Rethinking Provenance Completeness with a Learning-Based Linux Scheduler

Jinsong Mao
jinsongmao@umass.edu
UMass Amherst
Amherst, MA, USA

Benjamin E. Ujcich
Georgetown University
Washington, D.C., USA

Shiqing Ma
UMass Amherst
Amherst, MA, USA

Abstract

Provenance plays a critical role in maintaining traceability of a system's actions for root cause analysis of security threats and impacts. Provenance collection is often incorporated into the reference monitor of systems to ensure that an audit trail exists of all events, that events are completely captured, and that logging of such events cannot be bypassed. However, recent research has questioned whether existing state-of-the-art provenance collection systems fail to ensure the security guarantees of a true reference monitor due to the "super producer threat" in which provenance generation can overload a system to force the system to drop security-relevant events and allow an attacker to hide their actions. One approach towards solving this threat is to enforce resource isolation, but that does not fully solve the problems resulting from hardware dependencies and performance limitations.

In this paper, we show how an operating system's kernel scheduler can mitigate this threat, and we introduce Aegis, a learned scheduler for Linux specifically designed for provenance. Unlike conventional schedulers that ignore provenance completeness requirements, Aegis leverages reinforcement learning to learn provenance task behavior and to dynamically optimize resource allocation. We evaluate Aegis's efficacy and show that Aegis significantly improves both the completeness and efficiency of provenance collection systems compared to traditional scheduling, while maintaining reasonable overheads and even improving overall runtime in certain cases compared to the default Linux scheduler.

1 Introduction

The global average cost of a data breach in 2024 marked a 10% increase from 2023, reaching an all-time high [50], underscoring the critical need for robust defense and forensic mechanisms. Provenance systems address this need by systematically capturing and tracing the origins and pathways of data flows, offering invaluable insights into the evolution of data over time. This functionality is essential for applications such as forensic analysis and intrusion detection. By recording detailed interactions within computing environments, provenance systems promote transparency and accountability, even in highly complex scenarios. The reliability and effectiveness of these systems, however, hinge on the *completeness* of captured provenance events. In other words, provenance systems should follow the reference monitor design [26], ensuring that *all* system events are captured and logged (i.e., guaranteeing the completeness of collected provenance).

Revisiting reference monitor guarantees We revisit the reference monitor concept in the context of provenance systems and find that existing solutions do not adequately guarantee the completeness of provenance events. In particular, the "super producer threat" [55] in which provenance event generation overloads the system and leads to losses of capturing security-relevant events, poses a significant challenge to the reference monitor guarantees. This attack is surprisingly effective and easy to implement, as it does not require any special privileges or knowledge of the provenance system.

Recent solutions, such as NoDrop [55], propose resource isolation so that the host process consumes its own generated events to prevent provenance event loss. Such losses are curtailed by injecting consumer code into the host process and enforcing its execution to clean the full-filled event buffer. Unfortunately, such resource isolation is challenging to deploy: it requires specialized hardware support (i.e., hardware-based Intel Memory Protection Keys), it is tightly coupled to specific legacy kernels, and it introduces issues when executing atomic operations that may lead to errors (e.g., `Scheduling while atomic`) and potential system crashes. Additionally, the enforcement of injected consumer code may introduce performance overheads and fairness issues.

Our insight The root cause of why super producer threats exist is that the provenance system is not timely consuming the generated events, leading to buffer overflows and event loss. Our insight is that a kernel scheduler can solve this problem by ensuring that the provenance system is allocated sufficient resources to consume the generated events in a timely manner. However, tackling this challenge is not straightforward. Traditional schedulers, such as Linux's default schedulers (e.g., the Completely Fair Scheduler (CFS) [3] and the Earliest Eligible Virtual Deadline First (EEVDF) [5] scheduler), are designed to provide a generic and balanced scheduling experience optimized around performance metrics like throughput, fairness, and latency among generic workloads, but they lack an understanding of provenance systems' unique characteristics and requirements. This deficiency can result in lost provenance, which compromises

provenance's reliability and security and the overall system's reference monitor guarantees.

Overview The design of Aegis is driven by three primary goals: completeness, efficiency, and fairness. Completeness ensures that all provenance events are captured, even under extreme workloads, safeguarding the integrity and reliability of provenance systems. Efficiency aims to maintain or improve system performance by optimizing resource allocation while minimizing computational overhead. Fairness ensures balanced resource distribution, preventing the starvation of provenance-related tasks or the over-prioritization of any specific task, thus maintaining proportional CPU utilization across diverse workloads.

To achieve these goals, Aegis introduces a kernel-space learned scheduler tailored to the unique demands of provenance systems. The method combines a queue-based scheduling framework with reinforcement learning, offering dynamic adaptation and resource optimization.

At a high level, Aegis organizes tasks into multiple queues, each equipped with predefined waiting times and resource budgets. Provenance-related tasks are allocated to non-primary queues, which enforce strict resource constraints, while general tasks are managed by a primary queue. By dynamically selecting tasks from the most "hungry" eligible queue, Aegis ensures fair and proportional resource allocation, preventing starvation and maintaining system stability.

Aegis is powered by a lightweight Deep Q-Network (DQN) that predicts the optimal scheduling decisions based on task behavior and system context. This reinforcement learning model analyzes features, such as event generation rates, buffer availability, and task runtime patterns, to dynamically adjust queue placement. A dual reward mechanism underpins the learning process: a provenance reward penalizes event loss to ensure provenance completeness, while a utilization reward minimizes CPU idleness to enhance overall efficiency. This reward enables Aegis to learn and adapt to diverse workloads, addressing the challenges posed by unpredictable and bursty event streams. To further improve efficiency, Aegis incorporates a delta function to skip unnecessary scheduling decisions when task contexts are stable, reducing computational overhead. It also employs an exponential waiting time strategy for queue consumption, ensuring predictable scheduling and precise resource control.

Ensuring the security and stability of the extended kernel functionality, Aegis is implemented within the Linux kernel using the eBPF subsystem and the `sched_ext` framework. Both provide strong guarantees through rigorous verification and controlled kernel extensions. This kernel-space implementation also avoids unnecessary data transfer and context switching, collecting provenance and task features directly from kernel data structures. By integrating core components into the kernel, Aegis ensures compatibility with modern Linux versions and minimizes integration complexity.

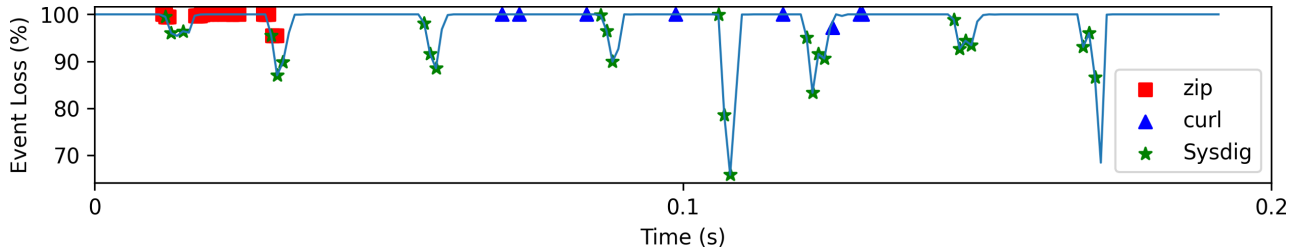
Contributions We evaluate Aegis with eight benchmarks, two provenance systems and compare it with three other schedulers. Results show that Aegis is capable of preventing provenance event loss while other schedulers lose 39% to 98% events. In the meantime, Aegis respectively improves the runtime of three macro benchmarks by an average of 3.42%, 0.90% and 0.25% compared to EEVDF[5], LAVD[12], and Rusty[13] schedulers. Our contributions are summarized as:

1. We revisit the long-standing reference monitor concept in the context of provenance systems and discover that the super producer threat, which leads to the loss of provenance events, poses a significant challenge to the reference monitor guarantees even in state-of-the-art systems.
2. We propose and implement a novel reinforcement learning based scheduler, Aegis, that achieves security guarantees and desired performance properties. To the best of our knowledge, Aegis is the first machine learning based kernel scheduler. Our tool will be open-sourced to encourage wider adoption in the research community.
3. We evaluate Aegis with macro and micro benchmarks, illustrating its superior effectiveness and performance compared to existing schedulers.

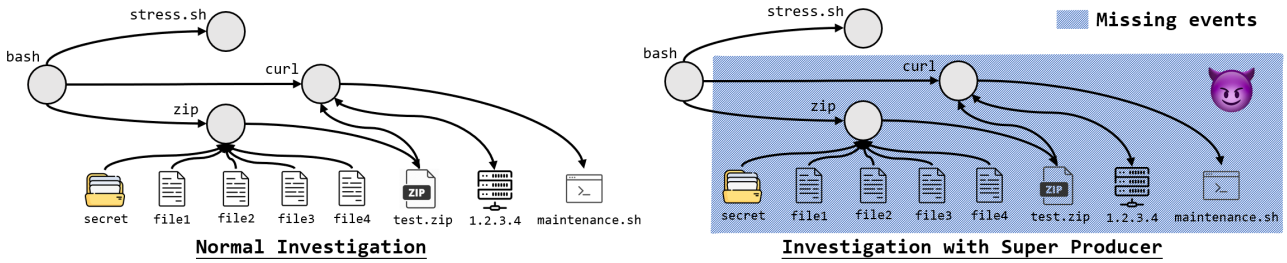
2 Background & Challenges

Provenance Provenance systems serve as the foundation for understanding the lifecycle of data within computational systems. These systems capture, record, and analyze detailed information about the origins, transformations, and consumption of data as it flows through various processes. Provenance data is critical in various applications, ranging from system auditing[29, 35, 51, 72, 81], forensic analysis[30, 33, 74, 89, 95], data security[23, 38, 79, 86, 90] and reproducibility[27, 52, 53, 80, 83]. For example, by capturing a comprehensive record of system events, provenance systems enable the reconstruction of attack vectors, detection of unauthorized access, and identification of malicious activities[23, 35, 38, 48, 58, 59, 84, 91, 96].

Provenance systems are typically structured around *producers* and *consumers*. Producers use hooks integrated across various layers of the software stack that enable the capture of a wide range of activities (e.g., file operations, network sockets, IPC, process execution). For instance, hooks embedded within the operating system kernel can capture system calls such as `open`, `read`, and `write` for file access or `fork` and `execve` for process creation and execution. These kernel-level hooks ensure low-level event capture with minimal latency. Consumers, in contrast, receive, store, and analyze the produced provenance data. Consumers can range from simple storage mechanisms that generate logs for offline analysis to sophisticated online analytics engines. However,



(a) Sysdig suffers from inadequate scheduling and resource allocation. The blue line indicates what percentage of provenance events are lost due to buffer overflow.



(b) Causal graphs generated during attack investigations, assuming no provenance event loss (left) and provenance event loss caused by a super producer (right). With a super producer, the adversary hides evidence of malicious events since such events were not captured.

Figure 1. Scheduling statistics and causal graphs of the Sysdig and EEVDF "super producer" [55] case study example.

producers and consumers share a common challenge: efficient and provenance-aware scheduling mechanisms to ensure no events are lost or excessive overhead burdens.

Reference monitors Reference monitors [26] are a fundamental security mechanism in operating system design that enforce system-critical security policies (e.g., access control) by acting as an interposer on every access attempt between subjects (e.g., users or processes) and objects (e.g., files, devices). They provide the properties of being non-bypassable, evaluable, always invoked, and tamper-proof.

Most existing system-level provenance solutions[14, 16, 17, 31, 55, 81] embed provenance collectors within the overall system's kernel-space reference monitor like Linux Provenance Modules[31] or eBPF subsystem to ensure that the access decisions, along with their context (e.g., who accessed what and why), are recorded securely and cannot be bypassed, with the reference monitor serving as part of the trusted computing base (TCB). This enhances the overall system's security and provides the basis for provenance-informed audit trails.

2.1 Case Study: Super Producer Threat

We consider a representative vulnerability of a provenance system within a reference monitor when confronted with the "super producer threat" [55] in which an adversary attempts to defeat the reference monitor security guarantees. We devise a representative example using Sysdig [14], a widely used open-source provenance system, and the earliest eligible virtual deadline first (EEVDF) scheduler that is enabled by default in the Linux kernel. Sysdig continuously collects and logs system events, saving the provenance data to the local disk.

Attack We assume that an adversary aims to exploit scheduling weaknesses to mask malicious events. The adversary deploys a "super producer," which generates a significant volume of system calls. While the super producer is running, the adversary compresses a secret folder, `secret`, which contains four files (`file1` to `file4`), into an archive, `secret.zip`. The compressed archive is then uploaded to a remote server at `1.2.3.4` using `curl`. Subsequently, the adversary downloads a script, `suspicious.sh`, from the same host and overwrites a previously benign script, `maintenance.sh`, with this malicious payload. The numerous events triggered by the super producer overwhelm the consumer buffer, leading to missed events associated with these malicious operations.

Results Figure 1a illustrates the event loss statistics and the scheduling order of all programs during the malicious operations. The blue line represents the average number of events lost over 1 ms intervals. Stars, squares and triangles indicate when a corresponding program is scheduled. The results reveal that the consumer underwent insufficient scheduling and the system experienced a significant loss of provenance events during the super producer due to buffer overflow. All the malicious workloads were executed in the context of near 100% event loss, indicating the provenance system is not recording these malicious operations.

Figure 1b shows the causal graphs generated during attack investigation. Normally the provenance system collects all

events (left), and the causal graph shows that the adversary who invoked `stress.sh` also visited `secret` and its files. The upload and consequent injection to `maintenance.sh` are also shown. However, due to incomplete provenance log (right), the records of these malicious operations are lost and the adversary successfully hides the malicious events.

2.2 Defense Challenges

We identify what makes the super producer threat challenging to defend against with existing approaches.

Low-cost barrier to entry The attack does not cost an adversary much in resources to effectively perform. In our experiment, the stress imposed on the provenance system was minimal, achieved by scanning the `/usr` directory with just five concurrent threads in addition to data collection. This type of operation can be invoked by any user at any time, regardless of the machine type or scenario, making it easy to execute. Furthermore, the attack is also stealthy: the atack operates independently and is not directly linked to any subsequent malicious actions or external software. As a result, such activity is more likely to be misclassified as benign.

Limitations of existing defenses Based on known prior work, we found that nearly all state-of-the-art provenance systems suffer from this attack, and state-of-the-art solutions fail to fully address the problem, as summarized in Table 1.

We measured three key aspects of a provenance system that are important to the attack: *completeness* (whether the system ensures no provenance event loss or is capable of performing mitigation); *budget-aware* (whether the system is aware of resource allocation and makes efficient use of it); and *compatibility* (whether the solution can perform generally regardless of software, hardware and provenance systems requirements). We find the following observations and trends:

- Nearly all modern provenance systems[14, 16, 17, 43, 76, 81] suffer from event loss, which undermines the reference monitor security guarantees of a provenance system.
- Some systems [20, 81] are budget-aware since they prioritize critical events to be stored in a buffer, but they are not aware of the overall system's capacity or the buffer state. As a result, they do not prevent buffer overflow.
- Some systems require specific auditing devices [20, 42], software that lives outside of the standard kernel source tree [81], or hardware-based defenses [55], which precludes those systems from general wider deployment.
- Efficient encoding [81] only delays failure: the buffer fills at the producer's rate, and overflow occurs when production outpaces consumption. In the meantime, a larger buffer is bounded with high-end specifications and is not applicable to all scenarios.

Table 1. State-of-the-art provenance systems.

Provenance System	Completeness	Budget-Aware	Compatibility
eAudit[81]	No	Partial	Portable
HardLog[20]	No	Partial	Constrained
NoDrop[55]	Partial	Yes	Constrained
Sysdig[14]	No	No	Yes
auditd[43]	No	No	Yes
Tetragon[16]	No	No	Portable
Tracee[17]	No	No	Portable
Camflow[76]	No	No	Portable

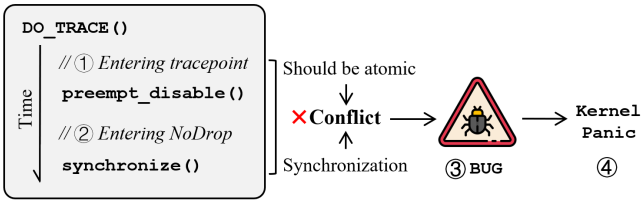


Figure 2. NoDrop needs to perform synchronization in critical regions, causing kernel bugs, deadlocks and potential kernel panic [1, 7, 15].

The most relevant state-of-the-art defense is NoDrop [55], which achieves completeness by introducing resource isolation in a per-thread manner to process and store events. However, as shown in Figure 2, its design requires synchronization inside a tracepoint, which is an atomic section. This synchronization can lead to scheduler timeout and forced rescheduling, incurring potential subsequent deadlocks and kernel panics [1, 7, 15]. In addition, its resource isolation design depends on hardware-based Intel memory protection keys that limits which platforms it can be deployed.

Ideally, we want a system that is free from hardware constraints, is portable, is performant and works seamlessly with the kernel. We consider the *kernel scheduler* as a way to overcome those challenges, which offers fundamental and universally available mechanisms for resource allocation.

3 Aegis Goals

Threat model We assume that the scheduler and the provenance system are part of the system's trusted computing base (TCB) and that the scheduler is capable of accessing statistics from the provenance system. Any user can invoke any userspace workload that does not harm kernel space, including potential super producer invoked by an adversary. The security of the transmission and storage of provenance logs is well-studied[20, 24, 32, 37, 39, 44, 82, 84, 88, 96] and beyond the scope of this paper.

3.1 Guarantees and Desired Properties

Based on the threat model, we consider the following guarantees and properties of what a scheduler should provide.

First, we define a set of hard guarantees that Aegis must uphold to maintain the completeness and usability of provenance systems, particularly under adversarial or high-load conditions. Second, we outline a set of desired properties that, while not strictly required, enhance the performance and stability of Aegis in diverse environments.

Guarantees Since the attack originates from corrupting the availability of provenance events, such integrity of the completeness of collecting all provenance is the top priority of the scheduler. At the same time, Aegis must prevent starvation. We propose the guarantees as follows:

G1. ***Overload prevention:*** The scheduler must have the capability to restrict the tasks' resource allocations (Section 4.1.1). This capability must be applied effectively to postpone or throttle jobs that risk exceeding the system's safe capacity. (Section 4.2.1)

G2. ***Quality of service:*** All tasks scheduled by Aegis must be guaranteed to complete execution within a predictable time, ensuring consistent forward progress. (Section 4.2.1)

Desirable Properties Additionally, the scheduler should minimize the performance impact on the operating system:

G3. ***Fairness:*** Each runnable process should receive a proportional share of CPU time over the long run, preventing starvation and ensuring balanced resource allocation. (Section 4.1.1)

G4. ***Throughput:*** The system should maximize the total amount of work completed in a given timeframe, increasing overall efficiency. (Section 4.2.1)

G5. ***Overhead:*** The computational costs associated with scheduling should be minimized, including decision making and regular maintenance. The scheduler should also minimize other resource consumptions. (Section 4.2.2)

For practical concerns, we want Aegis to be forward compatible and adaptable to different provenance systems.

By achieving these goals, the scheduler can help ensure that the provenance system meets the reference monitor security guarantees, particularly the properties of being non-bypassable, evaluable, and always invoked.

3.2 Scheduling Solutions and Limitations

While there are various mechanisms for task prioritization and resource management that we could attempt to use to defend against the super producer threat (e.g., tuning the default Linux scheduler or components to mitigate the attack), we argue that such mechanisms fall short in satisfying the overload prevention guarantees and performance properties that we want for a provenance scheduler, as summarized in Table 2.

Nice values reflect the priority of tasks, which influences their scheduling order. Adjusting time slices will change the CPU resource assigned to the tasks. While these mechanisms allow us to prioritize the consumer, they *struggle to prevent overload during bursty workloads.* This is because either CFS or EEVDF are designed to be fair among all tasks. Similar to what we have shown in Figure 1a, their fair-oriented vruntime mechanism does not guarantee the buffer is consumed in time.

Table 2. Existing adjustable scheduling options.

Scheduling Options	Reason
Adjusting nice values	✘ No overload prevention
Adjusting slices	✘ No overload prevention
Cgroups	✘ No overload prevention [55]
Consumer preemption	✘ Performance downgrade
Pinning to cores	✘ Performance downgrade

Consumer preemption[20] could give a provenance consumer priority by interrupting other tasks. While this can temporarily alleviate the load on the provenance system, it leads to *significant performance degradation for non-provenance tasks.* Consider a multi-core system where the provenance system share a global buffer. When a preemption is invoked, it has to stop tasks on all cores to prevent new event generation and buffer overflow, which leads to heavy performance degrade. Pinning the provenance consumer to specific cores can isolate its execution from other tasks, theoretically reducing contention. However, this approach *results in suboptimal core utilization and overall performance downgrades.*

Control groups (cgroups), a mechanism in the Linux kernel that allows for hierarchical resource management and isolation, provides fine-grained control over resource allocation and can be used to prioritize provenance tasks. However, cgroups alone is *insufficient for defending against the super producer threat* [55] because such defense requires additional components to determine proper cgroups resource allocation.

We argue that a data-driven approach is needed. Traditional schedulers are well-engineered with simple heuristics. However, they can only react once pressure is already high, and any mitigation will arrive too late. What is needed instead is a data-driven scheduler that reasons over richer signals—context-switch patterns, event-generation rates, and more features, to estimate the incoming stress and to allocate CPU time proactively. Thus, in this paper we propose Aegis.

We summarize the properties provided by Aegis in Table 3, compared with other schedulers. Compared to MLFQ and MLQ, Aegis provides better fairness design that aligns with CFS and EEVDF scheduler and never starves. Aegis provides overload prevention capability over CFS, EEVDF and MLFQ. The superior properties of Aegis, tuned with a data-driven approach, deliver a powerful and performant scheduler for provenance systems.

4 Aegis Design

Figure 3 shows the overall system design of Aegis. Aegis uses a two-layer architecture. The first layer establishes the

Table 3. Properties of Aegis compared to existing schedulers. CFS / EEVDF are Linux default schedulers, whereas MLQ and MLFQ share Aegis 's multi-queue architecture.

Scheduler	Fairness (CPU Share)	Starvation Guarantee	Overload Prevention
CFS[3]	Proportional to priority	✓ Task eventually runs	✘ No capping mechanism
EEVDF[5]	Proportional to priority	✓ Task eventually runs	✘ No capping mechanism
MLFQ[36]	Related to CPU and I/O usage	✓ Task at least runs per fixed time	✘ Uncapped by priority boost
MLQ[9]	Related to priority	✘ Task starves due to queue domination	✓ Capped by queue domination
Aegis	Proportional to priority	✓ Task at least runs per fixed time	✓ Capped by queue waiting-time budget

backbone of Aegis (Section 4.1) with a complete scheduling framework and foundational capabilities of overload prevention and fairness. The second layer tunes this backbone with specific *optimization* goals (Section 4.2), such as attack mitigation and throughput maximization, to enable Aegis to adapt to the underlying machine and provenance system. Simultaneously, Aegis implements measures to minimize scheduling overhead, ensuring that the enhancements do not introduce significant performance penalties.

Security Implications Although Aegis introduces custom scheduling logic outside the default kernel schedulers, its design maintains strong reference-monitor guarantees. This is ensured by two mechanisms: eBPF and `sched_ext` . All Aegis components are implemented using eBPF, which enforces strict verification at load time, preventing unsafe or malicious code from entering the kernel[4]. In addition, `sched_ext` provides a built-in fallback to the default scheduler (e.g., EEVDF), ensuring that the system can safely recover in the event of misbehavior or starvation, preserving system integrity and availability.

4.1 Aegis Backbone

Similar to the classic multi-level queue (MLQ) and multi-level feedback queue (MLFQ) scheduling, a task under Aegis is placed into an appropriate queue according to its behavior. However, the keys in Aegis are ① the unique queue design gives Aegis overload prevention capability; ② Aegis collects the runtime statistics of the task, and uses a neural network to dynamically assign it to queues. Aegis also implements queue-based fairness, and provides a restricting capability with queue-wise budget. We first discuss how Aegis gains such capabilities, then show how Aegis adapts towards different workloads and provenance systems.

4.1.1 Ensuring balanced producer-consumer processing: Aegis queue design.

Aegis assigns tasks into different first-in-first-out (FIFO) queues with a fixed time slice. When a task becomes runnable, it is first moved to a dispatch queue. When the CPU asks for task to run, a queue is consumed (i.e., the first task in the selected queue is moved to the local run queue and is run). As a result, every task in the same queue has an equal chance to run.

Aegis employs a *primary queue* and several *non-primary queues*. The primary queue, operating on a FIFO basis, provides a default scheduling mechanism such that when all tasks are directed to this queue, Aegis behaves like a traditional FIFO scheduler. The overload prevention capability comes from non-primary queues. Non-primary queues are also FIFO but with designated waiting times, which are designed as natural resource budgets. A non-primary queue becomes eligible for consumption only when the elapsed time since the last task execution in that queue exceeds its waiting time, which enforces a strict resource allocation policy and ensures that tasks within one non-primary queue share the same pre-allocated budget. When all non-primary queues are not eligible to be consumed, the primary queue is consumed.

Takeaways: Aegis is able to achieve a balanced and controlled production-consumption process. That mitigates the risk of exceeding the system's safe capacity.

Fairness To achieve fairness among all queues, when the CPU is idle and requests tasks to execute, Aegis consumes the most "hungry" non-primary queues—those that have waited the longest since their last execution. If only the primary queue is eligible, the primary queue is consumed. This scheduling strategy ensures that queues operate in a fair and predictable manner.

Let N be the number of queues, where queue 1 being the primary queue and queue 2 to N being the non-primary queues. Let $T = \{t_1, ..., t_N\} \in \mathbb{N}^N$ be the time elapsed since the queues were picked and $\hat{T} = \{\hat{t}_2, ..., \hat{t}_N\} \in \mathbb{N}^{N-1}$ be the waiting time of the queues. A queue is more hungry when the waiting time is less and the elapsed time is more. We define the hungry factor h as follows:

$$\begin{aligned} h_i &= \frac{t_i}{\hat{t}_i},\ i > 1 \\ H &= \{h_1, ..., h_N\} \in \mathbb{R}^N \end{aligned} \tag{1}$$

Specifically, $h_1 = 0$ because the primary queue is only consumed when no other queues are eligible, and thus, the least hungry. Denote the number of tasks in the queues as $M = \{m_1, ..., m_N\} \in \mathbb{N}^N$. Aegis always chooses the most hungry eligible non-empty queue k as shown in Equation 2:

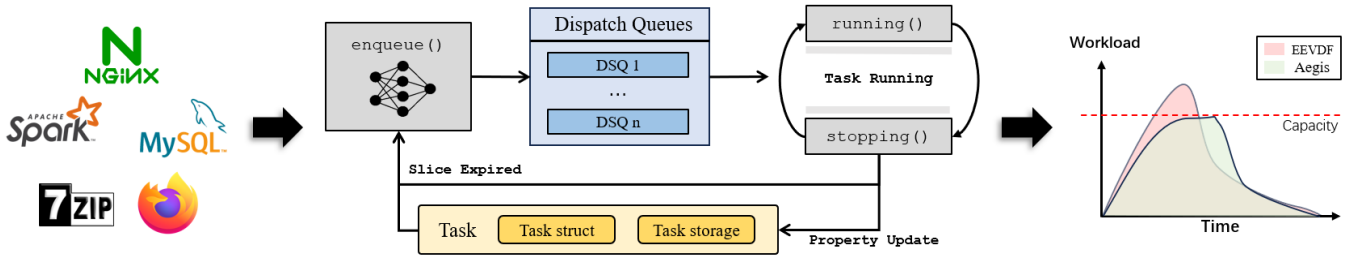


Figure 3. Overview of Aegis, using task and provenance features as input to ensure performant workloads with no event loss.

$$k = \text{argmax}_i(\ h_i \mid h_i \geq 1, m_i > 0) \tag{2}$$

Aegis provides no-starvation guarantee and proportional resource allocation for non-primary queues. Let the waiting time be in increasing order $\hat{t}_2 < ... < \hat{t}_N$. In the worst case for the lowest priority queue N, Aegis runs on a single-core system where low-priority queues have less chance to run than in a multi-core system, and the fixed time slice $s > \hat{t}_N$ so that high-priority queue will more likely compete with low-priority ones. We prove that the lowest priority queue, N, does not starve. Given any state at the beginning, let the elapsed time be t. Queue 3 is consumed when $\frac{t}{\hat{t}_3} > \frac{s \cdot N}{\hat{t}_2}$, giving the guarantee of queue 3 as $t_3 < \frac{s \cdot N \cdot \hat{t}_3}{\hat{t}_2}$. Similarly, we obtain the guarantee for queue N as follows:

$$t_N < \frac{s \cdot N \cdot \hat{t}_N}{\hat{t}_2}, \quad s > \hat{t}_N > ... > \hat{t}_2 \tag{3}$$

This indicates the consumption time is bounded and the bound for each queue is proportional to its waiting time.

Aegis provides finite execution guarantees. Denote $r_{N,RR}$ as the execution time of a certain task in queue N using round robin(RR) scheduling, $r_{N,\text{Aegis}}$ as of that using Aegis and r_{RAW} as the raw time needed to finish the task. Given the lower bound of no-starvation time in Equation 3, we gain the lower bound of $r_{N,\text{Aegis}}$ as following:

$$r_{N,\text{Aegis}} \leq \frac{\hat{t}_N}{\hat{t}_2} \cdot N \cdot m_N \cdot r_{RAW} \tag{4}$$

At the same time, we consider the same amount of tasks running in a RR system. The minimum number of tasks would be $N + m_N - 2$. Thus we have the time for $r_{N,RR}$ as following:

$$r_{N,RR} = (N + m_N - 2) \cdot r_{RAW} \tag{5}$$

Considering $N \geq 2$, we have an intuitive comparison between Aegis and RR:

$$\begin{aligned} r_{N,\text{Aegis}} &\leq \frac{\hat{t}_N}{\hat{t}_2} \cdot \frac{N \cdot m_N}{N + m_N - 2} \cdot r_{N,RR} \\ &\leq \frac{\hat{t}_N}{\hat{t}_2} \cdot N \cdot r_{N,RR} \end{aligned} \tag{6}$$

This indicates Aegis ensures that even tasks in the lowest-priority queue finish in a reasonable time - proportional to number of queues set and the corresponding waiting time - compared to the classic RR scheduling. In real-world cases, the execution time is often significantly less than the bound, through per-machine adaptation.

Takeaways: By preventing starvation and achieving proportional CPU share, tasks complete in long run with queue-based fairness.

4.1.2 Understanding the adaptation space.

The backbone leaves an adaptation space consisting of the neural network and the waiting time configuration.

Neural network and features An important question is which queue to put a specific task in. The behaviors of tasks can vary, and it is difficult for traditional methods to model and optimize over tasks, especially with different provenance configurations. In Aegis, we address the issue with a neural network. A neural network can be extensibly deployed to predict the appropriate queue for the task, as long as the network is well trained and adapts to the current configuration.

We configure the neural network to make queue predictions according to the context of the task, which describes the recent behavior of it and serves as input of the neural network. We set up the context with *task features* that describe how the task behaves and interacts with other tasks and *provenance features* that tell how the task produces events, whether the provenance system is capable of keeping all the events and its consuming speed, and the provenance system's average latency and availability at this time. A detailed overview of the task context is shown in Table 4.[1] With these features, the context gives comprehensive information about the task's nature and impact, and the provenance system's current state.

Waiting time Searching for a suitable neural network with undetermined waiting times is difficult. Potential combinations lead to a impractical searching space. To address this, we narrow the searching space by setting the waiting times based on the worst case. We first subject the provenance system to peak workloads by manually generating as many system calls as possible. During this stress testing phase, we

[1] `wait_freq` and `wake_freq` are adopted from the LAVD[12] scheduler; other features can be obtained from `task_struct` or the provenance system.

Table 4. Task and provenance context features collected by Aegis.

Category	Metric	Description
Task Features	`runtime`	Total CPU time used by this task
	`nvcsw`	Average number of voluntary context switches
	`wait_freq`	Frequency that the task waits for precedent tasks
	`wake_freq`	Frequency that the task wakes subsequent tasks
	`nr_stop`	The number of times this task stops in current slice
	`nr_idle`	The number of slices taken by idle task before this task
	`prio`	The queue that the task is currently in
Provenance Features	`nr_event`	Average number of events generated by the task
	`nr_drop`	Average number of events generated by the task but failed to collect
	`latency`	Average distance of consumer and producer
	`availability`	Current availability of provenance system's buffer

Table 5. Example scheduling with $\hat{T} = \{2, 4, 8\}$. Assuming the CPU has been idle for 8 time slots at start, the table gives the hungry factors and the final scheduling decision on which queue to consume.

Time	1	2	3	4	5	6	7	8	9	10	11	12
h_4	1	$\frac{9}{8}$	$\boxed{\frac{5}{4}}$	-	-	-	-	-	-	-	1	$\boxed{\frac{9}{8}}$
h_3	2	$\boxed{\frac{9}{4}}$	-	-	-	1	$\boxed{\frac{5}{4}}$	-	-	-	$\boxed{1}$	-
h_2	$\boxed{4}$	-	1	$\boxed{\frac{3}{2}}$	-	$\boxed{1}$	-	$\boxed{1}$	-	$\boxed{1}$	-	1
Result	2	3	4	2	1	2	3	2	1	2	3	4

configure only one non-primary queue and allocate all the stressing workloads to it. We assign other tasks, including the provenance consumer to primary queue to simulate the worst case. We then incrementally increase the waiting time until the system reaches a stationary state where no events are lost.

Let the resulting waiting time be denoted as $\hat{t}_\infty$. This indicates setting one non-primary queue with waiting time $\hat{t}_\infty$ leads to a baseline where the scheduler is capable of handling the worst case. However, the baseline fails to provide a fine-grained budget control and thus, leads to sub-optimal performance. Building upon this baseline, we generalize this approach for a system configured with N queues. With the intuition that exponential waiting times help queues being consumed in an ordered and predictable manner, we set the waiting time for each queue exponentially as follows:

$$\hat{t}_i = (\hat{t}_\infty)^{(i+0.5)/N},\ 1 < i \leq N \tag{7}$$

An example of a scheduling with exponential waiting time $\hat{T} = \{2, 4, 8\}$ is illustrated in Table 5. Within 12 time slots, the primary queue is consumed 2 times. The second, third and fourth queues are non-primary, and they are picked for 5, 3, 2 times, respectively. If all queues are not empty and reach stationary process, the three non-primary queues will have 50%, 25% and 12.5% share of the processor, leaving a 12.5% share to primary queue. The example shows that, with an exponential setting, the queues run in a fine-grained and predictable manner.

4.2 Aegis Learning Component

As shown in Figure 3, a dispatch cycle consists of a task running, the context being updated, and the network determining the re-scheduling decision. The task is then placed in a queue and waits to be run. In this section, we discuss how Aegis utilizes reinforcement learning to achieve zero event loss and high throughput with the backbone based on dispatch cycles.

4.2.1 Predicting task behaviors: RL in Aegis. Due to the dynamic nature of provenance workloads, traditional heuristic-based schedulers are unable to adapt to the diverse behaviors of provenance tasks. To address this, Aegis employs reinforcement learning (RL) to predict task behaviors and make intelligent scheduling decisions.

The overall RL training process is to maximize the expected reward by learning the optimal policy, i.e., the optimal action to take in a given state. In the context of Aegis, the state S is defined as the context of the task, which includes the task's behavior and the provenance system's state. The action A is the decision of which queue to put the task in, and the network is the neural network that predicts the action. Specifically, for the i-th queue q_i, we define the action table $A = \{a_i, \ldots, a_N\}$, where each action a_i puts the task into q_i:

$$a_i : (\text{task} \rightarrow \text{queue}_i) \tag{8}$$

Aegis employs state-of-the-art Deep Q-Network (DQN[71]) to predict actions the scheduler should take. Let the weight parameters of the DQN be θ. The DQN predicts Q-values, the reward, for each action and selects the k-th queue with the highest reward:

$$k = \text{argmax}_i\{\theta(S)\} \tag{9}$$

The reward r is the addition of r_c and r_p, $r = r_c + r_p$, reflecting both the scheduler's completeness goal using r_c and performance goal with r_p, as discussed in the following sections.

Optimizing provenance availability During training for provenance completeness, Aegis aims to zero out event loss in the simulated runs. At cycle t, let the total number of events be E and the number of dropped events be E^d, we define the provenance reward r_c as follows:

$$r_c = -\frac{E_t^d}{E_t} \tag{10}$$

The network is trained to minimize the event loss in the next cycle, $t + 1$. Knowing that there will always be scheduling policies that lead to zero event loss, e.g., inadvertently incentive the scheduler to consistently move tasks to low-priority queues, we can confirm that this reward can optimized to zero. The termination of the training is determined by the convergence of the reward to zero, which indicates that the scheduler has learned the optimal policy to prevent event loss. In this way, it learns the relationship between queue budgets and provenance capacity. Importantly, r_p is designed to be independent of the current state of the system as it is being scheduled. Instead, it directly reflects result of provenance collection in the single next cycle. It is irreversible to total rewards gained. The independence of r_p leads to a strong training behavior that the network prefers to minimize the event loss to maximize the overall Q value.

Optimizing throughput The performance goal of Aegis is to maximize system throughput by optimizing CPU utilization. Let C_t be the idleness of current dispatch cycle and C_{t+1} be the idleness of next dispatch cycle of the same task, where idleness refers to the number of slices taken by idle tasks before the task. The utilization reward r_p is defined as:

$$r_p = \frac{C_t}{C_t + C_{t+1} + 1} \tag{11}$$

This reward encourages the scheduler to minimize idleness in the next dispatch cycle by reducing C_{t+1}.

As discussed above, if r_p is not zero, the overall reward encourages the network to minimize event loss first. Otherwise, it optimizes for performance, leading to a hierarchical adaptation.

4.2.2 Reducing the overhead.

To reduce the cost of neural network inference, we introduce a δ-function to reduce unnecessary scheduling decisions given that Aegis's network inferences could be more computationally expensive than existing schedulers. With this optimization, the scheduler first compares the current task context f_t with its previous one f_{t-1} and then, only performs the neural network inference if the context has changed significantly, measured by a threshold δ:

$$|1 - \frac{f_{t-1}}{f_t}| < \delta, \ \forall f \in \{\texttt{nvcsw}, E\} \tag{12}$$

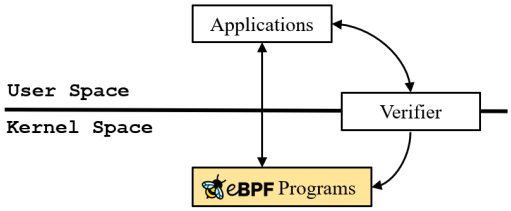


Figure 4. eBPF programs are subject to verification before being loaded into the kernel.

When comparing the context similarity, we consider two key factors: the average number of voluntary context switches (`nvcsw`) and the average number of events (E). A dramatic change in these metrics indicates workload shifts that require new scheduling actions.

5 Implementation Details

We implement Aegis's design into kernel-space and user-space components, as shown in Figure 5.

Kernel-space implementation We build the kernel space scheduler on the Extended Berkeley Packet Filter (eBPF) subsystem, which provides a forward-compatible and dynamic mechanism for attaching user programs to kernel objects. By compiling high-level code into eBPF bytecode and subjecting it to kernel verification, as depicted in Figure 4, eBPF ensures that loaded programs do not compromise system stability. The verifier rejects any operation deemed unsafe. Specifically, Aegis relies on `sched_ext` , an eBPF-driven scheduling framework introduced in the mainline Linux kernel since 6.13.

The main components in kernel space are data collection and neural network inference. The provenance features are collected from pinned eBPF maps, and the task features are collected from `task_struct` or from the `sched_ext` framework. These features are stored in eBPF `task_storage` maps. The map creates an eBPF dedicated storage inside `task_struct`. Adopting this map enables efficient data usage as stored features will be destroyed on task finish. The neural network inference is performed in kernel space, as user-space inference requires significant data transferring and unnecessary context switches. Unlike other approaches[34, 40, 45, 94] that compile the inference code with static weights included, Aegis's weight network is stored in an eBPF map and dynamically retrieved at runtime.

User-space implementation In the kernel space, Aegis collects task contexts in pairs as state transitions, denoted T. T consists of the task context of the same task in the current dispatch cycle S_t and the next dispatch cycle S_{t+1}, namely, $T = (S_t \rightarrow S_{t+1})$. These state transitions are passed to userspace via eBPF ring buffers. During preprocessing, we normalize the range of each feature to $[0, 128]$. Aegis adapts

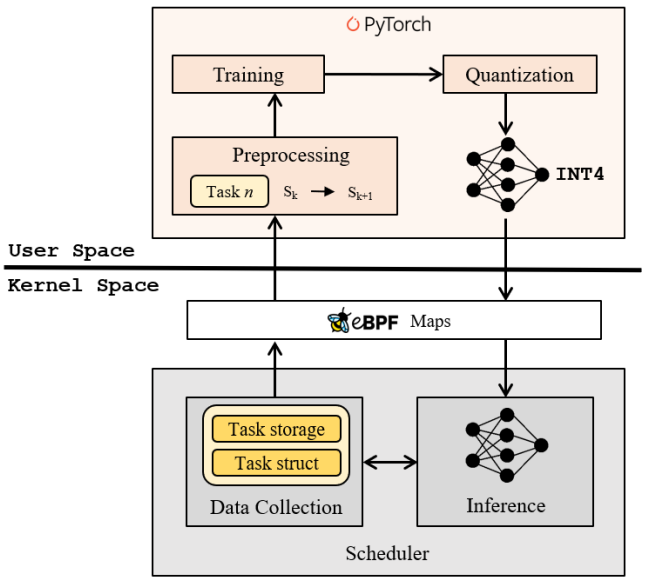


Figure 5. User-space and kernel-space Aegis implementation components.

to different feature distributions by leveraging different normalization methods. For uniformly distributed features, like `runtime`, Aegis multiply with a normalization term. Conversely, for features exhibiting long-tail distributions, like `event`, Aegis takes a log with base 2 prior to normalization. A userspace FIFO replay memory is then employed to record these transitions T, ensuring that recent and relevant transitions are retained. Aegis utilizes PyTorch[77] for training and quantization. During the training phase, a fixed number of transitions are randomly sampled from the replay memory to constitute the training batch, which helps in preventing overfitting and promotes generalization of the scheduler's decision-making process. After every training epoch, the model is quantized to `INT4` to acquire integer-only models and then passed to eBPF maps. By offloading quantization, preprocessing and training into userspace with mature framework like PyTorch, Aegis 's core architecture remains stable and widely applicable.

6 Evaluation

We empirically evaluate Aegis by answering the following research questions (RQs), with **[G3]** achieved by design with a theoretical evaluation in Section 4.1:

- **RQ1:** How well does Aegis prevent event loss? **[G1]**
- **RQ2:** What is the runtime performance of Aegis? **[G4]**
- **RQ3:** Is runtime bounded in the worst case? **[G2]**
- **RQ4:** What is the scheduling cost of Aegis? **[G5]**
- **RQ5:** How do different settings affect Aegis?

Table 6. Benchmarks for evaluating Aegis. Syscall speeds are average values under Sysdig and Aegis. Generic I/O syscalls include `openat`, `read`, `write`, `writev` and `close`. Network-specific syscalls include `recvmsg`, `recvfrom`, `accept4`, `socket` and `connect`.

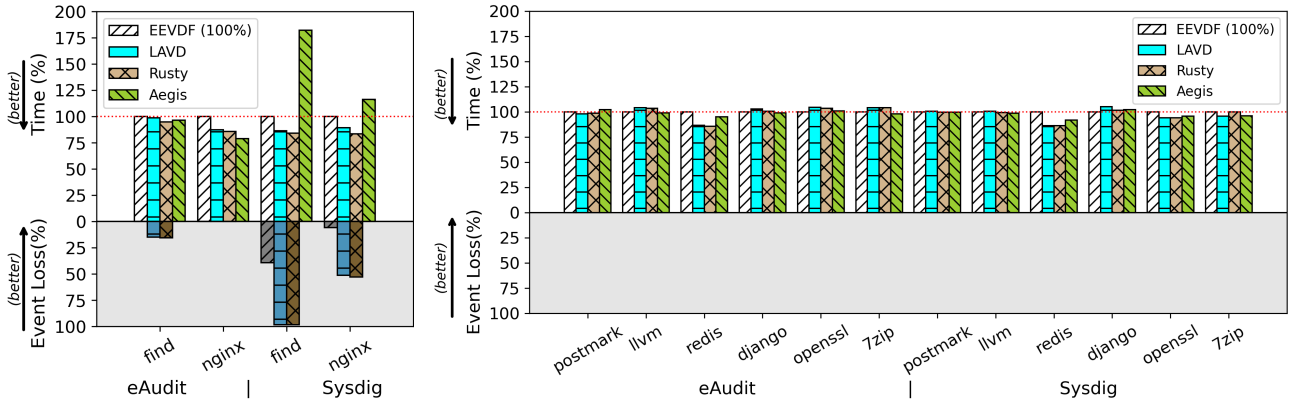
Benchmarks	Concurrency	Generic I/O (syscalls/s)	Network (syscalls/s)
7Zip[2]	Multi-core	/	/
OpenSSL[10]	Multi-core	/	/
Postmark[57]	Single-core	316,657	/
LLVM[8]	Multi-core	14,892	/
find[81]	Multi-core	1,098,652	/
Django[60]	Single-core	4,257	71
Redis[11]	Multi-core	298,124	122
Nginx[18]	Multi-core	915,771	304,856

6.1 Evaluation Setup

Our prototype scheduler and its userspace daemon are written in C and compiled using LLVM 18 with optimization set to `-O2`. Training and quantization use Python 3.10 and PyTorch 2.4.0. Communication between the scheduler and the user-space daemon is facilitated through sockets. We conduct experiments on an Ubuntu 24.04 QEMU virtual machine with an 8-core Intel 12900H processor, 32GB of DDR5-4800 memory, and an 1TB SSD. The system runs with a Linux 6.10 kernel with `sched_ext` support.

Provenance systems To demonstrate the generality of Aegis, we evaluate Aegis using two open-source provenance systems, **Sysdig**[14] and **eAudit**[81]. Specifically, we enable the modern_bpf probe for Sysdig. The evaluation with Sysdig highlights that Aegis effectively prevents significant event loss compared to all available schedulers, while the experiments with eAudit demonstrate Aegis's enhanced efficiency even in scenarios where no event loss occurs in some of the existing schedulers. We utilize the same system call settings as eAudit, allowing the auditing of a total of 80 system calls. We do not evaluate NoDrop[55] because its support is limited to legacy Linux kernels (v4.15) and is incompatible with modern 6.x kernels.

Schedulers We compare Aegis with three state-of-the-art schedulers. Aegis operates on the `sched_ext` framework, which requires Linux kernel version 6.8 or later. Thus, we use **EEVDF**[5] scheduler as baseline, which is Linux's default scheduler after kernel version 6.6. (We do not use CFS since one cannot revert to it with a recent kernel.) We also compare with **Rusty**[13] and **LAVD**[12], the most popular schedulers from `sched_ext` collections. Rusty is designed to be flexible, accommodating different architectures and workloads[13]. LAVD aims to provide better experience in latency-critical scenarios such as gaming.



(a) With over 1M syscalls/s, Aegis does not lose events in cases where others do.

(b) With less than 1M syscalls/s, Aegis performs no worse (i.e., offers comparable performance) compared to existing approaches.

Figure 6. Performance of Aegis on macrobenchmarks compared to EEVDF (baseline of 100%). The lower the better.

Benchmarks To evaluate the performance and resilience of Aegis, we select a diverse set of macrobenchmarks that utilize different kinds of system resources, performing CPU, I/O, and network activities. The benchmarks are summarized in Table 6 along with their system call rates, which we use as a proxy for provenance workload intensity. Among all the tests, **7Zip** and **OpenSSL** primarily stress the CPU, performing intensive compression / decompression algorithms and cryptographic operations. **Postmark** generates a large number of file operations. **LLVM** test compiles the LLVM project from source and **find** test spawns lots of processes to scan the `/usr/` directory simultaneously. These workloads evaluate the performance of Aegis in terms of computational and I/O resource allocation. Representative server benchmarks like **Django**, **Redis** and **Nginx** are introduced in addition to above tests. These tests incur heavy network activities in addition to disk and CPU workloads. We use the same settings as NoDrop[55] in 7Zip, OpenSSL, Django, Redis and Nginx tests, and the same settings as eAudit[81] in Postmark and find tests. Except pure computational tests like 7Zip and OpenSSL, all tests incur a considerable number of provenance events, stressing Aegis and provenance system at the same time.

Additionally, we introduce the **lmbench**[70] microbenchmark suite to measure low-level system performance. Lmbench includes a variety of microsecond-level tests, evaluating operations such as file system interactions, process forking, and execution. We report the event loss ratio and the runtime of each benchmark under different schedulers and provenance systems. All results reported are average of 10 attempts.

Queues We configure 4 queues: 1 primary queue and 3 additional queues. The provenance consumer, which receives and processes events, is pinned to the primary queue to facilitate training process. The threshold $\hat{t}_\infty$ is set to $5e^5$ns, determined through manual testing with Sysdig.

Neural networks For each provenance system, we train an independent neural network, which is a two-layer fully connected model activated by ReLU [73]. The hidden layer consists of 256 neurons, with an input size of 11 and an output size of 4, resulting in a total size of 11×256×4. Key hyperparameters include a discount factor $\gamma = 0.9$, a soft update coefficient $\tau = 0.005$, and a learning rate of $1e^{-4}$.

Training During the training process, Aegis continuously collects statistics of various workloads that include daily remote development, coding with VS Code and web browsing with Firefox, supplemented with periodically (e.g., every 3 mins) controlled stress using a lightweight variant of the find test with limited concurrency to prevent overfitting. The model will not see test workloads in Table 6. Once the model converges, the weights are frozen for testing.

6.2 RQ1: Efficacy of Aegis

A critical aspect of Aegis is its effectiveness in preventing provenance event loss. The grey area of Figure 6 summarizes the event loss statistics for the schedulers. For all tests, Aegis demonstrated zero event loss for both Sysdig and eAudit, showcasing its ability to handle high-frequency event streams reliably.

In contrast, in find test, EEVDF incurred a significant event loss rate of 39.34% with Sysdig, while maintaining no event loss with eAudit. LAVD and Rusty both exhibited severe event loss. LAVD reports event loss rates of 98.53% for Sysdig and 15.11% for eAudit, whereas eAudit is designed to achieve the completeness of events. Rusty shows nearly identical results. These results indicate that existing schedulers fail to prioritize provenance consumers sufficiently, underscoring Aegis's robust handling of diverse workloads by comparison. For the Nginx test, EEVDF reports a loss rate of 5.9% for

Sysdig. LAVD and Rusty also fails to provide the completeness of the events, where LAVD reports loss rate of 51.42% and Rusty reports 52.93% respectively, with Sysdig. However, Aegis achieved zero event loss, reaffirming its ability to sustain complete provenance data integrity in various conditions. All schedulers successfully avoided event loss for both Sysdig and eAudit for other benchmarks, which is expected since they are relatively lightweight compared to Nginx and find test. Overall, Aegis consistently achieves zero event loss across diverse benchmarks and provenance systems, outperforming state-of-the-art schedulers like EEVDF, LAVD, and Rusty.

Takeaways: Aegis ensures zero event loss across all test scenarios, even when other schedulers lose > 98% events.

6.3 RQ2: Performance of Aegis

We consider both Aegis's macrobenchmark and microbenchmark performance.

Macrobenchmarks As shown in Figure 6, Aegis has the best performance among 10 out of 16 tests. We observe that find and Nginx tests are the most provenance-demanding tests. In the two tests, although Rusty and LAVD are faster than Aegis in three cases, they fails to completely capture all provenance events. With eAudit, Aegis is faster than EEVDF, LAVD and Rusty in terms of average performance. With Sysdig, Aegis has average overhead of 49.24% compared to EEVDF, 70.96% compared to LAVD and 78.06% compared to Rusty. The overhead is high due to the fact that the event loss ratio is high (e.g., 39.34%, 98.53% and 98.54%, respectively).

For Django and LLVM tests, Aegis is on average 0.42%, 3.46% and 1.53% faster than EEVDF, LAVD and Rusty. This showcases efficient resource allocation of Aegis when handling daily workloads. In Postmark, Aegis is 0.86%, 1.55%, 1.76% slower, respectively. Such overhead is mainly caused by the scheduling cost and does not pose a significant concern in real scenarios. For Redis, Aegis is 6.63% faster than EEVDF but 8.07% and 8.90% slower than LAVD and Rusty. This is because Aegis is designed for general provenance scenarios rather than being a latency-critical scheduler like LAVD and Rusty. For computational workload including 7Zip and OpenSSL, Aegis dominates other schedulers, achieving 2.42%, 2.62% and 1.82% faster than the other three schedulers. This further corroborates Aegis for better task behavior recognition and resource utilization compared to state-of-the-art schedulers. In conclusion, all tests under Aegis finish in a reasonable time without provenance event loss.

Takeaways: Overall, Aegis outperforms EEVDF, LAVD, and Rusty by 3.42%, 0.90% and 0.25%, respectively, in macro workloads while incurring no event loss

Microbenchmarks We divide lmbench into four categories: system call, process, file system (FS) and inter-process communication (IPC). System call latency tests consist of NULL call, NULL I/O, stat, open/close file and signal install/handle. Process tests include fork, execve and shell tests. File system tests include file create and delete with size of 0k and 10k. IPC tests include AF_UNIX and pipe latency tests.

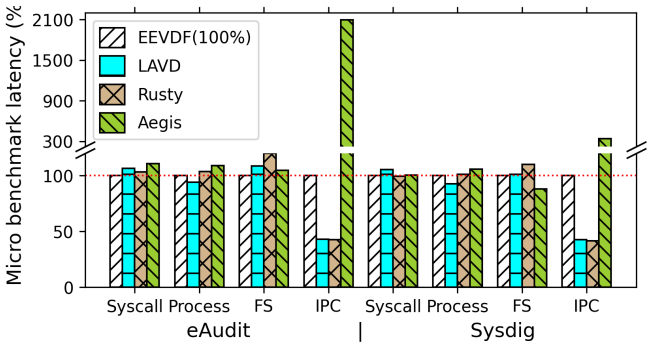


Figure 7. Average latency of microbenchmark tests compared to EEVDF (baseline of 100%).

The results in Figure 7 show that, compared to EEVDF and other schedulers, Aegis does not incur noticeable overheads in system call, process and file system tests. That is expected since many micro operations happen in one slice, and thus the performance is not affected by the scheduler. Aegis shows higher overhead for IPC; compared to the default EEVDF scheduler, Aegis has 2.58× / 2.17× overhead with Sysdig on AF_UNIX / pipe tests, respectively. For eAudit, the numbers are 20.12× / 19.79×. The reason behind this is Aegis's design using FIFO queues: when multiple tasks arrive at the same time, especially when related tasks (such as parent and child) are classified into different queues, the latency becomes high.

Takeaways: Aegis performs similarly to other schedulers for most microbenchmark tests. While Aegis introduces high latency in pipes and AF_UNIX, the latency is still under 0.3ms and does not pose significant concern to macro workloads, which Aegis aims to improve.

6.4 RQ3: Worst Case Analysis

To evaluate the fairness and starvation resistance of Aegis, we conduct a worst-case analysis. In the worst case all higher priority queues are occupied and ready for dispatch anytime as discussed in Section 4.1.1. We demonstrate Aegis's robustness with six configurations, as shown in Table 7. From **E1** to **E3**, we introduce 2 to 4 non-primary queues where the lowest priority queues share the same waiting time of $5e5$. From **E4** to **E6**, 3 non-primary queues are utilized and the waiting times of the lowest priority queue differ from $4e4$ to $7e7$. We introduce a pure computational task implemented as tight loops, and we intentionally saturate all higher-priority ones with 10 concurrent tasks. In the lowest priority queue we adjust the number of tasks to observe the finish time.

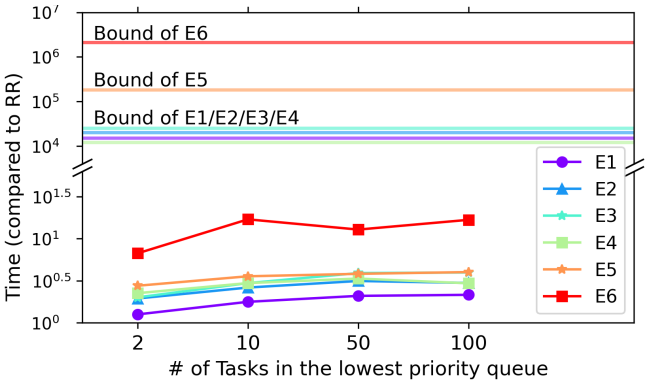


Figure 8. Performance of Aegis in the worst case as discussed in Section 4.1.1

Table 7. Setting of tasks and waiting times for the non primary queues for worst case analysis.

Setting	# of Tasks				Waiting Time			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
E1	10	2-100	/	/	1e2	5e5	/	/
E2	10	10	2-100	/	1e2	7e3	5e5	/
E3	10	10	10	2-100	1e2	2e3	3e4	5e5
E4	10	10	2-100	/	1e2	2e3	4e4	/
E5	10	10	2-100	/	1e2	3e4	6e6	/
E6	10	10	2-100	/	1e2	8e4	7e7	/
RR	5-104 Tasks							

Table 8. Scheduling cost of Aegis.

Metric	Min	Max	**Average**
Inference Cost (%)	0.09	2.75	**1.18**
Inference Frequency Per Core (Hz)	17.17	528.00	**294.77**
Inference Avg. Time (μs)	37.33	60.31	**50.16**
Maintenance Cost (%)	0.017	0.162	**0.08**
Delta Func Reduction (%)	0.04	0.85	**0.36**
Delta Func Effect Ratio (%)	11.50	98.46	**46.92**
Overall Cost (%)	0.01	2.44	**0.82**

The results are shown in Figure 8. Our results confirm that the lowest-priority task is consistently scheduled within the predicted bound. For **E1** to **E3**, the time compared to RR scheduler are 1.25×-2.15×, 1.94×-3.13× and 2.01× to 3.96×. The results indicate that the task in the lowest-priority queue maintains predictable execution proportional to the number of queues configured, validating the fairness guarantee enforced by Aegis. For **E4** to **E6**, the results are 2.24×-2.94×, 2.75×-4.01×, 6.69×-16.75×. We observe that with longer waiting time configured, while the lowest priority queue gets a longer runtime, it is still bounded by Aegis's queue-based mechanism. Overall, the results indicate that in the worst case no starvation or indefinite postponement was observed.

Takeaways: Aegis preserves queue-based fairness and bounded runtime for all tasks, even under intentionally adversarial settings.

6.5 RQ4: Scheduling Cost of Aegis

The scheduling cost breaks down into *inference cost* (percentage of time consumed by pre-processing and neural network inference for scheduling decisions) and *maintenance cost* (collecting and updating features of each task and selecting the appropriate queue). Table 8 shows the results.

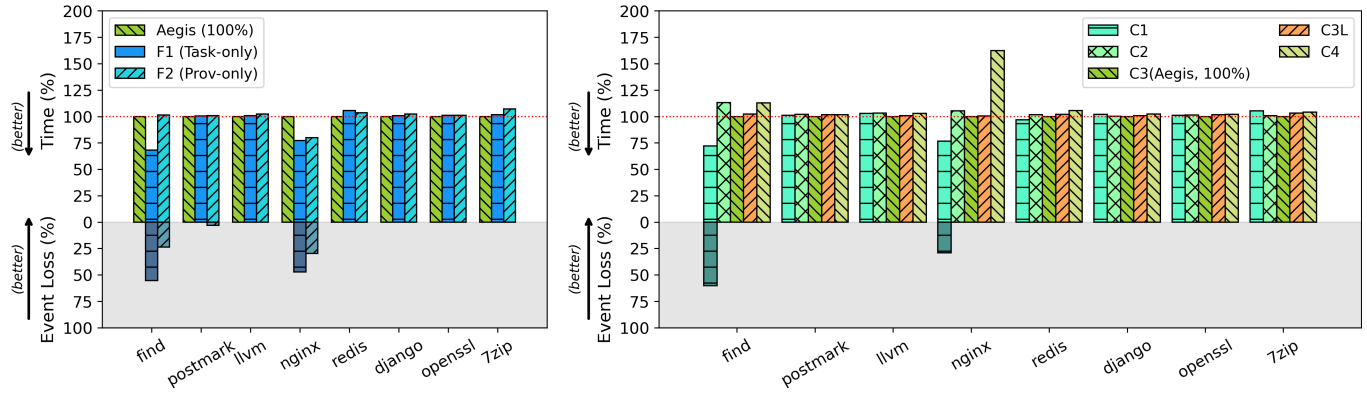
Inference cost Across the benchmarks, Aegis exhibits low inference costs. The most demanding benchmarks, the Nginx test, has the highest inference cost of 2.44%, and the find test has the highest inference frequency of 528 Hz. Workloads associated with multithreads along with I/O heavy in Aegis often incur with around 500Hz inference frequency per core such as find, LLVM, Nginx, and Redis. This is because these processes do not fully consume their slices and incur heavy scheduling decision makings. Table 8 also indicates computational workloads require <50 Hz of decision making. This is because they do not need to yield to other tasks.

At the same time, the results show that the average inference time is 50.16 μs. The lowest time, 37.33 μs appears in lmbench, and the highest 60.31 μs appears in 7Zip. This is because lmbench does not involve a lot of memory operations, and consequently the cache stays hot for the weight, reducing the overall weight accessing time. As a result, the inference cost is relatively lower.

Takeaways: The overall inference cost of Aegis is low, under 0.5% for typical scenarios and around 2% for most demanding scenarios.

Maintenance cost The maintenance cost, representing the time spent updating task contexts and features, remains negligible across all benchmarks, ranging from 0.007% in lmbench to 0.110% in Nginx. With eBPF and `sched_ext`, Aegis acquires the features and contexts of current task directly from `task_struct` and eBPF helper functions. Such design benefits Aegis's task context management cost, even under high-frequency decision-making scenarios.

Delta function The delta function in Aegis determines whether a task requires re-scheduling by evaluating changes in its context. We set $\delta = 0.25$, which is equivalent to right shifting by 2 digits for verification and efficiency need. As shown in Table 8, the delta function is particularly high for Postmark, OpenSSL, 7Zip and lmbench, saving 49.95%, 98.46%, 72.33% and 68.37% scheduling decisions. This is because these stable long-time workloads show similar behavior across the whole time, allowing the delta function to bypass inference frequently. In contrast, workloads like



(a) Performance using different features.

(b) Performance using different queue settings.

Figure 9. Performance of Aegis under different settings.

LLVM (33.93%) and Nginx (11.50%) see a lower effective ratio due to higher task context variability. The find test shows a low effective ratio of 23.12% due to the fact that it consists of many short-term tasks, reducing the chance of delta function taking place. In terms of runtime, the delta function is capable of reducing the overall runtime by 0.61% for most stressing workload and 0.36% on average among all the benchmarks. The results demonstrates the effectiveness of delta function in reducing the overall computational cost of Aegis, without sacrificing the performance, as discussed in Section 6.3.

6.6 RQ5: Ablation Study

To evaluate the contributions of various design choices in Aegis, we conduct an ablation study using Sysdig as the provenance system and $\hat{t}_\infty = 5e^5$ across two scenarios: feature configurations and queue configurations.

Feature settings We investigate the effect of using different feature sets to train the neural network. We evaluate using task-only features (**F1**) and using provenance-only features (**F2**) from Table 4 to train the neural network, and we compare the results with all features (**Aegis**).

Figure 9 shows the results. The runtime performance on each setting is shown in positive bars and the provenance performance is shown in negative bars in grey area. With **F1**, the system converges during training and is capable of preventing event loss under training workloads (i.e., 50 concurrent find workers). However, the scheduler is not robust enough and fails to work under unseen workloads (e.g., 288 workers). As a result, the find test **F1** is 31.81% and the Nginx test is 22.91% faster than that of using all of the features, but it fails to prevent event loss. **F2** leads to noticeable lower runtime (20.19%) in Nginx test but fails to adapt to even the training workloads, which leads to 23.63%, 3.29% and 29.74% event loss in find, Postmark and Nginx tests. The results indicate that Aegis's functionality relies on both kinds of features.

Queue settings We explore the impact of different queue configurations on system performance. We vary the number of queues and the waiting times allowed for tasks in each queue. The number of queues determines the granularity of task budget allocation, with fewer queues leading to broader groupings and potentially higher contention for resources, while more queues provide finer control but incur more difficult training and greater management overhead. We set up 4 variants for the cases of introducing 1 to 4 non-primary queues and denote them from **C1** to **C4**, whereas **C3** is the setting Aegis uses. Waiting times settings can also affect performance, so we use exponential (Aegis) and linear (**C3L**) settings.

Figure 9 shows the results. We observe that using only one non-primary queue (**C1**) converges the network during training and leads to less runtime in the find and Nginx test, but it loses up to 60.29% events during testing, which is similar to that of **F1**. When introducing two or four non-primary queues (**C2** or **C4**), we find that the trained schedulers are capable of preventing event loss but lead to additional runtime. Specifically, **C2**/**C4** are 13.04%/12.68% slower in find test, 5.37%/62.37% slower in Nginx test and are overall 3.40%/11.73% slower than **Aegis**. Thus, three non-primary queues is the best configuration since **C2** gives a coarse-grained setting while the situation in **C4** is too complex for simple networks. Fine-grained control is more difficult to learn in either of these settings. For linear waiting time **C3L**, we observe a slower but similar performance. On average, **C3L** incurs 1.60% extra runtime across three tests, indicating that exponential waiting time leads to easier learning patterns.

Takeaways: Configuring Aegis with exponential waiting times and 4 queues yields optimal performance.

7 Related Work and Discussion

Schedulers The `O(1)` scheduler[19] improves upon the original Linux scheduler's fairness and responsiveness. CFS[3] achieves finer granularity and better fairness. After Linux v6.6, CFS is replaced with EEVDF[5] for finer deadline-based fairness. Similar schedulers [13, 85] provide generic efficiency and fairness among all tasks. Domain-specific schedulers have been proposed for gaming[12], energy[61, 62] and micro operations[64, 69]. Recently, scheduler frameworks[6, 49, 54, 56, 68] enable user-defined scheduling that accommodates different needs; Aegis adopts `sched_ext` [6].

Provenance Early provenance systems rely on instrumenting the OS kernel, such as CamFlow[76], HiFi[78], LPM[31] and Kcal[65]. Recent systems are mainly built with eBPF, including Sysdig[14], Tracee[17], Tetragon[16], ProvBPF[63] and eAudit[81]. MPI[66], ProTracer[67] and ALchemist[92] implement high-level semantical provenance. However, the provenance completeness is less studied. eAudit[81] introduces a novel buffer design and NoDrop[55] introduces resource isolation, though these systems do not ensure all desired properties as discussed in Section 2.1. While Aegis addresses the critical challenges of efficient provenance-aware scheduling to prevent event loss and maintain system performance, a broader notion of provenance integrity requires complementary measures to ensure the overall trustworthiness of provenance systems, such as incorporating orthogonal threat detection for super producers and tamper-proofing techniques[42, 44, 47, 75, 82, 93].

Kernel-space machine learning Machine learning applications in kernel-space are bringing adaptive algorithms to scheduling[34], storage optimization[22, 45], networking applications[21, 41, 46, 94], intrusion detection[25, 28] and compartmentalization[87]. Prior work also considers kernel-space machine learning frameworks using floating points[22], integer quantization[94] and API remoting[40]. The most relevant work to Aegis is MLLB[34], which improves the `can_migrate_task()` function but does not implement a complete scheduler. Aegis uses a simple fully connected neural network to address the super producer issue. Employing larger neural network models or better training strategies could potentially allow Aegis to capture deeper patterns in workload behaviors, enabling more precise scheduling decisions. Additionally, hardware acceleration [40] with GPUs or NPUs, can significantly reduce inference costs, even enabling real-time high-frequency inference for large models. We leave these optimizations as possible future research directions.

8 Conclusion

We introduced Aegis, a kernel-space learned provenance scheduler that ensures that provenance events are completely captured to prevent super producer threats from hiding malicious system events and ensure the security guarantees of provenance systems embedded within reference monitors. We showed that existing solutions do not adequately defend against the threat because they either suffer from buffer overflows or fail to accommodate important functionalities. We leveraged reinforcement learning within the Linux kernel's scheduler to learn the behavior of provenance tasks and to optimize resource allocation to ensure completeness while being performant against other system tasks and events. We evaluated Aegis's efficacy using Sysdig and eAudit to study the runtime performance and scheduling cost and found that Aegis is capable of ensuring no event drops and ultimately ensuring that the provenance system meets the reference monitor's security goals.

References

[1] 6.8.7.arch1-1 bug: scheduling while atomic; watchdog: Bug: soft lockup / kernel & hardware / arch linux forums. https://bbs.archlinux.org/viewtopic.php?id=295345/. Accessed: 2025-7-25.

[2] 7-zip. https://www.7-zip.org//. Accessed: 2025-5-10.

[3] Cfs scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. Accessed: 2024-12-17.

[4] ebpf verifier - the linux kernel documentation. https://docs.kernel.org/bpf/verifier.html. Accessed: 2023-11-06.

[5] Eevdf scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html. Accessed: 2024-12-17.

[6] Extensible scheduler class. https://www.kernel.org/doc/html/next/scheduler/sched-ext.html. Accessed: 2024-12-17.

[7] kernel panic: imx-sdma causes scheduling while atomic - nxp community. https://community.nxp.com/t5/i-MX-Processors/kernel-panic-imx-sdma-causes-scheduling-while-atomic/m-p/1950884. Accessed: 2025-7-25.

[8] llvm/llvm-project: The llvm project is a collection of modular and reusable compiler and toolchain technologies. https://github.com/llvm/llvm-project//. Accessed: 2025-5-10.

[9] Multilevel queue - wikipedia. https://en.wikipedia.org/wiki/Multilevel_queue. Accessed: 2025-7-25.

[10] Openssl. https://www.openssl.org/. Accessed: 2025-5-10.

[11] Redis - the real-time data platform. https://redis.io. Accessed: 2025-5-10.

[12] scx_lavd scheduler. https://github.com/sched-ext/scx/blob/main/scheds/rust/scx_lavd. Accessed: 2024-12-11.

[13] scx_rusty scheduler. https://github.com/sched-ext/scx/blob/main/scheds/rust/scx_rusty. Accessed: 2024-12-11.

[14] Sysdig | security for containers, kubernetes, and cloud. https://sysdig.com/. Accessed: 2023-10-30.

[15] System panics with "bug: scheduling while atomic" at 'task_rq_lock+0x4d/0xa0' with xfs file system(s) mounted in rhel 6 - red hat customer portal. https://access.redhat.com/solutions/775673. Accessed: 2025-7-25.

[16] Tetragon - ebpf-based security observability and runtime enforcement. https://tetragon.io/. Accessed: 2024-10-30.

[17] Tracee - aqua security. https://www.aquasec.com/products/tracee/. Accessed: 2024-10-30.

[18] Welcome to f5 nginx. https://www.nginx.com. Accessed: 2025-5-10.

[19] Introducing the 2.6 kernel | linux journal. https://www.linuxjournal.com/article/6530, 2003. Accessed: 2024-12-17.

[20] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1791–1807. IEEE, 2022.

[21] Ibrahim Akgun, Santiago Vargas, Michael Arkhangelskiy, Andrew Burford, Michael McNeill, Aruna Balasubramanian, Anshul Gandhi,

and Erez Zadok. Predicting network buffer capacity for bbr fairness. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on ML for Systems*, 2022.

[22] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. Kmlib: Towards machine learning for operating systems. In *Proceedings of the On-Device Intelligence Workshop, co-located with the MLSys Conference*, pages 1–6, 2020.

[23] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. {ATLAS}: A sequence-based learning approach for attack investigation. In *30th USENIX security symposium (USENIX security 21)*, pages 3005–3022, 2021.

[24] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Efficient provenance storage over nested data collections. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 958–969, 2009.

[25] NEMALIKANTI ANAND, MA SAIFULLA, and Pavan Kumar Aakula. High-performance intrusion detection systemusing ebpf with machine learning algorithms. 2023.

[26] James P Anderson. Computer Security Technology Planning Study ESD-TR-73-51. *USAF Electronics Systems Division*, 1972.

[27] Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. Using reenactment to retroactively capture provenance for transactions. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):599–612, 2017.

[28] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A flow-based ids using machine learning in ebpf. *arXiv preprint arXiv:2102.09980*, 2021.

[29] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[30] Adam Bates, Devin J Pohly, and Kevin RB Butler. Secure and trustworthy provenance collection for digital forensics. *Digital Fingerprinting*, pages 141–176, 2016.

[31] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy {Whole-System} provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, 2015.

[32] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006, 2008.

[33] Ang Chen. *Secure Diagnostics and Forensics with Network Provenance*. University of Pennsylvania, 2017.

[34] Jingde Chen, Subho S Banerjee, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Machine learning for load balancing in the linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 67–74, 2020.

[35] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical intrusion detection and investigation using whole-system provenance. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3533–3551. IEEE, 2024.

[36] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 335–344, New York, NY, USA, 1962. Association for Computing Machinery.

[37] Hailun Ding, Juan Zhai, Dong Deng, and Shiqing Ma. The case for learned provenance graph storage systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3277–3294, 2023.

[38] Hailun Ding, Juan Zhai, Yuhong Nan, and Shiqing Ma. {AIRTAG}: Towards automated attack investigation by unsupervised learning with log texts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 373–390, 2023.

[39] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. {SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2987–3004, 2021.

[40] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J Rossbach. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 846–861, 2023.

[41] Jorge Gallego-Madrid, Irene Bru-Santa, Alvaro Ruiz-Rodenas, Ramon Sanchez-Iborra, and Antonio Skarmeta. Machine learning-powered traffic processing in commodity hardware with ebpf. *Computer Networks*, page 110295, 2024.

[42] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 391–408, 2023.

[43] Steve Grubb. Linux audit. https://people.redhat.com/sgrubb/audit/. Accessed: 2024-10-30.

[44] Gorka Guardiola-Múzquiz and Enrique Soriano-Salvador. Sealfsv2: combining storage-based and ratcheting for tamper-evident logging. *International Journal of Information Security*, 22(2):447–466, 2023.

[45] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi 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)*, pages 173–190, 2020.

[46] Takanori Hara and Masahiro Sasabe. On practicality of kernel packet processing empowered by lightweight neural network and decision tree. In *2023 14th International Conference on Network of the Future (NoF)*, pages 89–97. IEEE, 2023.

[47] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via {Fixed-Key} blockcipher. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2389–2406, 2022.

[48] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. {Dependence-Preserving} data compaction for scalable forensic analysis. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1723–1740, 2018.

[49] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[50] IBM. Cost of a data breach 2024 | IBM — ibm.com. https://www.ibm.com/reports/data-breach. [Accessed 23-01-2025].

[51] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2620–2638. IEEE, 2023.

[52] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 377–390, 2017.

[53] Yang Ji, Sangho Lee, and Wenke Lee. Recprov: Towards provenance-aware user space record and replay. In *Provenance and Annotation of Data and Processes: 6th International Provenance and Annotation Workshop, IPAW 2016, McLean, VA, USA, June 7-8, 2016, Proceedings 6*, pages 3–15. Springer, 2016.

[54] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A general high-efficient scheduling framework in user space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 265–279, 2024.

[55] Peng Jiang, Ruizhe Huang, Ding Li, Yao Guo, Xiangqun Chen, Jianhai Luan, Yuxin Ren, and Xinwei Hu. Auditing frameworks need resource isolation: A systematic study on the super producer threat to system auditing and its mitigation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 355–372, 2023.

[56] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[57] Jeffrey Katcher. Postmark: A new file system benchmark. *TR3022*, 1997. https://www.filesystems.org/docs/auto-pilot/Postmark.html.

[58] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[59] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *Ndss*, 2005.

[60] Michael Larabel. yperformance benchmark. https://openbenchmarking.org/test/pts/pyperformance. Accessed: 2025-5-10.

[61] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. Os scheduling with nest: Keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 368–383, 2022.

[62] Qianlin Liang, Walid A Hanafy, Noman Bashir, David Irwin, and Prashant Shenoy. Energy time fairness: Balancing fair allocation of energy and time for gpu workloads. In *2023 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 53–66. IEEE, 2023.

[63] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure namespaced kernel audit for containers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 518–532, 2021.

[64] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 280–295, 2024.

[65] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. {Kernel-Supported}{Cost-Effective} audit logging for causality tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 241–254, 2018.

[66] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, 2017.

[67] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network And Distributed System Security Symposium (NDSS 2016)*. Internet Soc, 2016.

[68] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. Efficient scheduler live update for linux kernel with modularization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 194–207, 2023.

[69] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

[70] Larry W McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996. https://lmbench.sourceforge.net/.

[71] Volodymyr Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[72] Xin Mu, Ming Pang, and Feida Zhu. Data provenance via differential auditing. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[73] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[74] Shaiqa Nasreen and Ajaz Hussain Mir. Cloud forensics: A centralized cloud provenance investigation system using mecc. *Concurrency and Computation: Practice and Experience*, 36(6):e7949, 2024.

[75] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1551–1574, 2020.

[76] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017.

[77] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[78] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 259–268, 2012.

[79] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[80] Lukas Rupprecht, James C Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. Improving reproducibility of data science pipelines through transparent provenance capture. *Proceedings of the VLDB Endowment*, 13(12):3354–3368, 2020.

[81] R Sekar, Hanke Kimm, and Rohit Aich. eaudit: A fast, scalable and deployable audit data collection system. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3571–3589. IEEE, 2024.

[82] Enrique Soriano-Salvador and Gorka Guardiola-Múzquiz. Sealfs: Storage-based tamper-evident logging. *Computers & Security*, 108:102325, 2021.

[83] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Decoupling provenance capture and analysis from execution. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, 2015.

[84] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1324–1337, 2018.

[85] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. Karma: Resource allocation for dynamic demands. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 645–662, 2023.

[86] Chundong Wang, Lei Yang, Yijie Wu, Yuduo Wu, Xiaochun Cheng, Zhaohui Li, and Zheli Liu. Data provenance with retention of reference relations. *IEEE Access*, 6:77033–77042, 2018.

[87] Zicheng Wang, Tiejin Chen, Qinrun Dai, Yueqi Chen, Hua Wei, and Qingkai Zeng. When ebpf meets machine learning: On-the-fly os kernel compartmentalization. *arXiv preprint arXiv:2401.05641*, 2024.

[88] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE Long. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages

1752–1756, 2012.

[89] Yulai Xie, Dan Feng, Zhipeng Tan, and Junzhe Zhou. Unifying intrusion detection and forensic analysis via provenance awareness. *Future Generation Computer Systems*, 61:26–36, 2016.

[90] Kui Xu, Huijun Xiong, Chehai Wu, Deian Stefan, and Danfeng Yao. Data-provenance verification for secure hosts. *IEEE Transactions on Dependable and Secure Computing*, 9(2):173–183, 2011.

[91] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 504–516, 2016.

[92] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela F Ciocarlie, Vinod Yegneswaran, et al. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *NDSS*, 2021.

[93] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3898–3912, 2024.

[94] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 414–427, 2022.

[95] Xu Zhang, Zhaohui H Sun, Svebor Karaman, and Shih-Fu Chang. Discovering image manipulation history by pairwise relation and forensics tools. *IEEE Journal of Selected Topics in Signal Processing*, 14(5):1012–1023, 2020.

[96] Tiantian Zhu, Jiayu Wang, Linqi Ruan, Chunlin Xiong, Jinkai Yu, Yaosheng Li, Yan Chen, Mingqi Lv, and Tieming Chen. General, efficient, and real-time data compaction strategy for apt forensic analysis. *IEEE Transactions on Information Forensics and Security*, 16:3312–3325, 2021.