.

Contributions.

- A protocol-close, reproducible *NVMe-lite* model for analyzing schedule-induced tail latency and recovery robustness.

[1]Source code, experiment harness, and plotting in https://github.com/TheBuccaneer/fgt_nvme-robustness

- A systematic robustness analysis with *bounded reordering* as a controllable parameter and multiple schedule policies.
- A *risk-cliff* presentation as well as reproducible *poison* traces for practical diagnosis and replay.

2 Related Work

Tail latency and predictability in NVMe/Flash.: Several works address tail latency by changing mechanisms in the host, firmware, or interface. [17] extend NVMe with semantic hints (latency-sensitive vs. not) to avoid catastrophic outliers under interrupt coalescing. [9] combine determinism and co-design to provide predictability contracts, e.g., via deliberate failing and reconstructive paths. [10] reduce extreme read tails despite writes through hardware-close techniques (RAIL/Hot-Cold separation). In contrast, we do not propose a new optimization, but a *systematic robustness evaluation*: we treat completion reordering freedom (bound k, policy) as controllable parameters, measure realized reordering (*RD*), and map *risk cliffs* as transitions into a high-tail regime, producing reproducible *poison schedules* as regression artifacts.

Protocol-close emulation: NVMeVirt [8] provides a kernel-based virtual NVMe device for rapid prototyping and evaluation of NVMe features. Our *NVMe-lite* is lightweight and deterministic. It preserves key protocol mechanisms (queues, completions, notifications) while abstracting enough to enable traceable schedule exploration and deterministic replay.

Exploration and fuzzing of schedules. CHESS shows that bounded exploration of thread interleavings (preemption bounding) can make rare concurrency bugs reproducible [13]. We transfer this mindset from thread schedules to host–device *completion schedules*. Unlike CHESS, which varies CPU thread orderings, we vary legal NVMe completion reorderings (via k/*RD*) to expose QoS degradation regimes rather than memory-safety bugs. Complementary driver-testing work explores rare executions via simulation and model-guided fuzzing, e.g., PrIntFuzz [11], DevFuzz [18], and VIRTFUZZ [6]. We align with the goal of targeting rare behaviors, but focus on QoS robustness under completion scheduling (including fault/recovery), and contribute poison schedules plus risk-cliff maps for diagnosis and regression testing.

3 Methodology

We implement a lightweight, event-driven model of a host-device I/O protocol (NVMe-lite) that preserves central NVMe mechanisms: submission and completion queues as ring buffers, command identifiers (CID), as well as the phase tag to distinguish new completion entries across queue wrap-around. As with NVMe, completions are asynchronous and may (legally) occur *out-of-order* [14]. To enable systematic studies, we strictly separate protocol state (queues, pointers, phase tags, pending set) and scheduler logic (selection of the next completion/fault).

We treat the *schedule* as a second input dimension and a key enabler for reproducibility. A *schedule* is a sequence of discrete scheduler steps (*event ticks*) that trigger device events, primarily `COMPLETE` for a pending command, optionally augmented by fault events (e.g., `RESET/TIMEOUT`). Each run is uniquely labeled by the tuple (seed_id, schedule_seed, policy, bound_k, fault_mode, scheduler_version) and is therefore 1:1 reproducible. We measure latencies in *scheduler steps*, i.e., as the difference in ticks between `SUBMIT` and `COMPLETE`. Ticks index the globally serialized event trace (including `SUBMIT` and `COMPLETE`). Nondeterminism arises exclusively from the scheduler's choice of the next *device-side* event.

A central parameter is `bound_k`, which limits the reordering freedom of the device scheduler. In each tick, there exists a set P of pending commands, which we put into a *canonical order* (ascending by `cmd_id`, so that a stable order is guaranteed for equal submit ticks). A scheduler with bound k may, in the next tick, choose only an element from a 0-based window of the canonical list:

$$\text{choose index } i \in \{0, \ldots, \min(k, |P| - 1)\}.$$

Thus, $k = 0$ enforces strict FIFO (only the head element is selectable), and $k = inf$ allows choosing any pending command. Note that $k = 0$ eliminates *reordering* (RD= 0), but policies may still differ in *timing* because cheduler also controls internal device progress steps. Hence small p95 differences at $k = 0$ are possible even without reordering. The motivation follows the bounding idea from systematic concurrency exploration: nondeterminism is parameterized as a *scalable search space* [13].

Schedule policies: We consider four policies that cover orthogonal operating modes. (1) FIFO always selects the head ($i = 0$). (2) Random selects uniformly at random from the k-window, driven by `schedule_seed`. (3) Batched models coalescing by bundling up to n completions per tick (default: n=4). (4) Adversarial approximates a worst-case firmware scheduling. Within the permitted window, it prefers a *late* element (e.g., maximum index) in order to keep older commands pending as long as possible and thereby amplify backlog and tail latency. This policy is intentionally not a security threat model, but a robustness stress test within legal protocol degrees of freedom.

Adversarial robustness setting. In this *schedule threat model*, we model a worst-case *legal* device scheduler. The device may choose any completion order permitted by the bounded reordering parameter k (e. g., due to firmware heuristics, coalescing, or queue management), but we do *not* assume data

corruption, DMA attacks, or malicious host behavior. Thus, adversarial refers to stress-testing QoS robustness under worst-case *within-spec* scheduling freedom. This distinction is critical. Our goal is not security analysis but robustness validation of compliant implementations.

Fault- and recovery model. To stress error handling as a first-class aspect of schedule exploration, we evaluate fault paths in addition to `fault_mode=NONE` using `TIMEOUT` and `RESET`. Faults are modeled as explicit scheduler events (deterministically derived from `schedule_seed` and `fault_mode` or serializable as `INJECT_FAULT` at a fixed tick position), to reproducibly stress recovery paths. `TIMEOUT` stands for permanently missing completions of a command. `RESET` resets controller state and can discard pending work. We anchor performance/tail statements primarily in `fault_mode=NONE`. Fault modes serve the robustness analysis of error handling and cleanup.

Experiment matrix and stress setting. Our main setting is `submit_window=inf` (SWinf)[2] as a high-concurrency setting with maximal overlap. We sweep $k \in \{0, 1, 2, 3, 5, 10, inf\}$[3] across all policies, typically with `schedule_seed` drawn from a fixed pool (e.g., 0-99) per cell. Smaller submit windows (SW2/SW4) serve as a sensitivity study. Reduced parallelism limits the size of the pending set and thus the effective reordering freedom, so we expect the realized degree of reordering *RD* to decrease (fewer opportunities for inversions) and tail cliffs to be attenuated (less schedule sensitivity).

Tracing and evaluation pipeline. Each run emits a chronological log (header + events such as `SUBMIT/COMPLETE/FENCE/RESET/RUN_END` and state snapshots). An offline pipeline parses logs into a run-level CSV and aggregates these into a summary CSV. Plots and tables are generated from these CSVs.

Metrics: RD, tail, and risk cliffs. **Reordering Degree (RD)** measures whether the k knob *has an effect*. We compute the normalized inversion count between submit and completion order. For n completed commands, let inv be the number of pairs (i, j) that are $i < j$ in submit order, but $i > j$ in completion order. Then

$$\mathrm{RD} = \frac{2 \cdot \mathrm{inv}}{n(n-1)} \in [0, 1],$$

and RD is thus Kendall-τ-like (without tie corrections in the simplest case) [7]. We report tail latency as p95 in scheduler steps. We operationalize risk cliffs as abrupt p95 increases when incrementing k: a risk cliff occurs if p95 between two successive k values rises by more than 20%. At the cell level (Policy×k×Fault) we primarily aggregate p95 means.

Poison schedules and differential validation. For practical usability, we extract *poison schedules* (top-K=10 runs with the largest p95 across all Policy×k×Fault combinations) and export condensed traces that can be deterministically re-played. To reduce single-implementation artifacts, we validate selected schedules differentially: identical seeds and schedules are executed both on the Rust reference implementation and on an independent C DUT, and normalized event sequences are compared via diff-based comparison [3].

4 Evaluation

This section presents a combined *results and discussion* of our evaluation. We study how bounded completion reordering affects QoS robustness in NVMe-like host-device protocols. All experiments use **SWinf** as the primary stress setting unless stated otherwise.

We organize the evaluation around four research questions: (RQ1) how k controls realized reordering (RD), (RQ2) how tail latency evolves and when risk cliffs occur, (RQ3) whether worst-case behavior can be captured as reproducible poison schedules, and (RQ4) how robust these findings are across seeds, fault modes, and differential validation.[4]

4.1 Caveat on Fault Modes

All QoS results (RD, p95, and risk-cliffs) are anchored to `fault_mode=NONE`. Fault modes (`TIMEOUT`, `RESET`) are evaluated separately to assess recovery robustness. Under faults, run termination and resets can truncate or reshape the latency distribution, so p95 values are reported for transparency but are not directly comparable to the `NONE` baseline.

4.2 RQ1: Realized Reordering (RD) and Validation of the k-Bound

Increasing the reordering bound k should monotonically increase the*realized reordering* (RD), while different scheduling policies should realize this freedom to different extents.

Figure 1 reports *mean* realized reordering (RD) across 100 schedule seeds per cell (`fault_mode=NONE`, SWinf). At $k = 0$, RD is zero for all policies, confirming strict FIFO behavior. As k increases, mean RD grows monotonically but remains quantitatively small because most runs realize only limited reordering on average. At $k = 3$, FIFO remains at RD= 0.000, Batched reaches RD≈ 0.027, Random RD≈ 0.061, and Adversarial RD≈ 0.125. At $k = inf$, mean RD saturates at

[2]Here, inf denotes a very large submit window that is effectively unbounded for our workloads. It is of course not mathematically infinite, but chosen large enough such that the host does not become the limiting factor.

[3]As above, inf indicates an effectively unbounded bound k, implemented as a sufficiently large value so that any pending command is selectable.

[4]Beyond uniform schedule-seed sampling, we also evaluated an RD-guided sampling variant (RDSS); results and scripts are provided in the artifact repository.

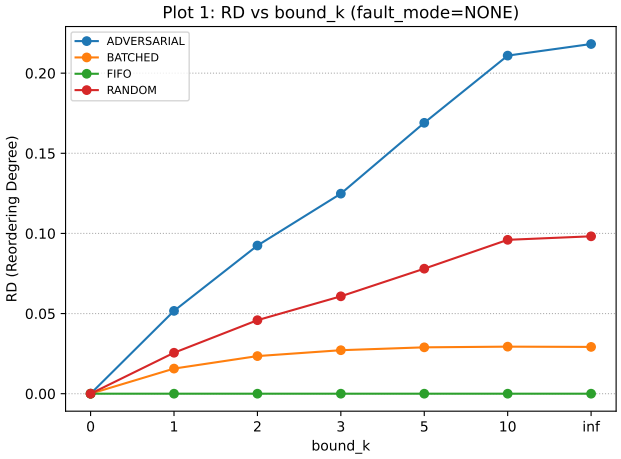

Figure 1: Mean RD (realized reordering degree) vs. reordering bound k (`fault_mode=NONE`, SWinf), aggregated over 100 schedule seeds per cell. Mean RD increases monotonically with k; at $k = inf$ Adversarial reaches RD≈ 0.218, Random RD≈ 0.098, Batched RD≈ 0.029, while FIFO remains at zero.

RD≈ 0.218 for Adversarial, RD≈ 0.098 for Random, and RD≈ 0.029 for Batched, while FIFO stays at zero.

These results validate the k-bound as an effective *control knob* over completion reordering. Importantly, RD does not merely track k itself but reflects how policies *exploit* the allowed freedom. Even under identical bounds, policies induce qualitatively different ordering behavior, justifying the need for systematic, policy-aware exploration rather than random schedule sampling.

For system designers, RD provides a quantitative bridge between specification-level freedom ("completions may reorder") and observable runtime behavior. Bounding reordering enables controlled stress testing without assuming pathological, fully adversarial hardware behavior.

4.3 RQ2: Tail-Latency Risk Cliffs under Completion Reordering

Our Hypothesis: Tail latency does not degrade smoothly with increasing reordering freedom but exhibits *risk cliffs*: sharp transitions into high-tail regimes once a policy-specific threshold in k is crossed.

Figure 2 reports p95 completion latency at $k = inf$ and `fault_mode=NONE`. FIFO remains flat with p95≈ 13.70 steps. Batched shows lower and stable tails (p95≈ 6.52 steps, achieving completion-coalescing via bursts of up to four consecutive `COMPLETE` steps). Random increases moderately to p95≈ 18.25 steps. In contrast, Adversarial jumps to p95≈ 28.46 steps (∼2× the FIFO baseline).

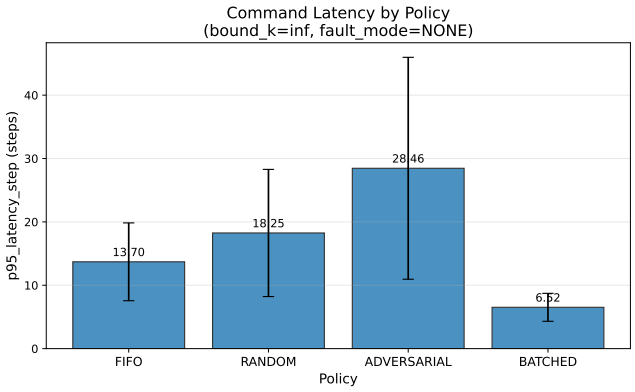

Figure 2: p95 completion latency (scheduler steps) by policy at $k = inf$ (fault_mode=NONE). Error bars show standard deviation across 100 schedule seeds per policy. Adversarial exhibits highest tail latency (28.46 steps). Batched achieves lowest (6.52 steps). FIFO and Random intermediate.

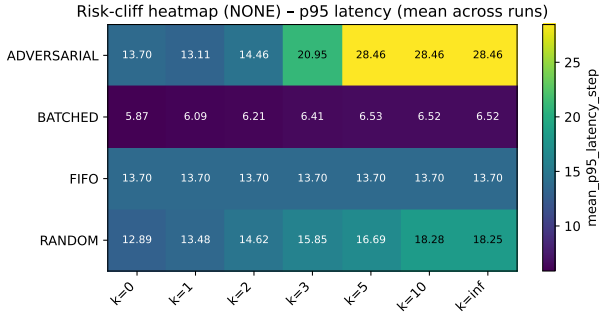

Figure 3: Risk-cliff heatmap: mean p95 latency (scheduler steps) over policy × reordering bound k (`fault_mode=NONE`). Adversarial exhibits a sharp jump at $k = 2 \rightarrow 3$. Random shows gradual increase. Batched remains stable. FIFO is flat (unaffected by k).

Risk-Cliff Heatmap. While Figure 2 reports the $k = inf$ slice, Figure 3 shows how tail latency evolves across the full k spectrum. Figure 3 visualizes mean p95 latency (measured in scheduler steps. Values are the mean of per-run p95 latencies) across all policies and reordering bounds $k \in \{0, 1, 2, 3, 5, 10, inf\}$.

Sharp Cliff (Adversarial): p95 remains low at $k = 0$ (13.70 steps) and $k = 1$ (13.11 steps), but jumps abruptly at $k = 2 \rightarrow k = 3$ from 14.46 to **20.95 steps** (+45%). For $k \geq 5$, p95 saturates at 28.46 steps. This sharp transition confirms our hypothesis: once $k \geq 3$, the adversarial policy's newest-first heuristic has enough freedom to systematically delay

older commands, triggering head-of-line blocking and severe tail-latency amplification.

Gradual Cliff (Random): Random exhibits continuous degradation from 12.89 (k=0) to 18.25 steps (k=inf, +42%). The absence of a sharp jump reflects unbiased selection: each increment in k marginally increases the probability of delaying older commands, but no single threshold triggers catastrophic behavior.

Robust (Batched): Batched remains stable across all k: 5.87–6.53 steps (< 11% variation). This robustness arises because completion coalescing (bursting up to four consecutive completions once the first completion is triggered) creates a regular pattern that dampens the effect of reordering freedom.

Baseline (FIFO): FIFO shows no variation (13.70 steps) because only the head element is selectable; the k-bound has no effect. We call a transition a *risk cliff* when mean p95 increases by more than 20% between adjacent k values (sharp) rather than changing gradually. These results demonstrate that QoS degradation follows a non-linear, policy-dependent trajectory. The sharp cliff for Adversarial is particularly striking. A single increment from $k = 2$ to $k = 3$ (moving from 2 to 3 selectable commands) precipitates a 45% latency jump. This behavior invalidates linear assumptions and underscores the need for systematic, bounded exploration rather than conservative assume-worst-case designs.

For NVMe firmware and host-stack developers, risk cliffs represent a critical failure mode. A device implementation that is demonstrably robust at $k = 2$ may violate strict tail-latency budgets at $k = 3$, even though both bounds are legal under the NVMe specification. This finding motivates the extraction of poison schedules (RQ3) as concrete regression-test artifacts that developers can integrate into CI/CD pipelines.

Mechanism: pending pressure to head-of-line blocking. To explain *why* the risk cliff emerges, we inspect queue backpressure via the pending set. In SWinf, Adversarial systematically delays older commands while completing younger ones within the allowed k-window. This increases the lifetime of long-running commands in the pending set and creates a sustained backlog. Concretely, for `fault_mode=NONE` the mean p95 under Adversarial increases from 14.46 steps at $k = 2$ to 20.95 at $k = 3$ and 28.46 at $k \geq 5$, while RD simultaneously rises (Section 4.2). Once k becomes large enough for the scheduler to repeatedly skip over stalled/old requests, the system enters a *high-tail state* where the pending set acts as a reservoir for tail amplification. A small fraction of commands is delayed across many ticks and eventually dominates the p95. In contrast, Batched keeps tails low even at $k = inf$ (mean p95=6.52) because completion coalescing reduces host-visible head-of-line exposure and limits the frequency at which single delayed requests can dominate the tail. Overall, the cliff is therefore best understood as a *backpressure transition*. Once the scheduler has enough freedom to perpetually defer some older commands, pending pressure and head-of-line effects amplify into a high-tail regime.

Table 1: Poison schedules (seed1, SWinf, fault_mode=NONE, $k = inf$). Top-10 runs ranked by p95 latency. RD values joined from run-level results. All belong to Adversarial policy.

Rank	schedule_seed	p95 (steps)	RD
1	10	64	0.383
2	18	64	0.524
3	53	64	0.722
4	26	62	0.395
5	30	62	0.373
6	59	62	0.444
7	76	62	0.462
8	98	62	0.549
9	27	60	0.475
10	62	60	0.470

Submit-window ablation (SW2/SW4): Limiting the number of in-flight commands caps the pending-set size, so realized reordering and tail impact saturate once k exceeds the typical pending depth. Accordingly, the risk-cliff boundary shifts with workload pressure: under lower concurrency the transition into the high-tail regime occurs at smaller effective k (or becomes less pronounced), whereas under SWinf policies can sustain a large pending reservoir and amplify tails. Practically, k should be interpreted relative to the typical pending depth, not as an absolute "safe" bound.

4.4 RQ3: Poison Schedules as Practical Robustness Artifacts

Worst-case schedules are highly structured and reproducible, making them valuable artifacts beyond aggregate statistics.

Table 1 lists the top-10 poison schedules at $k = inf$ under `fault_mode=NONE`, ranked by p95 completion latency. All ten schedules are produced by the Adversarial policy. The strongest poison schedule (schedule_seed=10) reaches a p95 latency of 64 scheduler steps with RD = 0.383, compared to the FIFO baseline mean p95 of 13.70 steps (Section 4.3). Even the weakest poison schedule in the top-10 (schedule_seed=62) still exceeds p95=60 steps, demonstrating that extreme tail behavior is systematically triggerable and not a rare outlier.

These schedules are not rare outliers but represent a distinct class of execution patterns that consistently trigger

Table 2: Generality check: mean p95 (fault_mode=NONE) for seed1 vs seed2 across k (SWinf).

Seed	Policy	k=0	1	2	3	5	10	inf
seed1	FIFO	13.70	13.70	13.70	13.70	13.70	13.70	13.70
seed1	Random	12.89	13.48	14.62	15.85	16.69	18.28	18.25
seed1	Batched	5.87	6.09	6.21	6.41	6.53	6.52	6.52
seed1	Adversarial	13.70	13.11	14.46	20.95	28.46	28.46	28.46
seed2	FIFO	4.84	4.84	4.84	4.84	4.84	4.84	4.84
seed2	Random	4.94	5.04	5.38	5.47	5.69	5.68	5.68
seed2	Batched	3.66	3.72	3.97	4.00	4.00	4.00	4.00
seed2	Adversarial	4.84	4.19	6.41	6.48	6.48	6.48	6.48

worst-case behavior. Their compact trace representation allows deterministic replay, enabling regression testing and targeted debugging of host-side logic.
Impact of our discussion: poison schedules turn robustness analysis into a practical engineering tool: instead of reasoning abstractly about bad schedules, developers can replay concrete traces that reliably expose tail-latency pathologies.

4.5 RQ4: Generality and Robustness of Findings

Seed Generality: A second workload seed (seed2, $n_{\text{cmds}} \approx 8$ commands vs. seed1's 32, Table 2) confirms that the qualitative phenomena are robust: all policies maintain their ranking (Batched lowest latency, FIFO close second, Adversarial highest), and the transition into a higher-tail regime remains visible but shifts to lower k (seed2 Adversarial: jump at $k = 1 \rightarrow 2$ vs. seed1's $k = 2 \rightarrow 3$). This shift reflects workload-induced pending pressure. Smaller workloads have less queue depth and thus reach the high-tail regime at lower reordering bounds. Absolute p95 values differ across seeds (consistent with different workload sizes and pending pressure), while the qualitative ordering and the presence/position of transition regimes remain stable.

Fault Modes: Table 3 reports mean p95 latency at $k = inf$ (seed1, SWinf) under different fault modes. Under `NONE`, all runs complete the full workload (`completion_rate`= 1.0, $n_{\text{ok}} = 32$), so p95 reflects genuine schedule-induced tail amplification. Under `TIMEOUT`, runs terminate early (median $n_{\text{ok}} \approx$ 19–20, `completion_rate`$\approx$ 0.59–0.63), which truncates the completion tail and thus reduces measured p95 even for adversarial schedules. Under `RESET`, the model deterministically resolves the run after exactly half of the commands ($n_{\text{ok}} = 16$, `completion_rate`= 0.5 across all policies), effectively capping backlog growth. Consequently, policy differences in tail latency are largely suppressed. Therefore, we interpret p95 primarily under `NONE` as a QoS metric, while

Table 3: Fault modes at $k = inf$: mean p95 latency (SWinf, seed1). Note: TIMEOUT/RESET truncate runs (reduced completion rate), which affects tail measurements.

Fault mode	FIFO	Random	Batched	Adversarial
NONE	13.70	18.25	6.52	28.46
TIMEOUT	12.08	16.22	6.36	24.71
RESET	11.28	12.05	5.82	9.52

`TIMEOUT`/`RESET` are reported as robustness/recovery behavior.

C-DUT Validation: To reduce single-implementation artifacts, we differentially validate a targeted subset of executions against an independent C implementation. Concretely, we replay the Top-K poison schedules (Table 1) and a small set of representative (policy, k, fault_mode) cells. Across this checked subset we observe no semantic mismatches, increasing confidence that the reported phenomena are not artifacts of a single implementation.

4.6 Limitations

NVMe-lite is a deterministic, logical model and does not predict microsecond-level performance. SWinf represents a stress scenario rather than typical deployment, and the evaluated workload size is fixed at $n_{\text{cmds}} = 32$ commands in the main experiment matrix. Nevertheless, these choices intentionally isolate schedule-induced effects, making the observed risk cliffs attributable to completion reordering rather than confounding hardware variability.

5 Conclusion

This paper systematically explores NVMe-lite host–device completion-schedule robustness via bounded reordering. We uncover sharp risk-cliffs in tail latency: at $k = 3$, Adversarial reaches $\approx 1.5\times$ FIFO p95; at $k \geq 5$, $\approx 2\times$. We extract poison schedules as regression-test artifacts and demonstrate that recovery mechanisms dominate schedule effects under faults. Key contributions: (1) systematic completion-schedule robustness exploration, (2) risk-cliff quantification, (3) poison schedules as practical evaluation artifacts.

Future Work

Extensions include empirical validation on real NVMe hardware with work-constraining schedulers [12], and extending NVMe-lite with host thread models to study OS scheduling interaction with device-side completion reordering. Further directions include CI/CD integration of poison schedules [4] and automated risk-cliff detection [2].

References

[1] Amazon Web Services. 2026. NVMe I/O operation timeout for Amazon EBS volumes. https://docs.aws.amazon.com/ebs/latest/userguide/timeout-nvme-ebs-volumes.html

[2] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching—Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 195–212. https://www.usenix.org/system/files/osdi18-berger.pdf

[3] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *PLDI*. https://www.cs.ucdavis.edu/~su/publications/pldi16.pdf

[4] Kai Cong, Fengwei Zhang, Shi Jiantao, Kun Cheng, Fan Wang, and Xiaoqin Zhao. 2015. Automatic Fault Injection for Driver Robustness Testing. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 361–372. doi:10.1109/ICSE.2015.55

[5] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

[6] Sönke Huster, Matthias Hollick, and Jiska Classen. 2024. To Boldly Go Where No Fuzzer Has Gone Before: Finding Bugs in Linux' Wireless Stacks through VirtIO Devices. In *2024 IEEE Symposium on Security and Privacy (SP)*. doi:10.1109/SP54263.2024.00024

[7] M. G. Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 30, 1–2 (1938), 81–93. doi:10.1093/biomet/30.1-2.81

[8] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. NVMeVirt: A Versatile Software-defined Virtual NVMe Device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, 379–394. https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon

[9] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 263–279. doi:10.1145/3477132.3483573

[10] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2022. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Transactions on Storage* (2022). doi:10.1145/3465406

[11] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. 404–416. doi:10.1145/3533767.3534226

[12] Tristan Miemietz, Jan Richling, and Mathias Pacher. 2019. K2: Work-Constraining Scheduling of NVMe-Attached Storage. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. 340–351. doi:10.1109/RTSS46320.2019.00039

[13] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs *(OSDI'08)*. USENIX Association, USA, 267–280.

[14] NVM Express, Inc. 2024. NVM Express Base Specification, Revision 2.0e. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0e-2024.07.29-Ratified.pdf. Ratified 2024-07-29.

[15] NVM Express, Inc. 2025. NVM Express NVMe over PCIe Transport Specification, Revision 1.3 (2025.08.01 Ratified). https://nvmexpress.org/wp-content/uploads/NVM-Express-NVMe-over-PCIe-Transport-Specification-Revision-1.3-2025.08.01-Ratified.pdf

[16] OASIS. 2016. Virtual I/O Device (VIRTIO) Version 1.0. https://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html

[17] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2022. Optimizing Storage Performance with Calibrated Interrupts. *ACM Trans. Storage* 18, 1, Article 3 (March 2022), 32 pages. doi:10.1145/3505139

[18] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. doi:10.1109/SP46215.2023.10179293