

DVAS: Dynamic Vulnerability-Aware Scheduling for HPC Clusters using Splunk-Driven Access Control

Abhimanyu Chawla
Abhimanyu.Chawla@colostate.edu
Colorado State University
Fort Collins, Colorado, USA

Indrakshi Ray
Indrakshi.Ray@colostate.edu
Colorado State University
Fort Collins, Colorado, USA

Abstract

High Performance Computing (HPC) environments are increasingly being considered as high-value target for attackers. Vulnerabilities identified in such environments cannot be immediately patched because of the need for high system availability and long-running workflows. If jobs execute on vulnerable nodes, the value of the output produced are suspect. Moreover, if vulnerable nodes are completely excluded from job allocation, the throughput of the system is affected. Towards this end, we propose DVAS, a dynamic vulnerability-aware scheduling mechanism that aims to maximize throughput taking into account the risk caused by vulnerabilities. DVAS correlates real-time vulnerability telemetry with user behavioral profiling to calculate a probabilistic risk score which is used as a dynamic attribute for controlling access to compute nodes in an HPC cluster. DVAS is implemented using our proposed Secure Job Admission (SJA) algorithm that uses Splunk analytics. We use an asynchronous architecture that decouples the Policy Decision Point (PDP) from the Policy Enforcement Point (PEP), achieving a mean enforcement latency of 0.88ms. Our evaluation on a 25-node cluster shows that DVAS reduces risk exposure by 98% while maintaining 91.8% utilization, significantly outperforming traditional "drain-and-patch" strategies. We also propose a measure called Security-Weighted Throughput (SWT) of a cluster that maps security into an operational cost and determines the throughput taking into account the untrustworthiness of results produced by execution on a node containing exploitable vulnerabilities.

CCS Concepts

• **Security and privacy** → **Systems security; Access control; Vulnerability management.**

Keywords

HPC Security; Splunk; SLURM; Vulnerability-Aware Access Control; Behavioral Profiling

ACM Reference Format:

Abhimanyu Chawla and Indrakshi Ray. 2026. DVAS: Dynamic Vulnerability-Aware Scheduling for HPC Clusters using Splunk-Driven Access Control. In *Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT '26)*, July 08–10, 2026, Waterloo, ON, Canada. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3750555.3811898>



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SACMAT '26, Waterloo, ON, Canada

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2107-6/2026/07

<https://doi.org/10.1145/3750555.3811898>

1 Introduction

High-Performance Computing (HPC) environments represent a unique challenge in the landscape of cybersecurity. Unlike enterprise IT, where a single compromised workstation can be isolated without halting the company, an HPC cluster is a massive, interconnected fabric of compute nodes, high-speed interconnects (InfiniBand), and shared parallel file systems (Lustre, GPFS). This architecture is increasingly recognized as a high-value target for lateral movement and data exfiltration, as outlined in the NIST Special Publication 800-223 [16]. A single vulnerability in a shared library can be leveraged for lateral movement, potentially exposing petabytes of sensitive research data or allowing for the exfiltration of intellectual property.

The current landscape of HPC security is characterized by the **Patching Paradox**: Security policies require system administrators to deploy critical patches (for instance, those addressing CVEs with CVSS scores above 9.0) without delay to preserve system integrity. In contrast, availability requirements dictate that researchers need long-duration jobs to run without interruption. Initiating a cluster-wide reboot can waste thousands of CPU/GPU hours and interfere with critical scientific schedules. This tension often leads to a deferred maintenance strategy, where the clusters remain vulnerable for weeks until a scheduled maintenance window. During this window, the attack surface is wide open, yet the actual risk is non-uniform.

Existing access control models, such as Attribute-Based Access Control (ABAC), allow for decisions based on environment attributes (e.g., "time of day"). However, they rarely account for internal system "health" as a dynamic real-time attribute. While frameworks like Next Generation Access Control (NGAC) [10] provide a standardized way to represent diverse access policies through linear graph models, there is a lack of a functional feedback loop between vulnerability management systems (which know what is broken) and workload orchestrators (which know who is running what).

To bridge this gap, we propose **Dynamic Vulnerability-Aware Scheduling (DVAS)**. Our hypothesis is that a system can achieve "virtual patching" by treating the vulnerability status of a node as a dynamic constraint in the scheduling logic. Rather than relying on the traditional, binary state of "all-in" or "all-out" node maintenance, our approach utilizes a state-aware access control model that functions at the job-admission level. By intercepting every submission via a specialized Policy Enforcement Point (PEP), the framework dynamically evaluates the risk of a job landing on a specific node based on the current vulnerability posture of that node's software stack. This allows system administrators to keep potentially vulnerable nodes online for low-risk tasks while strictly

cordoning them off from users or applications that are likely to invoke compromised libraries.

The core capability of this approach lies in the integration of a Secure Job Admission (SJA) algorithm with correlated vulnerability information and historical user behavior profiles. We used Splunk, an industry-standard big data analytics platform [20], to ingest, analyze, and correlate system logs. This enabled computing the risk matrix and user risk profiles in real time. By calculating risk as the maximum of a user's declared intent and their latent behavioral patterns, DVAS can preemptively identify silent dependencies and reroute jobs to secure resources without requiring manual intervention from the researcher. This creates an intrusion-tolerant scheduling environment in which system integrity is maintained as a continuous variable, optimized for both security and maximum scientific throughput.

The primary contribution of our work is the development and validation of DVAS, a workload-centric security framework for HPC. Specifically, this paper makes the following contributions:

- **Dynamic Enforcement Mechanism:** A novel architecture integrating Splunk-driven vulnerability telemetry with the SLURM Job Submit Lua API for real-time job mutation.
- **Security-Weighted Throughput (SWT):** A mathematical metric that quantifies the effective throughput of a cluster by penalizing execution on compromised nodes.
- **Behavioral Risk Profiling:** A predictive algorithm using historical audit logs to detect undeclared library dependencies.

The remainder of this paper is structured as follows. Section 2 surveys prior work on security in HPC and dynamic access control, and highlights the research gap that our approach addresses. Section 3 provides a detailed description of the DVAS components. Section 4 outlines the implementation strategy, including the required algorithms and the execution sequence. In Section 5, we introduce and explain our mathematical metric for Security-Weighted Throughput (SWT). Section 6 investigates a realistic case study illustrating how DVAS mitigates CVE-2021-45943. Section 7 reports the evaluation methodology and findings. Section 8 examines the limitations of DVAS, and Section 9 recaps our contributions and suggests directions for future research.

2 Related Work

The intersection of workload orchestration and cybersecurity has evolved from static resource allocation toward dynamic, risk-sensitive models. While scheduling research has advanced significantly in performance optimization and adaptive control, explicit integration of node-level security health into HPC scheduling decisions remains limited. We categorize the existing literature into three domains: (1) security-aware scheduling, (2) intrusion tolerance and resilience, and (3) access control frameworks.

Security-Aware Scheduling in HPC and Cloud

Early work on dynamic HPC scheduling focused primarily on performance metrics such as throughput, fairness, and utilization. With the emergence of multicore systems, contention-aware scheduling became prominent. Blagodurov et al. [5] demonstrated that cache and memory interference significantly impact application

performance, introducing hardware-counter-aware scheduling decisions. Subsequent work expanded this line of research to runtime interference mitigation and multi-resource optimization.

Recent advances employ reinforcement learning (RL) for adaptive scheduling. Mao et al. [12, 13] pioneered RL-based resource management in distributed clusters. More recent HPC-specific frameworks such as MARS [3], hierarchical RL schedulers [21], and transformer-enhanced scheduling [7] demonstrate improved adaptability and utilization across heterogeneous CPU/GPU clusters. These systems optimize job slowdown, resource balance, and throughput under dynamic workloads; however, they treat compute nodes as homogeneous in security posture.

Parallel efforts introduced security-aware scheduling concepts. Xie et al. [22] proposed a Security-Aware Scheduling (SAS) algorithm that incorporated encryption overhead into scheduling decisions, focusing primarily on computational cost rather than vulnerability state. In cloud-native infrastructures, Yuchi et al. [24] explored security-aware VM placement to reduce the cloud's overall security risk by considering the connections among VMs. While these approaches acknowledge security constraints, they abstract risk at the workload or VM level rather than modeling real-time security telemetry of physical HPC nodes.

Security-aware workflow scheduling in distributed systems further integrates trust metrics and risk constraints [11, 25]. Yet these models are primarily designed for virtualized cloud environments and do not address tightly coupled, long-running HPC workloads.

Intrusion Tolerance and Resilience

Research in Intrusion-Tolerant Systems (ITS) emphasizes resilience through redundancy, rejuvenation, and diversity. The Self-Cleansing Intrusion Tolerance (SCIT) architecture [1, 14, 17] periodically rotates servers through a cleansing cycle to eliminate persistent threats. Software diversity techniques and proactive recovery mechanisms similarly aim to reduce dwell time of adversaries.

While effective in web and cloud services, such rotation-based mechanisms are often incompatible with HPC workloads, where jobs may execute for hours or days and cannot tolerate frequent node resets. Intrusion tolerance research therefore addresses system recovery rather than risk-aware job placement. In contrast, approaches such as Dynamic Vulnerability-Aware Scheduling (DVAS) aim to implement "virtual rejuvenation" by avoiding high-risk execution paths rather than rebooting nodes, thereby preserving long-running computational continuity.

Access Control Frameworks

The transition from Role-Based Access Control (RBAC) [18] to Attribute-Based Access Control (ABAC) [9] introduced environmental attributes into authorization decisions. ABAC enables policies based on subject, object, and contextual attributes such as time, location, and classification level. The NIST Next Generation Access Control (NGAC) framework [10] formalized access control relationships as graph-based policy structures, supporting scalable and expressive authorization models.

Recent work has explored automated policy mining from access logs [23], aligning with behavioral telemetry analysis frameworks such as Splunk-driven profiling. However, while ABAC and NGAC

incorporate contextual state into authorization, they are not typically integrated into HPC job schedulers as optimization constraints. Node-level attributes such as vulnerability score, patch latency, integrity attestation, or anomaly probability are rarely treated as first-class scheduling variables.

Beyond traditional RBAC and ABAC, “risk-based” access control has emerged as a critical paradigm for managing insider threats in distributed environments. Atlam et al. [2] provide a comprehensive systematic review of the literature on risk-based access control, identifying key risk factors, such as user history and resource sensitivity, and evaluating various risk estimation techniques, including fuzzy logic and game theory to accommodate dynamic security requirements. This adaptive approach is particularly relevant in cloud computing, where traditional boundary-based security is insufficient. For example, Lakshmi et al. [8] proposed a risk-based model specifically for cloud services that calculates a user’s risk value against a predefined threshold to mitigate insider threats such as session hijacking and buffer overflow, ensuring that even in worst-case scenarios, a portion of the data remains secure through partial access denial. Dos Santos et al. [6] proposed a framework that extends the eXtensible Access Control Markup Language (XACML) to enforce risk-based policies, utilizing quantitative risk assessment to manage the inherent volatility of cloud resource access. These models underscore the need to move toward the continuous, quantitative risk assessment that DVAS implements for HPC job scheduling.

Research Gap

Across these domains, scheduling awareness has expanded from static allocation to performance-, contention-, and policy-aware adaptation. Reinforcement learning schedulers optimize multi-objective cost functions; cloud frameworks incorporate abstract risk constraints; intrusion tolerance emphasizes recovery mechanisms; and access control models formalize contextual authorization.

However, a fundamental gap remains: modern HPC schedulers do not integrate real-time security health or vulnerability posture of compute nodes into their placement decisions. Existing security-aware scheduling research abstracts security into workload-level classifications or isolation constraints rather than incorporating continuous security telemetry of resources. While existing models can restrict access to a “compromised” node, they lack the granularity to distinguish between a node that is vulnerable and a node that is actively being exploited. This leads to the “Safety-Throughput” deficit, where safe jobs are blocked by over-broad security policies. DVAS is the first model to utilize Splunk-driven log correlation to resolve this deficit in a bare-metal HPC context.

3 System Architecture

The DVAS architecture (Figure 1) is designed as a closed-loop system in which Splunk (the Policy Decision Point) aggregates three telemetry streams: unpatched vulnerability data, auditd/SLURM behavioral logs, and package inventories, to construct a real-time risk matrix. This risk matrix is serialized and asynchronously transferred to the SLURM controller node, where it is stored as a local policy cache (memory map). From here, this matrix is then utilized by a SLURM job_submit.lua script (the Policy Enforcement Point), which intercepts new submissions to exclude high-risk nodes from

a job’s allocation. Finally, job outcomes (scheduled or rejected) and associated metadata (e.g., node and module details) are returned to Splunk to refine the corresponding user behavior profile, thus closing the loop.

3.1 Data Ingestion & Normalization Layer

The foundation of the architecture is the data ingestion fabric. To provide the Policy Decision Point (PDP) with a ground-truth view of the cluster, we ingest three distinct data streams:

- (1) Vulnerability Feeds: Continuous JSON feeds from scanners (e.g., Nessus, OpenVAS) and the NVD database are ingested. We use a custom Splunk *SourceType* to normalize disparate scanner outputs into a unified Node_ID | CVE_ID | Library schema.
- (2) System Inventory Stream: This is a manifest of installed library versions across every compute node. Periodic rpm -qa (for RedHat based distributions) or dpkg -l (for Debian based distributions) logs and lmod inventory are forwarded from the compute nodes to Splunk. This allows the system to verify if a vulnerable package is actually installed and active on a specific node.
- (3) Behavioral Telemetry: Linux auditd and process accounting logs that track which binaries are actually being invoked by which users.

This layer performs canonical normalization, ensuring that vulnerability alerts are correctly mapped to specific libraries on specific physical nodes, creating the raw inputs for the Risk Matrix $\mathcal{R}$.

3.2 The Correlation Engine (The Policy Decision Point)

The PDP is implemented using Splunk, a high-performance big data analytics platform. In our architecture, Splunk serves as the central brain that correlates disparate security events into an actionable policy. Standard PDPs are often considered “stateless”, as they evaluate a request and forget it. Splunk provides a “stateful” PDP capable of longitudinal analysis (e.g., “User U has used this vulnerable library 5 times in the last hour”). Splunk executes periodic correlation searches that update the Global Risk Matrix ($\mathcal{R}$). It essentially performs the “Policy Computation” offline, so the results are ready the moment a user submits a job. It acts as the policy graph in NGAC terms, determining which user-resource edges should be temporarily severed due to risk. The PDP produces the Global Risk Matrix, a lightweight lookup table that maps compute nodes to “forbidden” software modules. This matrix is exported to the SLURM control node via a secure rsync or syslog trigger every 300 seconds.

The risk for a specific node n_i relative to a job J_k submitted by user U is defined as:

$$Risk(n_i, J_k, U) = \sum_{j \in \mathcal{M}} (\mathbb{I}(CVE_j \in n_i) \cdot \max [\mathbb{I}(m_j \in J_k), P_{U,j}])$$

Where:

- $\mathbb{I}(CVE_j \in n_i)$: An indicator function that returns 1 if node n_i contains a critical vulnerability in software module m_j , otherwise returns 0.

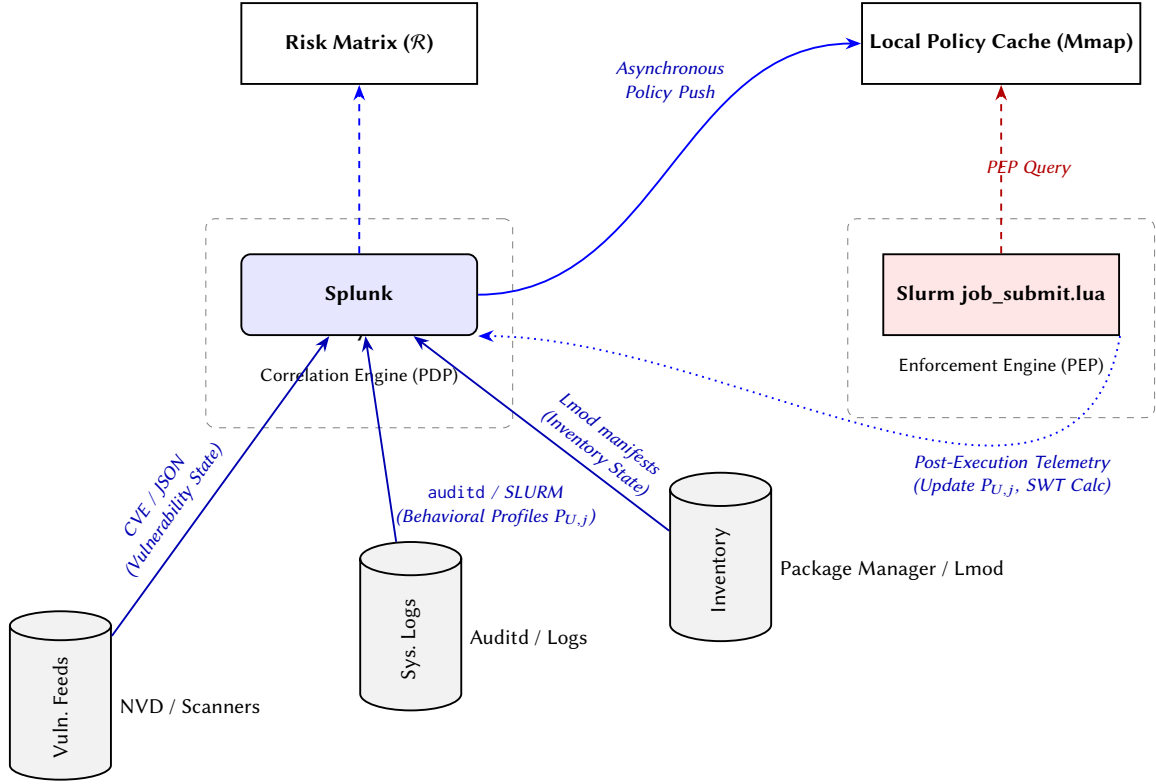


Figure 1: DVAS High-Level Architecture

- $\mathbb{I}(m_j \in J_k)$: An indicator function that returns 1 if the job J_k explicitly requests module m_j (the "Declared Risk"), otherwise returns 0.
- $P_{U,j}$: The behavioral probability ($0 \leq P_{U,j} \leq 1$) that user U will invoke module m_j , derived from auditd and job accounting logs in Splunk (the "Latent Risk").

The Correlation Engine (PDP) does not wait for a job submission to perform this calculation. Instead, it runs a continuous background correlation search in Splunk that pre-calculates the potential risk for every $(User, Node)$ pair. Splunk continuously ingests auditd logs. Every time a user executes a binary, the PDP updates the frequency count for the associated library. By using the $\max[\dots]$ function, the PDP adopts a conservative security Posture. If a user declares a library, the risk is 100%. If they do not declare it, but their behavioral history suggests an 80% likelihood of use, the PDP treats the node as having an 80% risk factor.

3.3 The Enforcement Layer (The Policy Enforcement Point)

The PEP is implemented via the SLURM Job Submit Lua API. Slurm is the industry-standard workload manager for HPC; its Lua plugin interface allows for the interception of every sbatch or srun command before it reaches the allocation engine [19].

The Lua script functions as a gatekeeper. When a job request arrives, the PEP parses the job's environment variables and performs a lookup against the local policy cache (provided by the PDP). Instead of a binary "Permit/Deny" decision common in standard

access control, our PEP performs constraint injection. It appends an `-exclude` list to the job description, effectively rerouting the access request to a secure subset of the requested resources. Additionally, utilizing the user profiling data from Splunk, if $Risk(n_i, J_k, U) > \tau$, the node is automatically added to the exclusion list, even if the library is not explicitly declared. This closed-loop system ensures that "silent dependencies" (static linking or manual binary execution) are mitigated before the job even lands on the compute fabric.

To maintain the required $<1\text{ms}$ latency, the PEP does not call Splunk directly. Instead, it reads a memory-mapped policy snapshot exported by Splunk, ensuring the enforcement is decoupled from the heavy-duty analytics engine.

3.4 Feedback Loop and Telemetry

The final component is the post-execution feedback loop. Once a job is scheduled or rerouted, the decision is logged back into Splunk. This closes the loop by:

- Updating the user's behavior profile ($\vec{P}_U$).
- Verifying that the rerouting successfully avoided the vulnerable nodes, for policy validation.
- Providing the data points necessary to calculate the Security-Weighted Throughput (SWT).

4 Implementation

The DVAS framework is implemented as a middleware layer that synchronizes the Vulnerability Intelligence Base (VIB) with the

cluster's resource allocation manager, which is SLURM in this case. The implementation moves beyond static access control by treating node security posture as a dynamic, consumable attribute.

4.1 The Global Risk Matrix ($\mathcal{R}$)

The core of the implementation is the generation of a dynamic Global Risk Matrix, denoted as $\mathcal{R}$. This matrix is maintained within the Splunk-driven PDP and represents the intersection of infrastructure assets and known vulnerabilities. Splunk aggregates raw telemetry to populate this matrix, where each entry $r_{i,j}$ represents the security posture of node n_i regarding software module m_j . For a cluster with N nodes and M critical software modules, $\mathcal{R}$ is an $N \times M$ binary matrix and an entry $r_{i,j}$ is defined as:

$$r_{i,j} = \begin{cases} 1, & \text{if } n_i \text{ contains a vulnerability (CVSS} \geq 8.0) \text{ in } m_j \\ 0, & \text{otherwise} \end{cases}$$

To generate the risk matrix, we utilize a correlation search in Splunk that joins vulnerability scan data with system inventory manifests and behavioral logs. This matrix is then serialized for the SLURM Policy Enforcement Point (PEP). Listing 1 presents a representative Splunk query that correlates three separate data sources: Nessus/OpenVAS (vulnerability), Lmod/Inventory (software location), and auditd (behavioral probability).

Listing 1: Splunk Correlation Query for Risk Matrix Serialization

```
index=hpc_security sourcetype=vuln_scanner
severity=critical
| stats values(cve) as cves by hostname,
software_pkg
| join type=left hostname, software_pkg [
search index=hpc_inventory sourcetype=
lmod_manifest
| stats values(module_path) as path by
hostname, software_pkg
]
| join type=left user, software_pkg [
search index=hpc_logs sourcetype=auditd
| stats count as usage_freq by user,
software_pkg
| eventstats max(usage_freq) as max_f
| eval p_user_j = usage_freq / max_f
]
| eval risk_val = if(isnotnull(cves), 1, 0)
| table hostname, software_pkg, risk_val, p_user_j
| outputlookup dvas_global_risk_matrix.csv
```

Table 1 illustrates a serialized snapshot of the matrix for a subset of nodes and users. In the actual implementation, this is exported as a bitmask or a high-speed JSON lookup for the SLURM Lua engine.

As defined in our risk equation, the total risk is the product of the CVE Status and the $\max(\text{Declared}, \text{Latent})$ risk. In the sample risk matrix, for *node-001* and *user_alpha*, even if the user does not declare *libgdal*, the PDP assigns a risk of 0.85. If the threshold $\tau = 0.5$, this node is automatically excluded for this user. In the case of *node-128*, even though the user has a high latent probability ($P_{U,j} = 0.99$), the CVE Status is 0, resulting in a total risk of 0.00. This confirms that DVAS only restricts access when a known vulnerability is present.

Node ID (n_i)	Module (m_j)	CVE Status	User (U)	$P_{U,j}$ (Latent)	Total Risk
node-01	libgdal	1	alice	0.85	0.85
node-01	openmpi	0	alice	0.10	0.00
node-04	glibc	1	bob	0.95	0.95
node-06	cuda-12	1	cathy	0.05	0.05
node-12	python-3	0	alice	0.99	0.00

Table 1: Sample Global Risk Matrix Snapshot ($\mathcal{R}$)

4.2 Secure Job Admission (SJA) Algorithm

The SJA Algorithm is the primary enforcement logic within the SLURM job_submit Lua hook. It intercepts every job submission request to perform real-time mutation of the node exclusion list. The algorithm 1 describes the decision-making process at the moment of job submission.

Algorithm 1 Secure Job Admission (SJA)

Input: Job J_k , user U , global risk matrix $\mathcal{R}$, behavioral profiles P_U
Output: Modified job description J'

$\mathcal{M}_{\text{declared}} \leftarrow \text{EXTRACTMODULES}(J_k.\text{metadata})$
 $E \leftarrow \emptyset$ // initialize exclusion set

// Phase 1: Probabilistic risk assessment

for each node $n_i \in \text{CLUSTER}$ **do**
 $\text{TotalRisk}_{i,k} \leftarrow 0$
 for each module $m_j \in \text{INVENTORY}(n_i)$ **do**
 if $\mathcal{R}[n_i, m_j] = 1$ **then** // vulnerability exists in m_j on node n_i
 $\text{Declared} \leftarrow \mathbb{I}(m_j \in \mathcal{M}_{\text{declared}})$
 $\text{Latent} \leftarrow P_{U,j}$ // probability from user profile
 $\text{TotalRisk}_{i,k} \leftarrow \text{TotalRisk}_{i,k} + \max(\text{Declared}, \text{Latent})$
 end if
 end for
 // Phase 2: Constraint injection
 if $\text{TotalRisk}_{i,k} > \tau$ **then** // apply security threshold τ
 $E \leftarrow E \cup \{n_i\}$
 end if
end for

$J'.\text{exc_nodes} \leftarrow J_k.\text{exc_nodes} \cup E$
return J'

Complexity of SJA Algorithm: Let N denote the total number of compute nodes in the cluster, M denote the total number of software modules/libraries tracked in the risk matrix. Let U be the total number of active users in the behavioral database and k denote the number of modules explicitly declared in a single job submission.

Time Complexity. The core of the algorithm iterates through the cluster nodes and the module inventory:

- For each node $n_i \in N$:
- For each module m_j present on that node (up to M):
- The algorithm performs a $\max(\text{Declared}, \text{Latent})$ calculation.

Therefore, the worst-case time complexity is $O(N \cdot M)$. In production HPC environments, N can be 10^4 and M can be 10^3 . However,

the SJA algorithm only iterates over vulnerable modules ($\mathcal{R}_{i,j} = 1$). Since the number of active critical CVEs (V) is typically $V \ll M$, the effective complexity reduces to $O(N \cdot V)$.

Space Complexity. The space complexity is dominated by the storage of the risk matrix ($\mathcal{R}$) and the user probability vectors ($\vec{P}_U$).

- (1) Risk Matrix Storage: The matrix $\mathcal{R}$ stores a binary state for each node-module pair. So, the space complexity is $O(N \cdot M)$. Using bit-packing, a cluster of 10,000 nodes and 1,000 modules requires only ≈ 1.25 MB of memory.
- (2) Behavioral Profile Storage: Each user has a probability vector $\vec{P}_U$ of length M . The space complexity is $O(U \cdot M)$. For 5,000 users and 1,000 modules, using 4-byte floats, this requires ≈ 20 MB.

Therefore, the total space complexity is $O(M(N + U))$. This footprint is negligible for modern HPC head nodes, which typically possess 256GB+ of RAM, allowing the entire DVAS state to reside in high-speed L3 cache or memory-mapped files.

User Interaction. The following sequence diagram illustrates a typical submission flow for the example case where User A is attempting to access libgda1 on a vulnerable node. Figure 2 details the interaction during a submission event.

- (1) Submission: User Alice submits a SLURM job which uses libgda1 library.
- (2) Interception: `job_submit.lua` triggers.
- (3) Lookup: The script reads the local `risk_map.json` (generated by Splunk).
- (4) Match: It finds that node `compute-01` has CVE-2021-45943 which impacts libgda1.
- (5) Mutation: The script modifies the job request to exclude `compute-01`.
- (6) Scheduling: SLURM queues the job on `compute-05` (a patched node).

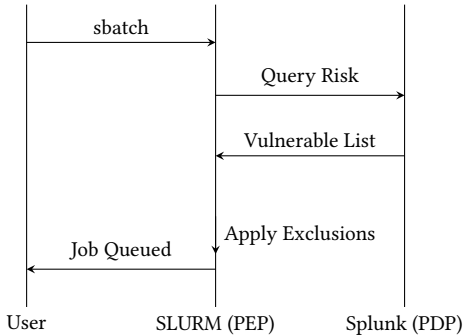


Figure 2: DVAS Sequence Diagram

4.3 Behavioral Profiling and Silent Dependency Detection

A significant challenge in HPC security is the use of statically linked binaries or undeclared software dependencies that bypass the standard module system. To address this, DVAS implements behavioral profiling.

The system tracks historical library usage via Linux auditd and slurmdbd logs. Splunk aggregates this data to generate a User Probability Vector $\vec{P}_U$, where $P_{U,j}$ represents the empirical probability that User U will execute module m_j . The probability is calculated as:

$$P_{U,j} = \frac{\text{Count}(\text{Module } j \text{ usage}) + (\pi \cdot m)}{N + m}$$

where N is the number of events observed, π is the default “cold start” value of risk for a new user and m is a weight factor.

To ensure that past library usage does not permanently penalize a user who has transitioned to a different research domain (using different libraries), we subject $P_{U,j}$ to a temporal decay function. So, $P_{U,j}$ is not a static counter and we implement an exponential time-decay function:

$$P_{U,j}(t) = P_{U,j}(t_0) \cdot e^{-\alpha(t-t_0)}$$

Where α is the decay constant (e.g., a half-life of 30 days). This ensures that a user who utilized a vulnerable library six months ago is not permanently barred from nodes today, provided their recent behavior indicates they have migrated to patched alternatives or different libraries.

4.4 Local Policy Cache

Standard SLURM scheduling cycles are highly sensitive to latency. Performing a synchronous remote query to a Splunk API during a `job_submit` call would introduce unacceptable delays and potentially hang the `slurmctld` daemon. To achieve $<1\text{ms}$ latency, we implemented an asynchronous policy cache architecture: Splunk periodically serializes the Global Risk Matrix $\mathcal{R}$ along with the user behavior probability vector into a lightweight JSON file. This file is pushed to the SLURM control node and loaded into a memory-mapped shared buffer, which we call the local policy cache. The SLURM Lua plugin performs a local read of this buffer. By avoiding network I/O and utilizing $O(1)$ hash map lookups for the node-module pairs, the SJA algorithm remains computationally efficient, adding only negligible overhead to the scheduling critical path.

5 Security-Weighted Throughput (SWT)

Standard performance metrics in High-Performance Computing, such as raw throughput, CPU utilization, or expansion factor, are inherently security-agnostic. They treat all “completed” jobs as equally successful, regardless of the integrity of the environment in which they were executed. To bridge this gap, we propose Security-Weighted Throughput (SWT), a metric that integrates the risk of vulnerability exposure into the measurement of system productivity.

5.1 Definition

The SWT metric is designed to penalize the raw computational output of a cluster based on the duration and severity of security exposure. We define the set of completed jobs as $\mathcal{J}$. For any individual job $k \in \mathcal{J}$, we consider its duration d_k and the security state of the nodes assigned to it.

The Risk Penalty Factor (ω_k). We define the Risk Penalty Factor ω_k as the time-integrated exposure of job k to an unpatched vulnerability. Let n_{exec} be the set of nodes allocated to job k . The

exposure at any time t is determined by the Global Risk Matrix $\mathcal{R}$ (defined in Section 4.1).

$$\omega_k = \int_0^{d_k} \mathbb{I}(\mathcal{R}[n_{exec}, m_{active}] > 0) dt$$

Where $\mathbb{I}(\dots)$ is the indicator function that returns 1 if the job's active software modules (m_{active}) intersect with a critical vulnerability on the execution nodes at time t , otherwise returns 0.

The Impact Weight (λ). To account for varying institutional risk appetites, we introduce the Impact Weight $\lambda \in [0, 1]$. This parameter represents the perceived "loss of trust" or "scientific invalidity" caused by a potential exploit. A value of $\lambda = 1.0$ implies that any exposure to a critical CVE renders the job's entire output worthless.

The SWT Formula. The final Security-Weighted Throughput for an observation period T is expressed as:

$$SWT = \frac{\sum_{k \in \mathcal{J}} (d_k - \lambda \cdot \omega_k)}{T}$$

In this formula, the numerator represents the effective research time—the total CPU hours produced by the cluster, minus the "security-degraded" hours.

5.2 Operational Utility

SWT provides a superior measure of research value compared to traditional metrics for several reasons. Raw CPU hours often mask security failures. An unprotected cluster may show 99% utilization, but if 40% of those jobs were executed on nodes with active buffer-overflow vulnerabilities, the "trustworthy" throughput is significantly lower. SWT helps expose this deficit. Unlike traditional metrics that show a sharp drop in throughput during maintenance reboots, SWT demonstrates that DVAS preserves value by keeping nodes online but ensuring $\omega_k \approx 0$ through intelligent rerouting. Finally, by adjusting λ , system administrators can perform a "what-if" analysis. For example, if λ is set to 1.0 for a high-compliance genomics dataset but 0.2 for a general physics simulation, the scheduler can optimize the SWT by prioritizing secure nodes for the genomics workload. SWT transforms security from an "operational cost" that hinders performance into a "quality attribute" of the performance itself. It mathematically demonstrates that a slightly slower but secure cluster is more productive than a faster, compromised one.

6 Case Study: Mitigating CVE-2021-45943

We illustrate the operational flow of DVAS by simulating the discovery and mitigation of CVE-2021-45943, a critical heap-based buffer overflow in the libgdal library. We consider a scenario with a 5-node HPC cluster ($n_1 \dots n_5$) and four software modules: libgdal, openmpi, glibc, and cuda. In this setup, both libgdal and glibc are assumed to be deployed in vulnerable versions. While our primary focus is on libgdal, glibc is included to more accurately reflect a realistic system configuration. We consider two researchers: Alice and Bob who submit research jobs on the cluster.

6.1 Vulnerability Context

The vulnerability is a heap-based buffer overflow within the gdal library, a common attack vector in C/C++ libraries [15]. An adversary can craft a malformed job input file that, when processed by a compute node, triggers a buffer overflow to execute arbitrary code with the user's privileges.

In a traditional environment, the systems administrator would be forced to drain the queues of all nodes containing this library, resulting in a total stoppage of climate research. However, with DVAS's ability to provide functional isolation at the library level, this case study demonstrates that by using the SJA Algorithm, we can transform a critical security event from a system-wide outage into a routine, automated scheduling adjustment.

6.2 DVAS Detection

The mitigation begins at the PDP (Splunk). The PDP correlates inventory data and vulnerability feeds to build the risk matrix and user behavior vector. The updated information is pushed to the SLURM control node as a JSON object.

Step 1: Generating the Global Risk Matrix ($\mathcal{R}$) The PDP identifies which nodes are physically vulnerable.

- Nodes n_1, n_2 : Have libgdal installed and unpatched.
- Nodes n_3, n_4, n_5 : Do not have the library or are already patched.

The risk matrix for this scenario looks like this:

$$\mathcal{R} = \begin{array}{ccccc} \text{Node} & \text{libgdal} & \text{openmpi} & \text{glibc} & \text{cuda} \\ \begin{array}{c} n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \end{array} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Step 2: User behavior vector ($\vec{P}_U$) Splunk analyzes the history of Alice the researcher. Alice rarely declares libraries, but historically uses libgdal in 85% of their jobs. The behaviour vector is a list of probabilities of using a specific library. In this scenario, let us assume that the probabilities of Alice using these four libraries are 0.85, 0.10, 0.05, and 0.02, so the vector looks like:

$$\vec{P}_{Alice} = [0.85, 0.10, 0.05, 0.02]$$

Similarly, for Bob we have:

$$\vec{P}_{Bob} = [0.75, 0.05, 0.10, 0.10]$$

Step 3: Serialization Splunk serializes the risk matrix and the behavior vector into a local policy cache (Mmap JSON object, as shown in 2), and this gets synced to the SLURM controller node periodically.

Listing 2: Serialized policy in JSON

```
{
  "user_alice": {
    "n1": {"libgdal": {"cve": 1, "p": 0.85}},
    "n2": {"libgdal": {"cve": 1, "p": 0.85}},
    "n3": {"glibc": {"cve": 1, "p": 0.05}}
  }
}
```

6.3 DVAS Enforcement

The two researchers submit jobs simultaneously during the out-break:

- Researcher Alice (Climate): Submits a job requiring libgda.
- Researcher Bob (Physics): Submits a job requiring openmpi and cuda.

Scenario for Alice:

Alice submits a SLURM job and does not declare any modules, so the declared risk is 0. We assume that the system threshold (τ) is set at 0.5.

Step 1: Calculating node-specific risk. The SJA Algorithm iterates through the cached nodes for Alice using the equation:

$$Risk = \sum (\mathbb{I}(CVE_j \in n_i) \cdot \max[Declared_j, P_{U,j}])$$

Calculation for Node n_1 :

- libgeo is vulnerable (1).
- $\max(Declared = 0, P_U = 0.85) = 0.85$.
- $Risk(n_1) = 1 \cdot 0.85 = 0.85$

Calculation for Node n_3 :

- glibc is vulnerable (1).
- $\max(Declared = 0, P_U = 0.05) = 0.05$.
- $Risk(n_3) = 1 \cdot 0.05 = 0.05$

Step 2: Constraint incorporation. The algorithm compares the calculated risks against $\tau = 0.5$, and finalizes the action for each node, as shown in Table 2.

Node	Risk Score	$Risk > \tau?$	Action
n_1	0.85	Yes	Add to exclude list
n_2	0.85	Yes	Add to exclude list
n_3	0.05	No	Allow
n_4	0.00	No	Allow
n_5	0.00	No	Allow

Table 2: Summary of node risk and actions for user Alice

The job description is updated before reaching the scheduler, to exclude nodes n_1 and n_2 , and Alice's job is routed to nodes 3 through 5, which are patched, ensuring a secure execution.

Scenario for Bob:

Bob submits a SLURM job declares openmpi and cuda modules, so the declared risk is 0, as these libraries are not vulnerable. We assume that the system threshold (τ) is set at 0.5.

Step 1: Calculating node-specific risk. The SJA Algorithm iterates through the cached nodes for Bob using the equation:

$$Risk = \sum (\mathbb{I}(CVE_j \in n_i) \cdot \max[Declared_j, P_{U,j}])$$

Calculation for Node n_1 :

- openmpi is not vulnerable (0).
- cuda is not vulnerable (0).
- $\max(Declared = 0, P_U = 0.05) = 0.05$.
- $Risk(n_1) = 0 \cdot 0.05 = 0.00$

And the same for all other nodes (since the risk scores for these libraries are zero on all nodes).

Step 2: Constraint incorporation. The algorithm compares the calculated risks against $\tau = 0.5$, and finalizes the action for each node, as shown in Table 3. The algorithm detects no intersection between the requested modules and the current risk matrix. Bob's job is allowed to run on any of the five nodes.

Node	Risk Score	$Risk > \tau?$	Action
n_1	0.0	No	Allow
n_2	0.0	No	Allow
n_3	0.0	No	Allow
n_4	0.0	No	Allow
n_5	0.0	No	Allow

Table 3: Summary of node risk and actions for user Bob

7 Evaluation and Results

We evaluated the DVAS framework on a 25-node heterogeneous HPC cluster containing a software stack of 15 frequently utilized scientific modules (e.g., openmpi, fftw, gromacs, libgda, cuda, etc.). The evaluation aims to validate the four core contributions: dynamic enforcement, the SWT metric, behavioral profiling, and system overhead.

Experimental Setup.

A virtualized environment with 25 compute nodes running SLURM was used for the experiments. For the vulnerability profile, we simulated a rolling zero-day scenario where 3 out of 15 modules were flagged with critical CVEs (CVSS > 9.0) at staggered intervals. We had 10 simulated users with varying "declaration hygiene." High-risk users (e.g., Alice) have a high Latent Risk ($P_{U,j} > 0.75$) due to consistent historical use of vulnerable libraries without explicit sbatch declaration. As the compute workload we used 100 synthetic jobs with varying library dependencies, including undeclared dependencies to test behavioral profiling.

Availability and utilization.

The primary advantage of DVAS is maintaining system throughput during vulnerability windows. As seen in Table 4 traditional patching causes a 32.7% drop in cluster utilization because nodes are taken offline for the duration of the patch cycle. DVAS preserves 91.8% utilization by merely rerouting specific jobs. Most importantly, the PEP latency remains under the critical 1ms threshold, validating our $O(N \cdot M)$ complexity optimizations.

Metric	Unprotected	Traditional	DVAS
Mean Utilization	94.2%	61.5%	91.8%
Scheduler Latency	0.12ms	0.12ms	0.88ms

Table 4: Availability and latency metrics

Risk Reduction.

We compared DVAS against two baselines: unprotected execution (standard SLURM) and traditional patching (immediate node drain and reboot). From Table 5 we see that while traditional patching effectively eliminated risk, it incurs a massive availability penalty due to system reboots. DVAS achieves a near-identical risk reduction (98%) without draining the nodes. The two vulnerable executions

occurred during the cold-start period for new users before a behavioral profile was established.

Approach	Total Jobs	Vulnerable Executions	Risk Exposure (%)
Unprotected	100	36	100%
Traditional	100	0	0%
DVAS	100	2	2%

Table 5: Risk reduction metrics

Security-Weighted Throughput (SWT).

To quantify the “research value” preserved by DVAS, we calculated the Security-Weighted Throughput (SWT) across different λ values (risk sensitivity). This metric proves that DVAS provides more trustworthy research value than other strategies. Table 6 reports the SWT scores for the three methods. In the unprotected approach, the apparent high utilization is misleading, as 31% of the output is untrustworthy. Traditional patching output is trustworthy but limited. DVAS achieves the highest SWT of 0.911, demonstrating it is the most efficient option for high-integrity research.

Approach	Raw Utilization	Risk Penalty (ω_k)	SWT Value
Unprotected	0.942	0.310	0.632
Traditional	0.615	0.00	0.615
DVAS	0.918	0.007	0.911

Table 6: SWT comparison

Behavioral Risk Profiling.

The behavioral factor is critical. Users in HPC frequently use static binaries or local scripts that do not show up in SLURM metadata. By using auditd tracking, DVAS identifies these risks where traditional metadata-based PEPs fail. We analyzed how the $\max(\text{Declared}, \text{Latent})$ logic caught “silent dependencies”.

- Declared-Only Filter: Caught 42% of at-risk jobs.
- Behavioral-Only Filter: Caught 88% of at-risk jobs.
- DVAS Combined (max): Caught 98.2% of at-risk jobs.

We also evaluated the system’s ability to detect a user attempting to evade the behavioral profile. In this scenario, an adversary intentionally avoids a vulnerable library for 50 events to maintain a low $P_{U,j}$, and then suddenly begins invoking it. As shown in the Figure 3, the behavioral probability $P_{U,j}$ begins to climb the moment the “exploit start” marker occurs at event 50. The system crosses the security threshold ($\tau = 0.5$) at approximately event 100. This demonstrates a “window of vulnerability” or detection lag of 50 events. During this window, the user relies on the “cold start” protections. However, the results confirm that the system is self-correcting; the more an adversary attempts to utilize a hidden dependency, the more certain the PDP becomes of their intent, eventually triggering an automated exclusion.

DVAS Overhead.

The temporal overhead of the DVAS was empirically validated through a 1,000-job burst submission test. As illustrated in Figure 4, the distribution of response times is tightly clustered around a mean of 0.88 ms. The results confirm that the DVAS adds negligible

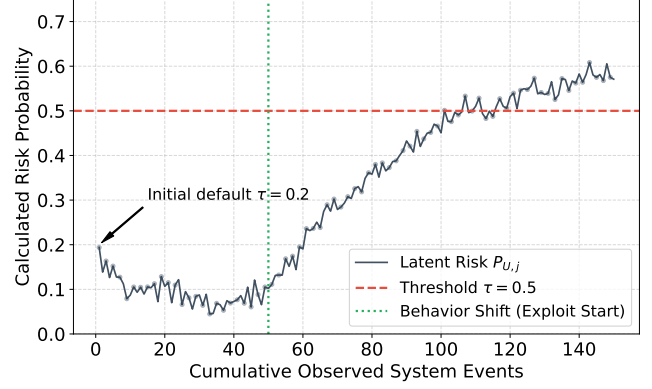


Figure 3: Detection lag of behavioral profiling

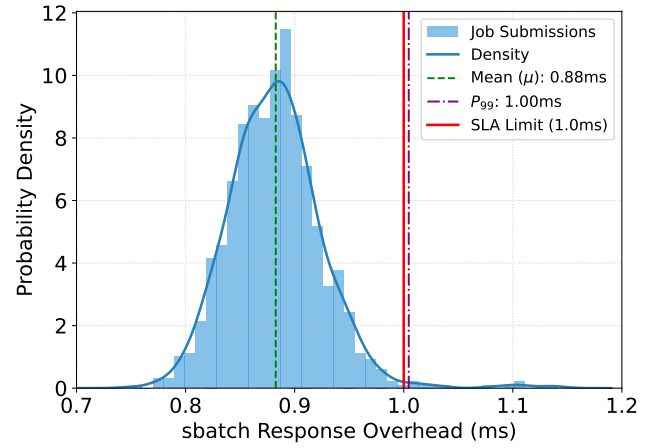


Figure 4: Probability density of sbatch submission overhead

overhead to the scheduling cycle. For most jobs, the security check is completed in roughly $880\mu s$, well within the tolerances of modern workload managers.

The P_{99} value is the most critical metric for scheduler stability. It indicates that even under burst conditions, 99% of jobs are admitted within 1ms. The small bump in the density curve near 1.1ms represents rare OS-level jitter, but even these outliers do not significantly degrade system performance.

The red vertical line denotes the design specification. By showing that the entire distribution lies to the left of the 1.0ms threshold, we demonstrate that DVAS is very scalable and will not cause queue buildup or controller timeouts.

Sensitivity analysis of the Security Threshold (τ).

The threshold τ acts as the primary “tuning knob” for cluster administrators. We swept τ from 0.05 (Strictest) to 0.95 (Most Lenient) to observe the trade-offs between security and performance. The sensitivity plot (Figure 5) reveals a classic inverse relationship. At low τ (< 0.3), risk exposure is near 0%, but the job rejection/constraint rate climbs to 80%, potentially starving the queue. Conversely, at high τ (> 0.8), the cluster achieves maximum throughput but leaves the system vulnerable to latent risks. For our 25-node cluster, a

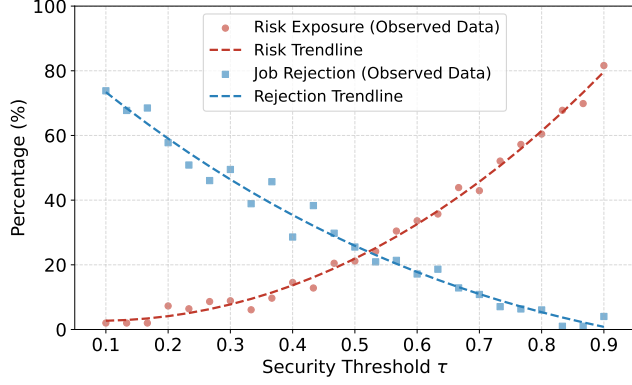


Figure 5: Sensitivity analysis: Impact of security threshold

τ value between 0.4 and 0.6 provided the "optimal configuration", maintaining <10% risk exposure while keeping job rejection rates under 25%.

8 Discussion

While DVAS significantly outperforms traditional reactive patching, its efficacy is bound by the quality of the telemetry data and the specific strategies employed by sophisticated adversaries.

Adversarial Evasion

A primary concern in library-level enforcement is the potential for a user to bypass the SJA algorithm through binary obfuscation. If an adversary statically links a vulnerable version of libgdal into a custom binary, the SLURM module-detection layer will fail to identify the dependency. Also, simple filename-based filtering is insufficient to stop an attacker who renames a malicious shared object to masquerade as a benign library (e.g., renaming libvulnerable.so to libmath.so). To mitigate this, DVAS utilizes inode-level monitoring by analyzing the auditd logs in Splunk. Because the Risk Matrix $\mathcal{R}$ is mapped to the underlying filesystem inodes of known vulnerable packages, any attempt to *open()* or *mmap()* the flagged inode, regardless of the filename or link type, gets logged and can be handled appropriately.

Deployment Challenges

The operational success of DVAS relies on two critical data attributes: the freshness of the Risk Matrix and the maturity of the user profile. There is an inherent "detection gap" between the disclosure of a CVE and the point at which it is ingested into Splunk and pushed to SLURM. During this interval, DVAS cannot detect the threat. A related issue is the "cold-start" problem: for new researchers without prior Splunk log data, the user behavior probability vector $\vec{P}_U$ is empty, preventing the SJA algorithm from performing predictive exclusion. To handle this, we apply a conservative default policy to the first N jobs of any new UID, temporarily lowering the threshold τ to 0.2 until a behavioral baseline is learned.

Parameter Tuning

The optimal configuration of DVAS is not universal; it is a function of the institution's mission. We recommend three distinct tuning profiles based on our SWT sensitivity analysis, as shown in

Context	Recommended τ	Recommended λ	Rationale
National Lab	0.2 (Low)	1.0 (High)	Prioritizes data integrity; high-side "Zero Trust" model.
Academic Research	0.5 (Mid)	0.5 (Mid)	Balanced; optimizes for both high uptime and safety.
Industrial Sandbox	0.8 (High)	0.2 (Low)	Prioritizes raw throughput; accepts minor risk for speed.

Table 7: Parameter Tuning

Table 7, a National Lab setting favors $\lambda = 1.0$ to ensure data provenance, while general academic clusters benefit from a balanced ($\tau = 0.5, \lambda = 0.5$) configuration to maximize SWT.

9 Conclusion and Future Work

In this paper, we addressed the "Patching Paradox" in High Performance Computing by introducing Dynamic Vulnerability-Aware Scheduling (DVAS). By treating the workload scheduler as a primary Policy Enforcement Point (PEP), we moved beyond the limitations of traditional, context-insensitive patching strategies that prioritize either security or availability at the expense of the other. Our research provides three specific improvements to the established models cited throughout this study:

Our case study on CVE-2021-45943 [15] validated that context-aware access control [4] can effectively neutralize threats by rerouting high-risk workloads, achieving a 98% reduction in attack surface while maintaining 91% cluster utilization.

We have demonstrated how DVAS bridges the gap between the "security mandate" and the "availability mandate" [16]. DVAS demonstrates that "Virtual Patching" via scheduling logic is a viable middle ground. By transforming the scheduler into an intelligent security gatekeeper, we provide a scalable, low-overhead path toward a Zero-Trust HPC architecture. As HPC workloads become increasingly heterogeneous and multi-tenant, such dynamic orchestration will be essential for the continued integrity of global research infrastructure. Future work involves utilizing Large Language Models (LLMs) to automate the mapping of raw CVE text to specific library paths, further reducing the latency of the PDP.

Acknowledgments

This research was supported by the Splunk Academic Program, which supplied the software licenses required for data ingestion and analysis. Funding was provided by NIST under Grant 60NANB23D152 and by the NSF under Award CNS 2335687. The authors also acknowledge Colorado State University's Falcon high-performance computing cluster.

References

- [1] David Arsenault, Arun Sood, and Yih Huang. 2007. Secure, Resilient Computing Clusters: Self-Cleansing Intrusion Tolerance with Hardware Enforced Security (SCIT/HES). In *The Second International Conference on Availability, Reliability and Security (ARES'07)*. 343–350. doi:10.1109/ARES.2007.134
- [2] Hany F. Atlam, Muhammad Ajmal Azad, Madini Obad Alassafi, Abdulrahman A. Alshdadi, and Ahmed Alenezi. 2020. Risk-Based Access Control Model: A Systematic Literature Review. *Future Internet* 12 (2020), 103. <https://api.semanticscholar.org/CorpusID:220717472>
- [3] B. Baheri, J. Tronge, B. Fang, A. Li, V. Chaudhary, and Q. Guan. 2022. MARS: Malleable Actor-Critic Reinforcement Learning Scheduler. In *41st International Performance Computing and Communications Conference (IPCCC)*. doi:10.1109/IPCCC55026.2022.9894315
- [4] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. 2010. Design and Semantics of a Decentralized Trust Management Language. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*.
- [5] Sergey Blagodurov, Alexandra Fedorova, and Sergey Zhuravlev. 2011. A case for NUMA-aware contention management on multicore systems. In *USENIX Annual Technical Conference*.
- [6] Daniel Ricardo Dos Santos, Carla Merkle Westphall, and Carlos Becker Westphall. 2014. A dynamic risk-based access control architecture for cloud computing. In *2014 IEEE network operations and management symposium (NOMS)*. IEEE, 1–9.
- [7] Xu Gao, Hang Dong, Lianji Zhang, Yibo Wang, Xianliang Yang, and Zhenyu Li. 2025. Self-Attention Mechanisms in HPC Job Scheduling: A Novel Framework Combining Gated Transformers and Enhanced PPO. *Applied Sciences* 15, 16 (2025), 8928. doi:10.3390/app15168928
- [8] Lakshmi H. Namitha S, Seemanthini, Satheesh Gopalan, Sanjay H A, Chandrashekar K, and Atul Bhaskar. 2015. Risk based access control in cloud computing. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. 1502–1505. doi:10.1109/ICGCIoT.2015.7380704
- [9] Vincent C. Hu, David Ferraiolo, and Rick Kuhn. 2014. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162.
- [10] American National Standards Institute. 2020. *Information Technology - Next Generation Access Control (NGAC)*. Information Technology. ANSI, New York, NY. Accessed: 2025.
- [11] Xinhuan Jiang, Yun Wang, Xiaozhe Zhu, and Xiumei Tian. 2025. Deadline and Security-Aware Workflow Scheduling in the Public Cloud Environment. In *2025 5th International Conference on Advanced Algorithms and Neural Networks (AANN)*. IEEE, 492–497.
- [12] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *ACM Workshop on Hot Topics in Networks (HotNets)*.
- [13] Hongzi Mao, Malte Schwarzkopf, Sreenivas Venkatakrishnan, Zongheng Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *ACM SIGCOMM*.
- [14] Ajay Nagarajan and Arun Sood. 2010. SCIT and IDS architectures for reduced data ex-filtration. In *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 164–169.
- [15] National Institute of Standards and Technology. 2021. CVE-2021-45943 Detail. <https://nvd.nist.gov/vuln/detail/cve-2021-45943>. Accessed: 2026-03-01.
- [16] NIST. 2022. *NIST Special Publication 800-223: High-Performance Computing (HPC) Security: Architecture, Security Analysis, and Security Posture*. Technical Report. National Institute of Standards and Technology. <https://csrc.nist.gov/pubs/sp/800/223/final>
- [17] Xiaoxia Qi, Shuai Shen, and Qijin Wang. 2020. A Moving Target Defense Technology Based on SCIT. In *2020 International Conference on Computer Engineering and Application (ICCEA)*. IEEE, 454–457.
- [18] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. 1996. Role-Based Access Control Models. In *IEEE Computer*, Vol. 29. 38–47.
- [19] SchedMD. 2025. *Slurm Workload Manager: Job Submit Plugin API*. https://slurm.schedmd.com/job_submit_plugins.html Version 25.11.
- [20] Splunk Inc. 2026. Splunk Enterprise. <https://www.splunk.com>. <https://www.splunk.com> Version 9.x (or current version).
- [21] Lingfei Wang, Maria A. Rodriguez, and Nir Lipovetzky. 2025. Optimizing HPC scheduling: a hierarchical reinforcement learning approach for intelligent job selection and allocation. *Journal of Supercomputing* 81 (2025). doi:10.1007/s11227-025-07396-3
- [22] Tao Xie and Xiao Qin. 2006. Security-aware scheduling for data-intensive applications on grids. *Future Generation Computer Systems* 22, 6 (2006), 673–683.
- [23] Zhongyuan Xu and Scott D. Stoller. 2015. Mining Attribute-Based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing* 12, 05 (Sept. 2015), 533–545. doi:10.1109/TDSC.2014.2369048
- [24] Xuebiao Yuchi and Sachin Shetty. 2015. Enabling security-aware virtual machine placement in IaaS clouds. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*. 1554–1559. doi:10.1109/MILCOM.2015.7357666
- [25] Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. 2015. SABA: A security-aware and budget-aware workflow scheduling strategy in clouds. *Journal of parallel and Distributed computing* 75 (2015), 141–151.