JANUS: Coordinating Vulnerability Prevention and Exploit-chain Mitigation for Containerized CI/CD Pipelines

Shafayat Hossain Majumder[1], Suryadipta Majumdar[1], Sourov Jajodia[1], and Patrick Jean-Baptiste[2]

[1] Concordia University, Montreal, Quebec, Canada
[2] Sunphinx Cybersecurity Inc., Montreal, Quebec, Canada
{shafayat.majumder,suryadipta.majumder, sourov.jajodia}@.concordia.ca, patrickjb@sunphinx.com

Abstract. Contemporary CI/CD practices typically address vulnerability prevention and intrusion mitigation as disjoint activities, which may result in residual risk even after container images pass pre-deployment checks. Pre-deployment scanners can overlook vulnerabilities obscured by image layering or insufficient metadata, whereas runtime detection mechanisms commonly rely on single-source observations and thus fail to capture multi-step exploit chains. This work presents JANUS, a coordinated framework for containerized CI/CD security across the pre- and post-deployment phases. Its key idea is that vulnerabilities should be handled according to when they can be addressed most effectively: JANUS uses a policy gate to decide which vulnerabilities must be remediated before release and which residual vulnerabilities are explicitly deferred to runtime monitoring. To support pre-deployment decisions under CI/CD time budgets, JANUS performs layer-aware incremental scanning and policy-controlled constrained patching on changed image layers. To protect against residual risk after deployment, it enforces the deferred set using abstract, stateful exploit-chain signatures that correlate system and network events through shallow-to-deep runtime inspection. Our evaluation on 346 container images (including all 146 official and 200 widely used community images) shows that JANUS improves vulnerability coverage by 42.4% relative to representative scanners in our setup (e.g., Grype, Trivy, CVE-Bin-Tool) while reducing analysis latency by 79.6%. Moreover, on our tested exploit chains, JANUS detects all chains that elude representative single-source IDS baselines. These results demonstrate the benefits of coordinating prevention and mitigation across the container lifecycle.

Keywords: Container Security · CI/CD Pipeline Security · Exploit Chain Detection · Vulnerability Scanning · Runtime Monitoring · eBPF

1 Introduction

Containerization has become a foundational technology for modern software deployment, offering portability, scalability, and rapid iteration across develop-

ment and production environments [55, 57]. As organizations increasingly adopt CI/CD pipelines to automate build, test, and deployment workflows [66, 75], securing container images throughout their lifecycle has become essential. Current practices typically rely on two major classes of defenses: pre-deployment vulnerability scanning [5, 15, 74] and post-deployment intrusion detection or mitigation [21, 63, 82]. Although each plays a critical role, these mechanisms are often treated as independent components, with little coordination between what is checked before deployment and what must be enforced at runtime.

Existing approaches that operationalize *shift-left* (pre-deployment) and *shift-right* (post-deployment) practices only partially address this coordination and face inherent limitations across the container lifecycle.

- **Shift-left (pre-deployment).** DevSecOps teams rely on Application Security Testing (AST) tools [73], including Static AST (SAST) and Dynamic AST (DAST). In the container setting, SAST-style image scanners (e.g., [5, 15, 74, 79]) can meet CI/CD latency requirements but are often strongest on OS/base-layer package metadata and may miss vulnerabilities introduced in application/package layers (e.g., due to layering effects or non-package artifacts), enabling cross-service or organization-level compromise [52]. DAST tools (e.g., [6, 13, 65]) can cover a broader class of vulnerabilities through active probing, but can incur substantial overhead under large rule sets and frequent releases, making naive use challenging for minute-level release cycles. Moreover, many existing tools lack efficient incremental mechanisms and must re-process unchanged layers when handling version updates [8].
- **Shift-right (post-deployment).** DevSecOps teams deploy Intrusion Prevention Systems (IPS) to limit exploitation of residual vulnerabilities [1, 61]. Behavior-based IPS [27, 44, 51] learn benign runtime profiles, but some CVE exploits and chains can evade these models by manifesting as low-noise or near-benign event sequences, including adversarially perturbed [36] or stepwise attack patterns [29]. Rule-based Host IPS (HIPS) and Network IPS (NIPS) systems (e.g., [18, 21, 23, 24, 41, 63]) can provide precise, interpretable signatures, but are commonly single-source and often limited in stateful correlation, making it difficult to capture multi-source, multi-step exploit chains that unfold over longer windows [35].

Additionally, recent works leveraging eBPF [48] and syscall/network telemetry [7, 26, 89] improve runtime visibility, but typically does not provide an explicit coordination mechanism to decide which vulnerabilities should be remediated pre-deployment versus accepted and deferred to runtime monitoring.

In this work, we propose Janus, a framework that coordinates pre-deployment vulnerability prevention with post-deployment exploit-chain mitigation for containerized systems. During pre-deployment prevention, Janus uses a configurable policy gate to decide whether vulnerabilities should be selectively remediated or explicitly accepted for runtime monitoring, making residual risk auditable and policy-driven. It also applies layer-aware incremental scanning to avoid redundant analysis [52, 90] and supports minimally invasive patching

for version pinning and outdated packages through controlled containerfile updates [10, 70]. During post-deployment mitigation, JANUS detects exploit chains by correlating system and network events with abstract state-machine signatures. Unlike prior work that typically monitors syscall, network, or host signals in isolation [24, 27, 41, 51], JANUS captures cross-source exploit behavior by extracting candidate event sequences following data provenance.

In summary, this work makes the following contributions:

- **Coordinated CI/CD security framework:** We present JANUS, a coordinated framework that links pre-deployment, layer-aware vulnerability prevention with post-deployment, stateful multi-source exploit-chain monitoring across the container lifecycle. JANUS supports configurable *policy gates* (numeric or tiered) to decide which vulnerabilities are remediated pre-deploy versus explicitly accepted and deferred to runtime monitoring, enabling auditable, policy-driven residual risk under CI/CD constraints.
- **Incremental, layer-aware vulnerability prevention:** We introduce a selective pre-deployment analysis pipeline that distinguishes base-layer from package-layer changes and reuses prior scan results to reduce redundant scanning in CI/CD. JANUS also provides a minimally invasive patching workflow for version-pinning and outdated-package patterns [10, 70], producing developer-visible patch diffs (and optionally policy-controlled auto-application) to reduce exposure while preserving build reproducibility.
- **Stateful, multi-source exploit-chain mitigation:** We develop a runtime detection mechanism that represents exploit chains as abstract state-machine signatures over system and network events. By correlating multi-source signals with contextual abstraction and shallow-to-deep inspection, JANUS increases the practical detectability of multi-step exploit chains compared to single-source signatures.
- **Empirical evaluation on real-world artifacts:** We evaluate JANUS on 346 real-world container images, including multi-step exploit chains across multiple applications. Our results show reduced pre-deployment analysis time via incremental scanning, automated identification of patch candidates for vulnerable packages, and successful detection of the tested exploit chains under our signature model when compared to representative single-source IDS baselines.

2 Preliminaries

This section provides background, motivating example and our models.

2.1 CI/CD Background

Fig. 1 provides a high-level view of the CI/CD workflow that produces container images through incremental, layer-based updates. In CI/CD, only a subset of layers typically change between versions, yet existing shift-left security tools

must re-scan the entire image and often rely on incomplete metadata or per-rule probing, making them poorly aligned with minute-level release cycles. Shift-right defenses encounter the opposite problem: host- and network-based IPS observe disjoint telemetry streams, limiting their ability to correlate multi-step behaviors that span system- and network-level events. These structural characteristics of container pipelines (frequent incremental updates, layered composition, heterogeneous telemetry, and strict time budgets) directly motivate the coordinated shift-left/shift-right design goals of JANUS.

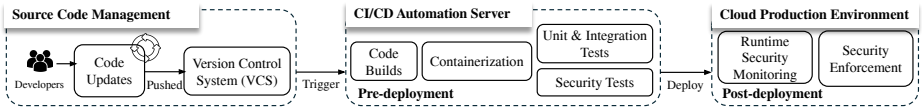


Fig. 1: Example of a typical CI/CD pipeline with integrated security measures.

2.2 Motivating Example

Fig. 2 shows an attack scenario (top), showcasing the limitations of existing works (middle), followed by hints to our solution (bottom).

Attack Scenario. This scenario (top of Fig. 2) illustrates the risk that arises when pre-deployment and post-deployment mechanisms operate independently. The Apache Superset v2.0.0 image contains a known critical vulnerability (CVE-2023-27524, CVSS 9.8) prior to deployment, which violates a typical pre-deployment policy gate. After deployment, two additional medium-severity vulnerabilities (CVE-2023-40610 and CVE-2023-37941) remain in the running container. Even if each is individually deemed acceptable under policy, an attacker can compose them into a multi-step exploit chain that yields a high-impact outcome. This example illustrates how uncoordinated shift-left and shift-right practices can allow exploit chains to persist in production.

Existing Works. The middle row of Fig. 2 illustrates why existing shift-left and shift-right mechanisms can fail to control this risk in practice. During pre-deployment, static image scanners such as Grype [5] are often strongest on OS/base-layer package metadata and can miss vulnerabilities introduced through application-layer artifacts or layering effects, allowing critical issues (e.g., CVE-2023-27524, CVSS 9.8) to slip through under tight CI/CD budgets. Dynamic scanners such as Nuclei can detect broader classes of issues through active probing, but may incur significant latency when running large rule sets, potentially exceeding the available CI/CD time budget. Consequently, critical vulnerabilities may still propagate into production. During post-deployment, residual vulnerabilities (e.g., CVE-2023-40610 and CVE-2023-37941) introduce additional runtime risk: network-only IPS may treat the triggering HTTP requests as benign (e.g., permitted `GET` paths), while host-based IPS may view the resulting system calls as indistinguishable from normal database access (e.g., `lseek`, `write`), making it difficult for either to detect the multi-step exploit chain in isolation.

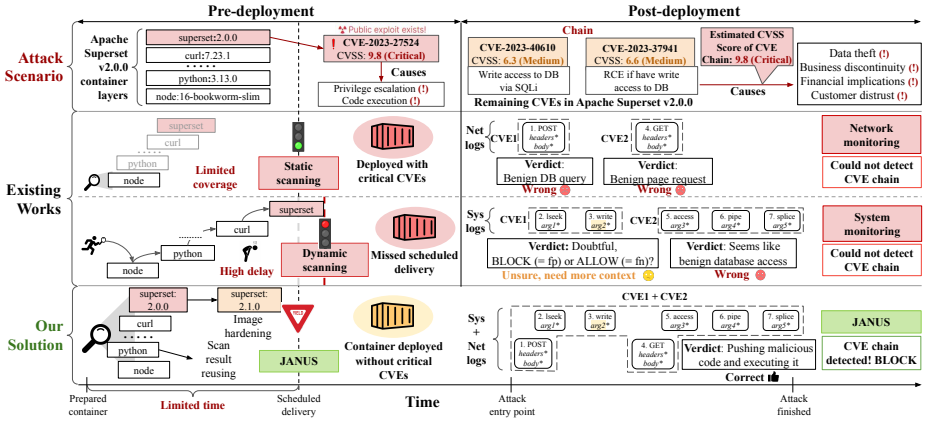


Fig. 2: A motivating example showing the attack scenario (top row), limitations of existing works (middle row) and JANUS's key ideas (last row).

Our Solution. The bottom of Fig. 2 shows how JANUS addresses these failures across both phases of the lifecycle. In pre-deployment, JANUS enforces a configurable *policy gate* (defined later in this section) by performing layer-aware analysis of both base and package layers and applying a minimally invasive, policy-controlled patching workflow (e.g., upgrading Superset v2.0.0 to v2.1.0 to address CVE-2023-27524 when such an upgrade is permitted). In post-deployment, JANUS monitors the explicitly deferred residual set and mitigates escalation by correlating system- and network-level events into multi-source exploit-chain signatures, enabling detection of chained attack progression based on their observable behaviors. We detail the methodology of JANUS in Sec. 3.

2.3 Threat Model

We consider a remote attacker who interacts with the containerized application through its externally exposed interfaces and can exploit any reachable vulnerability in application- or system-level packages, including combining multiple weaknesses into multi-step exploit chains. The attacker may craft low-noise or near-benign event sequences and interleave system-level and network-level actions to evade *single-source* intrusion detection, but is unable to compromise the host kernel, container runtime, CI/CD infrastructure, or modify the instrumentation used for system-call or network-level monitoring. On the defender side, the CI/CD environment, image registry, and container manifests (e.g., SBOM metadata) are assumed trustworthy, and the defender is able to observe system-call traces and network flows emitted by the container but may not rely on higher-level application logs. Attacks involving host/kernel compromise, compromised CI/CD or registry infrastructure, poisoned supply-chain components, hypervisor or hardware vulnerabilities, or purely stealthy zero-day exploits that leave no observable system or network footprint are out of scope. We do not claim robust-

ness against an adaptive adversary that actively studies and optimizes against a specific deployed signature set; rather, Janus targets known-vulnerability exploit chains and their observable multi-source behaviors in production deployments. Under these assumptions, the security objective of JANUS is (i) to enforce a configurable *policy gate* (defined below) that determines which vulnerabilities are remediated pre-deploy versus explicitly accepted and deferred to runtime monitoring, and (ii) to detect and mitigate the resulting monitored exploit chains at runtime when their signatures match observed multi-source behaviors.

2.4 Policy Gate Model

JANUS operationalizes deployment decisions through a configurable *policy gate* that maps vulnerability findings from pre-deployment analysis to one of three actions: *remediate before deploy*, *defer to runtime monitoring*, or *block deployment*. The goal is to make residual risk explicit and auditable while respecting CI/CD performance constraints and operational requirements (e.g., compatibility and release timelines). The policy gate is organization-configurable and can be instantiated either as a numeric rule (risk-score threshold) or as a tiered rule (severity/exploitability classes), optionally augmented with exception handling.

(i) Tiered Gates. A tiered gate assigns each vulnerability to an ordered category (e.g., *block*, *remediate*, *monitor*, *accept*) based on attributes such as severity (e.g., CVSS), exploit maturity/availability, affected component criticality, and whether the vulnerable component is reachable in the deployed image. In the simplest instantiation, vulnerabilities in the highest tier are *blocked* (deployment is prevented), mid-tier vulnerabilities are *remediated* when feasible, and remaining vulnerabilities are *deferred* and become the explicit target set for runtime monitoring.

(ii) Numeric Gates. A numeric gate aggregates vulnerability signals into a scalar risk score and compares it against a configurable threshold. Janus supports numeric scoring policies that sum or weight per-vulnerability contributions (e.g., by severity and exploitability), and uses the resulting value to decide whether (i) selective remediation is required prior to deployment, or (ii) the image may be deployed with a monitored residual set. This instantiation is compatible with the V-score/T-score formulation used in our implementation (Sec. 3.2B), even though JANUS is not tied to a specific scoring function.

(iii) Exceptions and Auditability. Real deployments often require controlled exceptions (e.g., temporary acceptance of a vulnerability due to compatibility constraints or pending upstream patches). Janus supports explicit allowlisting with metadata such as justification, owner, and expiry. Exceptions are treated as first-class policy outcomes: they are recorded for audit and, unless explicitly exempted, can be coupled with runtime monitoring to reduce time-at-risk.

Crucially, the policy gate defines the monitored set: vulnerabilities that are deferred (or accepted under exception) are translated into runtime monitoring obligations whenever signatures are available. This coupling ensures that

pre-deployment decisions do not merely "tolerate" residual vulnerabilities, but systematically bind them to post-deployment detection and mitigation.

3 Methodology

This section presents the methodology of JANUS.

3.1 Overview

Containerized CI/CD security poses two interlinked challenges: before deployment, vulnerability handling must support policy decisions under tight release budgets and incremental image updates; after deployment, residual risk must be monitored for multi-step exploit chains across heterogeneous event sources. JANUS addresses these challenges through exploring three ideas: (i) policy-gated coordination across pre- and post-deployment, (ii) layer-aware incremental analysis and constrained patching before deployment, and (iii) stateful multi-source runtime enforcement with shallow-to-deep inspection after deployment. Fig. 3 summarizes the two phases of JANUS. In pre-deployment prevention, JANUS scans image layers incrementally, evaluates findings against a configurable *policy gate* (numeric or tiered), and attempts constrained remediation for violations. Images that still violate policy are withheld unless explicitly overridden, while accepted vulnerabilities are recorded as residual risk and passed to runtime monitoring. In post-deployment mitigation, JANUS monitors this residual vulnerability set that are *accepted and deferred*, using curated exploit-chain signatures. It first abstracts known exploit traces into reusable state-machine signatures by removing instance-specific arguments (e.g., paths, IPs), then performs shallow inspection to flag early transitions and deep inspection to confirm terminal ones by correlating system-call and network events. When a final signature state is reached, JANUS executes the configured enforcement action (e.g., block or warn).

3.2 Pre-deployment Prevention

This section illustrates JANUS's vulnerability detection and prevention steps.

A. Vulnerability Detection. This step identifies vulnerabilities in a container image by separating layers that are amenable to fast metadata-based analysis from layers that require content-based analysis.

(i) Layer Classification. JANUS partitions each image into *base layers* and *package layers* so that each can be scanned with an appropriate method. Base layers typically modify system directories (e.g., `/lib`, `/etc`) and include package-manager metadata (e.g., `dpkg/status`, `rpm/Packages`), enabling efficient metadata-based scanning. Package layers modify application paths (e.g., `/app`, `/opt/app`) and often contain diverse, unstructured artifacts (binaries,

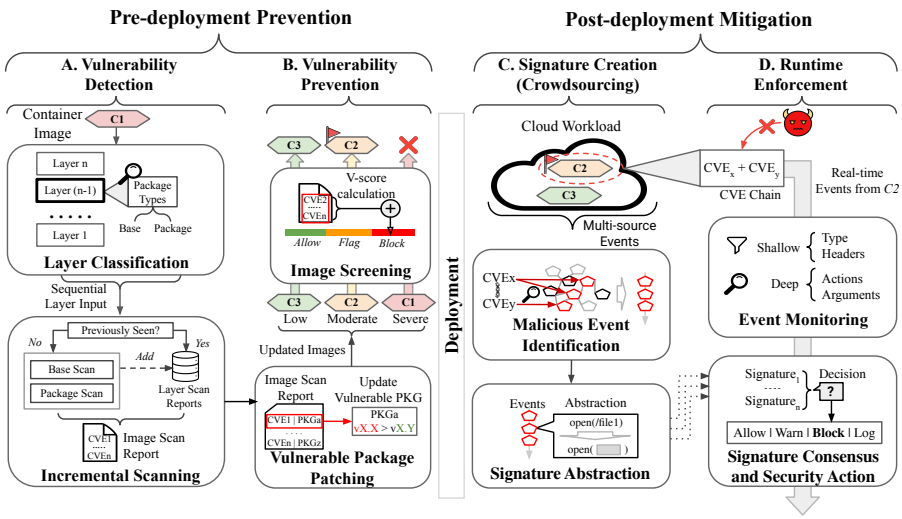


Fig. 3: Approach overview of JANUS.

scripts, configuration, archives) that benefit from content-based analysis. Classification is performed by unpacking each layer and extracting (i) the set of modified paths and (ii) the presence of package-manager metadata files; layers that modify predominantly system paths and contain package-manager metadata are labeled as base, while layers dominated by application paths (or lacking package metadata) are labeled as package. Although multi-stage or unconventional builds can blur these boundaries, our heuristic matches common container build practices and we evaluate its effectiveness in Sec. 4.1A. Ambiguous layers are conservatively classified as package layers to avoid missing application-level artifacts. This separation enables JANUS to apply more thorough analysis where needed (package layers) while keeping base-layer scanning efficient.

(ii) Incremental Scanning. This step reduces pre-deployment analysis cost by scanning only layers that change across image versions. As container builds commonly reuse layers via build caching and incremental development practices [20, 40], each layer's SHA256 digest (stored in the image manifest) allows JANUS to detect unchanged layers and reuse their prior scan artifacts (e.g., package inventory and scan findings). To account for evolving vulnerability knowledge, JANUS re-evaluates reused layer inventories against updated vulnerability databases (e.g., via SBOM/package metadata matching as supported by modern registries and scanners [3, 28]). For changed layers, JANUS performs full scanning and then merges layer-level results from top to bottom, giving precedence to newer layers to reflect file and package overrides introduced later in the build. This approach avoids redundant computation while preserving consistent, image-wide vulnerability reporting (see Sec. 4.1A).

Example 1. Fig. 4 illustrates vulnerability detection for Apache Superset v2.0.0 by contrasting newly introduced and previously scanned layers. After decompos-

ing the image, JANUS determines which layers have prior scan records (❶). For example, Layer 19 (`L19`, a package layer) is new, while Layer 1 (`L1`, a Debian base layer) has been seen before. In the new-layer case, JANUS performs full scanning on `L19`, extracting an inventory of 57 packages and matching them to associated CVEs (❷a), and stores the resulting layer report for reuse. In the reused-layer case, `L1`'s layer report is retrieved from the cache and re-evaluated against the current vulnerability database (❷b), yielding an average 80.4% reduction in processing time relative to rescanning all layers (Sec. 4.2). Finally, all layer-level results are merged to produce the complete image report (❸), including CVE severities and available patch candidates.

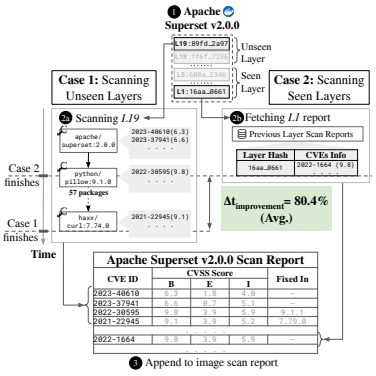


Fig. 4: Example of *vulnerability detection.*

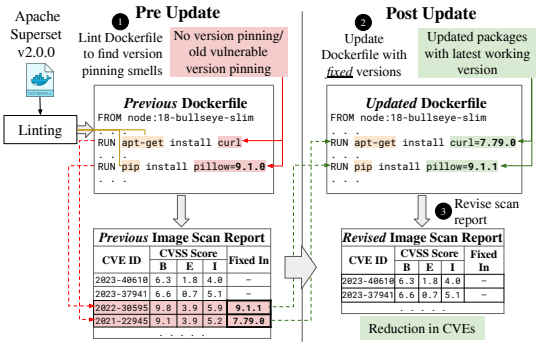


Fig. 5: Example of *vulnerable package patching.*

B. Vulnerability Prevention. This step applies policy-controlled remediation for vulnerabilities that violate the pre-deployment policy gate, and then re-screens the resulting image for deployment readiness.

(i) Vulnerable Package Patching. To remediate vulnerabilities that violate the policy gate before deployment, JANUS analyzes the container build specification (e.g., Dockerfile) for version-related issues and proposes constrained package upgrades. Outdated dependencies or missing version pinning are a common source of known vulnerabilities [10], and manual tracking is infeasible under frequent package updates [40]. JANUS detects such issues through static linting and rewrites the corresponding directives to target a patched version according to the underlying package ecosystem (e.g., distribution repositories and security advisories), guided by CVE metadata (e.g., affected version ranges) from vulnerability databases, such as NVD [81]. The image is then rebuilt via the standard build process (i.e., without directly modifying existing layers), producing a developer-visible patch diff and preserving build reproducibility. JANUS can be configured to require project tests or smoke checks when available; otherwise,

it defaults to developer-visible suggestions. When multiple upgrade candidates satisfy the configured constraints (e.g., same-major semver bump, allowlisted packages), JANUS selects the least intrusive candidate by default; if no candidate satisfies the constraints, JANUS marks the image non-compliant and surfaces the remediation requirement to the developer. The resulting image is rescanned to produce an updated vulnerability report.

Example 2. Fig. 5 illustrates vulnerable package patching for the same image as Example 1. Two Dockerfile lines exhibit version-pinning issues: one lacks an explicit version and another pins an outdated version. During pre-update, JANUS lints the Dockerfile to detect these issues (❶) and uses the vulnerability scan report together with version-range metadata to select minimally intrusive fixed-version candidates under the configured policy (❷). For instance, an unpinned `curl` installation exposes CVE-2021-22945, for which JANUS selects a fixed version (e.g., v7.79.0), while `pillow` v9.1.0 exposes CVE-2022-30595, for which JANUS selects an updated version (e.g., v9.1.1). After updating the Dockerfile (❸) and rebuilding the image, these CVEs are no longer reported in the resulting scan, illustrating how correcting version-pinning issues can mitigate known vulnerabilities.

(ii) Image Screening. This step determines whether an image satisfies the pre-deployment *policy gate* by estimating its aggregate vulnerability risk under a configurable scoring rule. As relying solely on CVSS base scores may misrepresent practical risk [46,62], JANUS supports numeric gating as one instantiation of the policy gate: it computes an image-level vulnerability score (V-score) from the scan report and compares it against a configurable threshold (T).[3] Numeric gating is one instantiation used in some CI/CD policies; we adopt it here for concreteness and auditability [16]. The V-score aggregates vulnerabilities using a weighted arithmetic mean inspired by [14, 85], combining CVSS base (B_i), exploitability (E_i), and impact (I_i) metrics [25] with user-defined weights (W_B, W_E, W_I). For each vulnerability i, JANUS computes $V_i = (W_B B_i + W_E E_i + W_I I_i)/(W_B + W_E + W_I)$, and the image-level score is the mean of all V_i. Under this numeric instantiation, deployment is permitted only if V remains below T; otherwise, the image is marked non-compliant and remediation (or an explicit exception) is required prior to deployment.

Example 3. Fig. 6 illustrates how JANUS applies the V-score under two risk preferences. The scan report for Apache Superset v2.0.0 contains two CVEs (❶), and both organizations use the same threshold (T=4.5) but assign different weights (❷). In Case 1, a financial institution prioritizes *impact* and sets W_I=0.7, yielding a V-score of 4.6>T, leading to rejection. In Case 2, a tech startup emphasizes *exploitability* with W_E=0.7, producing a lower V-score of 2.62<T, and therefore permits deployment under the gate (❸). This example demonstrates how distinct weighting policies result in different deployment outcomes despite identical vulnerability sets.

[3] Janus also supports tiered gating rules (e.g., block/remediate/monitor) as described in Sec. 2.4.

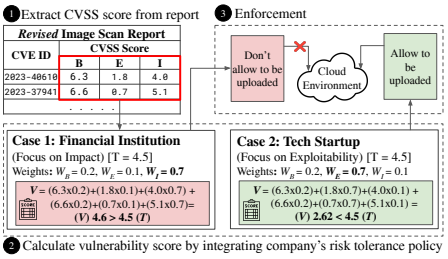


Fig. 6: Example of *image screening* for two security goals.

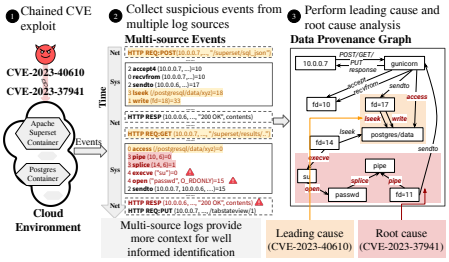


Fig. 7: Example of *malicious event identification*.

3.3 Post-deployment Mitigation

This section details JANUS's *signature creation* and *runtime enforcement*.

C. Signature Creation. This step constructs exploit-chain signatures as ordered, multi-source event patterns (e.g., network logs and system-call events).

(i) Malicious Event Identification. This step identifies candidate exploit sequences across multiple sources for signature construction. Traditional IPS tools often analyze either system or network events in isolation, which can miss cross-source attack context [42]. JANUS addresses this by tracing exploit behavior spanning both domains and, given the limitations of fully automated extraction [77], incorporates a human-in-the-loop workflow informed by public traces and root-cause analysis [33, 88]. The step collects system and network events (with additional sources discussed in Sec. 5), orders them temporally, and consolidates them into provenance graphs to represent candidate malicious chains (nodes: entities such as files or processes; edges: event relations derived from system calls or network logs).

Example 4. Fig. 7 shows how JANUS identifies candidate malicious events for the CVE-2023-40610 → CVE-2023-37941 chain in Apache Superset v2.0.0. After the exploit (❶), relevant system and network events from the container are collected and timestamp-sorted (❷). The chain begins with an HTTP `POST` to `/superset/sql_json` and ends with a `GET` to `/superset/results/`, with intermediate events forming a consolidated provenance graph (❸). An expert observes that the initial `POST` leads to a database update via a file-offset operation (e.g., `lseek`) (CVE-2023-40610, yellow), and escalation occurs when the attacker accesses the sensitive `passwd` file (CVE-2023-37941, red). This demonstrates how JANUS supports experts in surfacing causal event sequences for exploit-chain signature construction.

(ii) Signature Abstraction. This step removes instance-specific details from candidate events to produce reusable exploit-chain signatures. As values such as process IDs, arguments, and return codes can vary across deployments, abstraction helps signatures generalize beyond a single trace. JANUS follows the

three-level abstraction scheme of [41]: (i) precise matching for parameters that must remain exact (e.g., syscall names, flags), (ii) fuzzy matching for variable fields (e.g., IP addresses, file descriptors), and (iii) optional ignoring for context-dependent values when they are not essential to the chain semantics (e.g., specific argument values).

Example 5. Fig. 8 shows the signature abstraction step. The top table lists system calls and network logs from the Apache Superset v2.0.0 RCE sequence, with parameters marked for abstraction; the bottom table shows the resulting signature. A `POST` request to `/superset/sql_json` (green) is preserved as a precise match, while file descriptor 18 (red) is abstracted as variable A to capture instance-specific elements. Grey parameters (e.g., IP addresses) are ignored under the abstraction policy.

Scalability of Signature Creation. Community- and vendor-maintained signatures are common in both research and industry (e.g., [41, 63, 65]). In practice, signature authoring requires security expertise and is often contributed by independent researchers or dedicated security teams; broader vulnerability-disclosure ecosystems (e.g., HackerOne [31], BugCrowd [9]) further illustrate the availability of such expertise. While signature creation is not fully automated, community-driven template ecosystems demonstrate that it can grow to large scale; for example, Nuclei templates have nearly 1,000 open-source contributors.

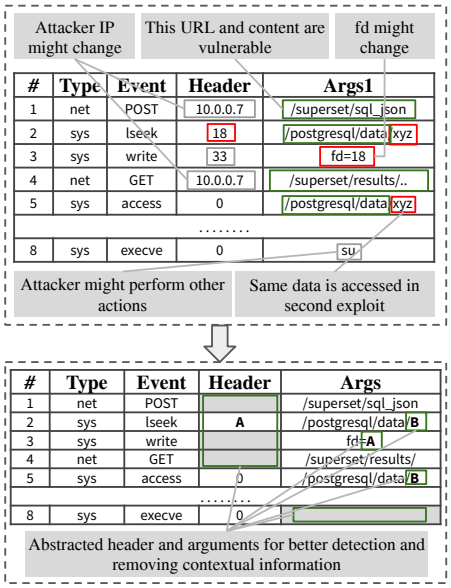


Fig. 8: Example of *signature abstraction* for Apache Superset RCE chain.

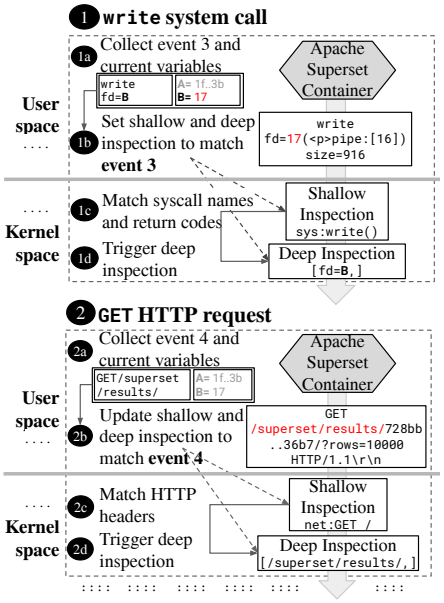


Fig. 9: Example of *runtime enforcement* for events 3 & 4 of RCE chain.

D. Runtime Enforcement. This step enables exploit prevention through shallow and deep event monitoring and security enforcement.

(i) Event Monitoring. The goal of *event monitoring* is to surface malicious activity from large volumes of real-time events. Unlike HIPS/NIPS systems that rely on user-space `ptrace` hooks [41,47], which incur latency and overhead [49], JANUS operates entirely in kernel space using lightweight kernel modules and performs detection in two stages: shallow filtering followed by deep inspection.
Shallow Filtering: This stage rapidly filters system and network events by matching lightweight metadata (such as event type, name, and return code) against the current signature step. Only events satisfying these coarse indicators are marked suspicious, substantially reducing deep inspections and avoiding unnecessary delays under high throughput (as evaluated in Sec. 4.2B, 4.2C).
Deep Inspection: For the filtered subset, JANUS performs argument-level verification. It checks each event's arguments against the exploit signature: abstracted arguments are matched through tracked variable bindings, while non-abstracted arguments require exact matches, ensuring accurate identification of exploits.

(ii) Signature Consensus and Security Action. This step combines verdicts from multiple signatures representing the same exploit chain to reduce the impact of noisy or conflicting signatures. As JANUS relies on community- or expert-authored signatures (Sec. 3.3C), PoC variants may disagree or contain errors, leading to false alarms or missed detections. JANUS addresses this using a K-out-of-N fusion rule [71], triggering enforcement when at least K of N signatures match. Organizations can tune K to balance sensitivity and false-alarm tolerance (related concerns are discussed in Sec. 5).

Fig. 10 summarizes the consensus process. Given real-time logs, a chosen K, and signatures $S_{\text{exp}} = \{s_1, \ldots, s_N\}$, the monitor outputs predictions $P = \{p_1, \ldots, p_N\}$ indicating which signatures match. If at least K signatures match, JANUS triggers the configured action. As future work, JANUS could incorporate confidence-weighted voting [43], where community feedback informs per-signature confidence scores to resolve conflicts more robustly. Upon confirmation, JANUS executes a security action (`allow`, `log`, `warn`, or `block`).

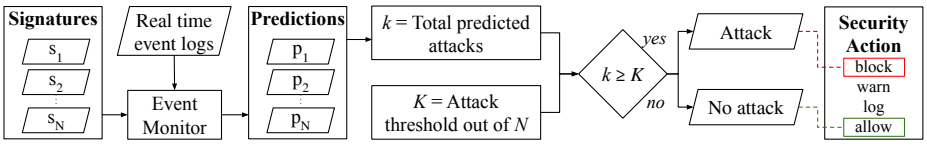


Fig. 10: Process of *signature consensus and security action.*

Example 6. Fig. 9 illustrates *runtime enforcement* for an Apache Superset RCE chain. Two consecutive signature steps are matched: `event 3` (a `write` syscall; ❶) and `event 4` (an HTTP `GET` request; ❷). For `event 3`, the file-descriptor variable is already bound as B=17 (❶a). Shallow filtering detects a `write` ref-

erencing `fd=B` (1b, 1c), and deep inspection confirms the argument match (1d). The monitor then advances to `event 4`, targeting `/superset/results` (2a). Shallow filtering selects incoming `GET` requests (2b), and deep inspection verifies the exact URL (2c, 2d). Once both events are validated, this signature segment is marked as matched. Upon matching all eight steps, JANUS triggers the configured `block` action (e.g., terminating the offending process or connection), thereby stopping the chain. If the chain remains incomplete, JANUS applies the configured non-blocking action (e.g., `allow` or `log`).

We provide JANUS's implementation challenges and design in App. A.

4 Experiment

This section evaluates JANUS through the following questions:
RQ1. How does JANUS compare to representative SAST/DAST-style baselines (as instantiated in our setup) in vulnerability discovery and coverage?
RQ2. To what extent does JANUS 's patching workflow eliminate vulnerabilities by upgrading vulnerable packages under the configured constraints?
RQ3. Can JANUS detect the evaluated multi-step exploit chains across the considered CVE classes under our signature model?
RQ4. How much does incremental scanning reduce pre-deployment analysis time compared to rescanning all layers?
RQ5. How much does shallow filtering reduce deep-inspection workload and overhead?
RQ6. What is the runtime overhead of JANUS, and how does it scale with event rate and the number of active signatures?

Experimental Setup and Dataset. We evaluate JANUS on a two-node local Kubernetes cluster (Ubuntu 22.04.3, 4 vCPUs/8 GB RAM per node) running Linux kernel 6.5.0–41, except for CVEs requiring older kernels (e.g., CVE-2024-21626). Our dataset includes 146 official Docker images and the top 200 community images on Docker Hub [19] (∼300M pulls each), plus Vulhub [87] for efficiency tests. For post-deployment evaluation, we construct exploit signatures using real PoCs, with each requiring 6–8 hours; automated sequencing and provenance analysis reduce manual effort. We implement signatures for 20 CISA KEV CVEs [17] and several multi-step chains from established exploit repositories (e.g., Metasploit [67]). All response-time measurements are performed on a local network to avoid external latency.

Ethical Consideration and Responsible Disclosure. We analyze only public Docker Hub images, but the vulnerabilities found are not public. Since maintainer contacts are unavailable, we disclosed all findings to Docker Hub and shared collection timestamps and scanner versions for transparency.

4.1 Security

A. Comparison against SAST/DAST on Detection Coverage, Accuracy and Efficiency. First, we evaluate coverage for RQ1 by comparing JANUS

with two widely used SAST tools: Grype [5] (metadata-based) and CVE-Bin-Tool [37] (content-based). For fairness, JANUS uses the same scanners internally for its metadata and content analyses. All three tools are run independently on our full image dataset with their latest CVE databases, and we compare total and unique CVEs detected. Results (Table 1) show that neither standalone scanner covers all vulnerabilities across heterogeneous layers. For official images, Grype and CVE-Bin-Tool detect 104.45 and 133.57 CVEs on average, whereas JANUS detects 217.06 by applying each scanner to the layer type where it is most effective. Similar trends for unique CVEs indicate that JANUS 's gains are not due to duplicates. We also evaluate a naive union of both scanners. Although this improves coverage over either tool alone, it requires re-scanning all layers on every run. In contrast, JANUS achieves comparable coverage while avoiding repetition through incremental, layer-aware scanning, yielding better efficiency. Overall, JANUS provides higher coverage than existing SAST tools across both official and community images.

Dataset	**Tool**	**Low**	**Medium**	**High**	**Critical**	**Total**
	Average Total CVEs Per Image					
Official (146 images)	Metadata-based [5]	19.08	37.70	38.32	9.35	104.45
	Content-based [37]	7.17	56.61	56.34	13.45	133.57
	JANUS	24.25	86.38	86.42	20.01	217.06
Community (200 images)	Metadata-based [5]	25.81	37.86	27.95	7.45	99.07
	Content-based [37]	12.58	98.94	86.84	15.36	213.72
	JANUS	38.25	129.78	107.61	21.05	296.69
	Average Unique CVEs Per Image					
Official (146 images)	Metadata-based [5]	16.19	26.31	24.67	5.87	73.04
	Content-based [37]	3.93	29.02	29.54	7.14	69.63
	JANUS	19.21	47.33	44.97	10.21	121.72
Community (200 images)	Metadata-based [5]	24.87	25.48	18.25	4.69	73.29
	Content-based [37]	5.58	43.74	39.78	8.01	97.11
	JANUS	30.30	62.20	50.84	10.94	154.28

Table 1: Comparison of JANUS with two types of SOTA vulnerability detection tools: metadata- [5] and content-based [37], on our dataset.

Second, we evaluate accuracy and time for RQ1 by comparing JANUS with SAST and DAST baselines on 164 labeled Vulhub environments, focusing on 57 CISA KEV CVEs. Grype [5] and Nuclei [65] serve as the SAST and DAST baselines, and JANUS uses the same scanners and CVE datasets for consistency. Fig. 11a shows that JANUS attains accuracy comparable to dynamic scanning and outperforms both SAST and DAST for low–high severity CVEs. Accuracy is slightly lower than Nuclei for critical CVEs because Nuclei's rules prioritize CVSS > 4.0; broader CVE datasets will improve JANUS 's critical-CVE coverage. Overall, JANUS improves accuracy over Grype by nearly 40%. Fig. 11b shows that JANUS reduces detection time by 76% relative to Nuclei while remaining

only marginally slower than Grype. Thus, JANUS achieves a balance of dynamic-level accuracy and static-level efficiency for fast, comprehensive detection.

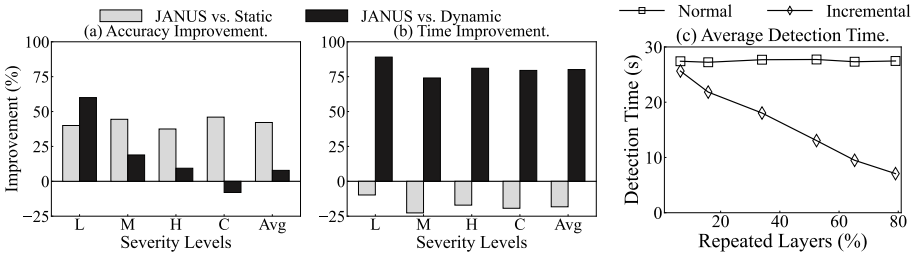


Fig. 11: Relative accuracy and runtime gains over static (SAST) and dynamic (DAST) scanners. `L`, `M`, `H`, `C`: low, medium, high, critical CVEs; `Avg`: Aggregation of all CVE severity levels, averaged.

Fig. 12: Detection time per image for six repeated layer datasets for normal vs. incremental scanning.

B. Evaluation of Vulnerable Package Patching. To answer RQ2, we evaluate the effectiveness of our Dockerfile-based package patching (Sec. 3.2) in updating vulnerable packages while preserving build success. We crawl GitHub repositories linked to our image dataset and collect 80 Dockerfiles (40 official, 40 community). Each is linted with Hadolint [32] to detect version-pinning issues; patched versions are substituted, images are rebuilt to verify build success, and re-scanned to quantify vulnerability reduction. Table 2 shows substantial reductions in vulnerable packages after patching: 79.7% for official images (26.05 fewer packages on average) and 74.7% for community images, with all patched Dockerfiles building successfully. Thus, selective package-level patching significantly reduces exposure without introducing build failures.

C. Comparison against IPS Tools on CVE Chain Detection. To answer RQ3, we first compare JANUS with Suricata [63] and Sysdig [21] on detecting multi-step CVE chains, which require cross-source context missing in existing IPS tools. We evaluate five chains (Table 3), each exploited at least ten times using custom PoCs and Metasploit. For fairness, JANUS loads only chain-level signatures, and the IPS baselines load only signatures for the individual CVEs composing each chain. Table 3 shows that chained exploits often yield higher effective severity than their individual CVEs (e.g., the rConfig SQLi→RCE chain reaches CVSS 10.0 despite ending in a medium-severity CVE). Suricata and Sysdig fail to detect any full chains, missing intermediate steps or misclassifying sequences due to their single-source views. In contrast, JANUS 's multi-source, stateful matching detects all five chains without disrupting benign activity, demonstrating that CVE-chain detection requires coordinated system–network logic that existing IPS tools lack.

Metrics	Official Images		Community Images	
	Vulnerable Packages	Build Success	Vulnerable Packages	Build Success
Before	32.70 (100%)	40 (100%)	44.71 (100%)	40 (100%)
After	6.65 (20.3%)	40 (100%)	11.30 (25.3%)	40 (100%)

Table 2: Comparison of CVE reduction before and after *Vulnerable Package Patching* step.

CVE Chains	Name	Sev.	Det. [63]	Det. [21]	Det. JANUS
2020-10220 (7.5) & 2020-23151 (7.5) & 2019-19585 (4.6)	rConfig SQLi & Command Injection RCE	10.0	×	×	✓
2021-34074 (7.5) & 2021-35501 (3.5)	PandoraFMS 754 - XSS and RCE	9.0	×	×	✓
2023-40610 (6.3) & 2023-37941 (6.6)	Apache Superset Signed Cookie RCE	9.8	×	×	✓
2022-46169 (9.8) & 2024-21626 (8.6)	Cacti Command Execution To Container Escape	10	×	×	✓
2021-44228 (9.3) & 2024-21626 (8.6)	Log4jShell To Container Breakout	10	×	×	✓

Sev. : Severity, Det. : Detected By

Table 3: Comparison of Suricata [63], Sysdig [21], and JANUS in detecting five real CVE chains.

CVE	Name	Sev.	Det.
2023-28432	MinIO Information Disclosure	7.5	-S–J
2023-23752	Joomla Access Control	5.3	-S–J
2021-40438	Apache SSRF	9.0	-S–J
2021-35501	PandoraFMS Stored XSS	5.4	-S–J
2020-10220	rConfig SQL Injection	9.8	-S–J
2024-21626	Leaky Vessels	8.6	—FJ
2021-4034	Polkit Privilege Escalation	7.8	—FJ
2020-23151	rConfig Command Injection	9.8	—FJ
2019-19585	rConfig Privilege Escalation	4.6	—FJ
2023-40610	Apache Superset Privilege Escalation	6.3	—-J
2023-37941	Apache Superset RCE	6.6	—-J
2022-46169	Cacti Command Injection	9.8	—-J
2021-34074	PandoraFMS File Upload	9.8	—-J
2020–11978	Apache Airflow RCE	9.8	—-J
2018-14634	Integer Overflow in Linux Kernel	7.8	—-J
2023-46604	Apache ActiveMQ RCE	9.8	-S-FJ
2023-29300	Adobe ColdFusion Deserialization	9.8	-S-FJ
2021-44228	Log4j2 Remote Execution	9.3	-SfFJ
2021-22205	GitLab RCE	10	sS-FJ

s: detected by Suricata [63] (default rule), S: detected by Suricata (custom rule), f: detected by Sysdig [21] (default rule), F: detected by Sysdig (custom rule), J: detected by JANUS, -: not detected, Sev.: Severity, Det.: Detected By

Table 4: Comparison of Suricata [63], Sysdig [21], and JANUS in detecting exploitation of 20 CVEs across multiple types and severity.

Second, we test whether JANUS 's signatures generalize across diverse CVE classes. We evaluate 20 representative CVEs from the 2023 CISA KEV catalog (CVSS 4.6–10), spanning SQL injection, command injection, RCE, etc. Suricata, Sysdig, and JANUS are deployed on a Kubernetes host, and each CVE is exploited at least ten times using scripts derived from public PoCs. For fairness, IPS baselines load only the signatures for these CVEs, and we assess both their default rules and manually refined custom rules (methodology in App. B). As shown in Table 4, default IPS rules detect almost none of the CVEs (1/20 for both tools). Custom rules improve detection (9 for Suricata, 8 for Sysdig) but still miss several exploits, including high-severity cases such as CVE-2022-46169 (CVSS 9.8). These gaps stem from the same limitation observed in chained exploits (Sec. 4.1): single-source, single-event matching cannot capture the multi-step behavior required for accurate recognition. Even template rules fall short, as Suricata catches only one of two SQL injection CVEs. In contrast, JANUS's multi-source, stateful signatures detect all 20 exploits, demonstrating broader class coverage and consistently outperforming existing IPS tools.

4.2 Efficiency

A. Impact of Incremental Scanning on Efficiency. To answer RQ4, we measure the time savings from JANUS 's pre-deployment *incremental scanning*, which reuses updated scan results for repeated layers. To vary repetition levels, we collect 50 recent tags from 25 popular community images and form six

datasets with 6%–80% repeated layers. Each image is scanned twice: once using incremental scanning and once using full scanning. Fig. 12 shows scan time falling as layer repetition increases. At 6% repetition both methods take 27 seconds, but at 80% repetition incremental scanning drops to 7 seconds while full scanning is unchanged. Thus, reusing prior layer results yields substantial time savings when images share layers.

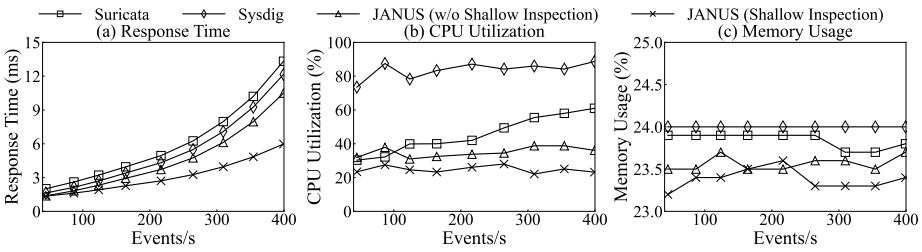


Fig. 13: Efficiency of *runtime event monitoring* shown by measuring response time (per event), CPU and memory utilization for varying loads (events/second).

B. Time and Resource Efficiency of Runtime Event Monitoring. To answer RQ5, we evaluate whether JANUS 's runtime monitoring remains efficient under high event volumes. Using the same setup as earlier post-deployment tests, we simulate 50–400 events per second and measure response time, CPU, and memory usage for Sysdig, Suricata, and JANUS (with and without shallow inspection). For fairness, all tools load only the rules relevant to the 20 CVEs and 5 CVE chains. Fig. 13a shows that JANUS attains the lowest detection latency, reaching 5.9 ms at 400 events/s with shallow inspection, compared to 12.1 ms for Sysdig and 12.8 ms for Suricata. JANUS also uses far less CPU (22%) than Sysdig (84%) or Suricata (61%) (Fig. 13b), with similar memory usage across tools (Fig. 13c). These results show that shallow inspection enables fast, low-overhead monitoring even under high load.

C. Cost Efficiency of Deployment Rent against IPS Tools. To answer RQ6, compare 30-day deployment rental costs for AWS, Azure, and GCP to assess JANUS's cost-efficiency against other IPS tools, both with and without shallow inspection. JANUS, particularly with shallow inspection, is significantly more cost-efficient, especially in CPU usage. Using similar basic plans, CPU costs on AWS are $99.09 (Sysdig), $71.11 (Suricata), and $29.15 (JANUS). On Azure, costs are $99.14, $72.32, and $29.16, respectively, while on GCP, they are $158.63, $113.84, and $46.66. Memory costs are similar across all tools, with slight improvements for JANUS. Overall, JANUS offers nearly 60% CPU cost savings while maintaining comparable memory usage, making it highly cost-effective for container workload monitoring.

5 Discussion

False Positives in Post-deployment Mitigation. During runtime mitigation, false positives (FPs) may occur when benign actions are mistakenly flagged as exploits—an inherent risk in heterogeneous container workloads, triggering unnecessary blocking or investigation, increasing operational overhead and potentially disrupting legitimate services. Although attackers could attempt evasion through log manipulation, cross-cloud attack paths [68], or intentionally slowed exploit sequences [29], such strategies fall outside our current threat model. JANUS mitigates FP risk through shallow-to-deep inspection that filters noise, multi-source (system and network) correlation that adds context, and signature consensus that avoids reliance on a single signature. While FP rates were not measured in this study, future work will evaluate them by running benign workloads and recording false alerts.

Inclusion of Other Sources of Event Logs. While JANUS currently uses only system-call and network events (Sec. 3.3C), additional kernel-level sources (such as file-system changes [56], process events [91], and performance metrics [84]) could help detect broader attack types, including client-side exploits, control-flow manipulation, and DDoS activity. Incorporating these sources is a promising direction for future work.

Leveraging and Integrating Other Security Tools. Our framework currently employs Grype [5] and CVE-Bin-Tool [37] for pre-deployment scanning and uses eBPF [48] to integrate multi-source events at runtime. Although we demonstrate only these tools, the framework can incorporate others (e.g., [15,74]) and more advanced techniques, including ML- or GPT-based analysis [30], to detect a wider range of threats. This flexibility enables JANUS to operate as an extensible security layer alongside existing tools.

Policy Configuration and Practicality. In practice, the required inputs for configuring JANUS are already standard in vulnerability management: such as vulnerability identifiers [17,83], severity and prioritization signals [3,5,15,28,74,79]. Tiered gates suit governance-oriented actions such as *block*, *remediate*, and *monitor*, whereas numeric gates fit workflows using continuous risk scores [14,25,38,80,85]. Though Sec. 3.2 gives a simple example, complex deployments should guide JANUS with internal and/or external risk frameworks. Finally, temporary CVE exceptions can be captured as policy records with identifier, justification, owner, and expiry, matching common ignore and accepted-risk workflows [58,78].

6 Related Works

Pre-deployment Vulnerability Prevention. Vulnerability scanning is widely used pre-deployment, yet multiple studies report blind spots and inconsistencies in traditional scanners [4,50,54,69,76]. Prior work also analyzes container-ecosystem risks, including latent issues in images [59,60], inconsistent registry assessments [53], and dependency-related weaknesses [12]. Practical tools such as

Grype/Trivy/Clair [5, 15, 74] and Dockerfile analyzers [10, 22, 34, 70, 86] improve detection and hygiene, but typically provide passive analysis and limited CI/CD-oriented incremental reuse or remediation. In contrast, our work integrates layer-aware incremental scanning with a policy-controlled patching workflow as part of a coordinated lifecycle approach.

Post-deployment Vulnerability Mitigation. Post-deployment mitigation research explores several strategies for preserving system integrity. System-call whitelisting [11, 18] can disrupt normal workloads, and some approaches require kernel modifications [11], which our design avoids. Container-oriented policy generation [27] and provenance-based monitoring [41] are typically single-source and offer limited correlation across event types. Network-based intrusion detection methods [24, 35, 51] likewise miss system-level behavior and have difficulty capturing multi-step attacks. In contrast, JANUS correlates system and network events via stateful signatures to improve practical detectability of multi-step exploit chains under our threat model.

Hybrid Pre-to-post Solutions. Effective container security requires coverage across both pre- and post-deployment phases [61]. Prior works combine static and dynamic techniques in specific domains (e.g., data-flow protection [64], taint analysis [39], software integrity [45], and memory safety [2]) and highlight cross-environment inconsistencies in CVE interpretation [72]. However, these efforts do not provide a coordinated mechanism that links pre-deployment remediation decisions to post-deployment monitoring across the container lifecycle. In contrast, JANUS couples policy-controlled pre-deployment remediation with stateful, multi-source exploit-chain monitoring for end-to-end lifecycle defense.

7 Conclusion

This paper presented JANUS, a coordinated vulnerability prevention and mitigation framework for CI/CD-native container environments. JANUS uses a configurable *policy gate* to decide which vulnerabilities are remediated pre-deployment versus explicitly accepted and deferred to runtime monitoring. It combines layer-aware incremental scanning with post-deployment, multi-source monitoring that correlates system and network events to detect multi-step exploit chains under curated state-machine signatures. Our evaluation shows that JANUS reduces pre-deployment analysis time via incremental scanning, supports Dockerfile-level patch candidate generation, and detects the evaluated exploit chains with low overhead under our instrumentation.

Limitations and Future Work. JANUS depends on deployment-specific policy gates (numeric or tiered), targets known-vulnerability chains for which signatures/traces are available, and its Dockerfile-based patching is limited to dependency/version updates in the build specification. Moreover, signature creation remains largely manual. Future work includes data-driven policy recommendation, improved automation, GenAI-assisted signature authoring, and extending monitoring beyond known CVEs with behavior-based signals.

Acknowledgement

The authors thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Department of National Defence Canada (DND) under the Discovery Grants RGPIN-2021-04106 and DGDND-2021-04106. The authors also acknowledge support from Mitacs Accelerate in collaboration with Sunphinx Inc.

References

1. Al-Ghuwairi, A.R., Sharrab, Y., Al-Fraihat, D., AlElaimat, M., Alsarhan, A., Algarni, A.: Intrusion Detection in Cloud Computing Based on Time Series Anomalies Utilizing Machine Learning. Journal of Cloud Computing **12**(1), 127 (2023)
2. Alshmrany, K., Bhayat, A., Braußе, F., Cordeiro, L., Korovin, K., Melham, T., Mustafa, M.A., Olivier, P., Reger, G., Shmarov, F.: Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities. In: SecDev. pp. 52–58. IEEE (2022)
3. Amazon Web Services: Amazon ECR Image Scanning. https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-scanning.html, accessed: 2025-12-04
4. Ami, A.S., Moran, K., Poshyvanyk, D., Nadkarni, A.: False Negative-That One Is Going to Kill You: Understanding Industry Perspectives of Static Analysis Based Security Testing. In: S&P. pp. 3979–3997. IEEE (2024)
5. Anchore: Grype. https://github.com/anchore/grype, accessed: 2025-10-05
6. Aqua: Aqua Security. https://github.com/aquasecurity (2025), accessed: 2025-11-25
7. Axelsson, S.: A Survey and Taxonomy of Signature-Based Intrusion Detection Systems. Computing Surveys **35**(4), 294–347 (2003)
8. Brand, A.: Successfully Integrating Dynamic Security Testing into Your CI/CD Pipeline. https://www.infoq.com/articles/integrating-dast-ci-cd/ (Oct 2022), accessed: 2025-11-25
9. Bugcrowd: Bugcrowd: Number One Crowdsourced Cybersecurity Platform. https://www.bugcrowd.com/, accessed: 2025-11-25
10. Bui, Q.C., Laukötter, M., Scandariato, R.: Dockercleaner: Automatic Repair of Security Smells in Dockerfiles. In: ICSME. pp. 160–170. IEEE (2023)
11. Canella, C., Werner, M., Gruss, D., Schwarz, M.: Automating Seccomp Filter Generation for Linux Applications. In: CCSW. pp. 139–151 (2021)
12. Chen, L., Xia, Y., Ma, Z., Zhao, R., Wang, Y., Liu, Y., Sun, W., Xue, Z.: SEAF: A Scalable, Efficient, and Application-Independent Framework for Container Security Detection. JISA **71**, 103351 (2022)
13. Chen, Y., Li, Y., Pan, Z., Lu, Y., Chen, J., Ji, S.: URadar: Discovering Unrestricted File Upload Vulnerabilities via Adaptive Dynamic Testing. TIFS **19**, 1251–1266 (2024). https://doi.org/10.1109/TIFS.2023.3335885
14. CISA: CISA National Cyber Incident Scoring System (NCISS). https://www.cisa.gov/news-events/news/cisa-national-cyber-incident-scoring-system-nciss (Jul 2024), accessed: 2025-11-25
15. CoreOS: Clair. https://coreos.com/clair/, accessed: 2025-10-05

16. Coston, I., Hezel, K.D., Plotnizky, E., Nojoumian, M.: Enhancing secure software development with AZTRM-D: An AI-integrated approach combining devsecops, risk management, and zero trust. Applied Sciences **15**(15), 8163 (2025)
17. Cybersecurity and Infrastructure Security Agency: CISA Known Exploited Vulnerabilities Catalog. https://www.cisa.gov/known-exploited-vulnerabilities-catalog, accessed: 2025-12-04
18. DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., Kemerlis, V.P.: Sysfilter: Automated system call filtering for commodity software. In: RAID. pp. 459–474 (2020)
19. Docker: Docker Hub. https://hub.docker.com/, accessed: 2025-10-05
20. Docker: Docker Container Building Best Practices. https://docs.docker.com/build/building/best-practices/ (Apr 2024), accessed: 2025-11-25
21. Draios: Sysdig. https://github.com/draios/sysdig, accessed: 2024-11-17
22. Durieux, T.: Empirical Study of the Docker Smells Impact on the Image Size. In: ICSE. pp. 1–12. IEEE (2024)
23. Erlacher, F., Dressler, F.: FIXIDS: A High-Speed Signature-Based Flow Intrusion Detection System. In: NOMS. pp. 1–8. IEEE (2018)
24. Erlacher, F., Dressler, F.: On High-Speed Flow-Based Intrusion Detection Using Snort-Compatible Signatures. TDSC **19**(1), 495–506 (2020)
25. First Organization: CVSS - Common Vulnerability Scoring System. https://www.first.org/cvss/ (2023), accessed: 2025-10-05
26. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: S&P. pp. 120–128. IEEE (1996)
27. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated system call policy generation for container attack surface reduction. In: RAID. pp. 443–458 (2020)
28. Google Cloud Platform: Container Analysis: Scanning Types. https://docs.cloud.google.com/artifact-analysis/docs/scanning-types#continuous_analysis, accessed: 2025-12-04
29. Goyal, A., Han, X., Wang, G., Bates, A.: Sometimes, You Aren't What You Do: Mimicry Attacks Against Provenance Graph Host Intrusion Detection Systems. In: NDSS (2023)
30. Gulmezoglu, B., Eisenbarth, T., Sunar, B.: Cache-Based Application Detection in the Cloud Using Machine Learning. In: ACM CCS. pp. 288–300 (2017)
31. HackerOne: HackerOne. https://www.hackerone.com/, accessed: 2025-11-25
32. Hadolint: Hadolint. https://github.com/hadolint/hadolint, accessed: 2025-11-17
33. Han, X., Pasquier, T., Bates, A., Mickens, J., Seltzer, M.: UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. NDSS (2020)
34. Hao, J., Lu, H., Jiang, Y., Gupta, B.B., Almomani, A., Zhang, M., Tian, Z.: DFScan: Security Scanner of the Dockerfile Based on Instruction Coverage and Attack Perspective. Human-centric Computing and Information Sciences **14** (2024)
35. He, J., Tang, C., Li, W., Li, T., Chen, L., Lan, X.: BR-HIDF: An Anti-Sparsity and Effective Host Intrusion Detection Framework Based on Multi-Granularity Feature Extraction. TIFS (2023)
36. He, K., Kim, D.D., Asghar, M.R.: Adversarial Machine Learning for Network Intrusion Detection Systems: A Comprehensive Survey. ICST **25**(1), 538–566 (2023)
37. Intel: CVE Bin Tool. https://github.com/intel/cve-bin-tool, accessed: 2025-10-05
38. Jacobs, J., Romanosky, S., Edwards, B., Adjerid, I., Roytman, M.: Exploit Prediction Scoring System (EPSS). Digital Threats: Research and Practice **2**(3), 1–17 (2021)

39. Ji, M., Yin, M., Zhou, Y.H.: Application of Static Taint Analysis in Rasp Protection Strategy. In: Proceedings of the 2022 International Conference on Cyber Security. pp. 40–45 (2022)
40. Kaur, B., Dugré, M., Hanna, A., Glatard, T.: An Analysis of Security Vulnerabilities in Container Images for Scientific Data Analysis. GigaScience **10**(6), giab025 (2021)
41. Kermabon-Bobinnec, H., Jarraya, Y., Wang, L., Majumdar, S., Pourzandi, M.: Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences. NDSS (2023)
42. Khraisat, A., Gondal, I., Vamplew, P., Kamruzzaman, J.: Survey of Intrusion Detection Systems: Techniques, Datasets and Challenges. Cybersecurity **2**(1), 1–22 (2019)
43. Kittler, J., Hatef, M., Duin, R.P., Matas, J.: On combining classifiers. IEEE transactions on pattern analysis and machine intelligence **20**(3), 226–239 (2002)
44. Krishnaveni, S., Sivamohan, S., Sridhar, S., Prabakaran, S.: Efficient Feature Selection and Classification Through Ensemble Method for Network Intrusion Detection on Cloud Computing. Cluster Computing **24**(3), 1761–1779 (2021)
45. Kucab, M., Boryło, P., Chołda, P.: Hardware-Assisted Static and Runtime Attestation for Cloud Deployments. IEEE Transactions on Cloud Computing (2023)
46. Kwon, S., Lee, J.H.: DIVDS: Docker Image Vulnerability Diagnostic System. IEEE access **8**, 42666–42673 (2020)
47. Li, W., Zhang, X.Y., Bao, H., Yang, B., Li, Z., Shi, H., Wang, Q.: Prism: Real-Time Privacy Protection Against Temporal Network Traffic Analyzers. TIFS **18**, 2524–2537 (2023)
48. Linux: ebpf. https://ebpf.io/what-is-ebpf/, accessed: 2025-11-17
49. Linux: Ptrace. https://man7.org/linux/man-pages/man2/ptrace.2.html, accessed: 2025-11-17
50. Liu, P., Ji, S., Fu, L., Lu, K., Zhang, X., Lee, W.H., Lu, T., Chen, W., Beyah, R.: Understanding the Security Risks of Docker Hub. In: ESORICS. pp. 257–276. Springer (2020)
51. Long, Z., Yan, H., Shen, G., Zhang, X., He, H., Cheng, L.: A Transformer-Based Network Intrusion Detection Approach for Cloud Security. JCC **13**(1), 5 (2024)
52. Majumder, S.H., Jajodia, S., Majumdar, S., Hossain, M.S.: Layered Security Analysis for Container Images: Expanding Lightweight Pre-Deployment Scanning. In: PST. pp. 1–10. IEEE (2023)
53. Malhotra, R., Bansal, A., Kessentini, M.: Vulnerability Analysis of Docker Hub Official Images and Verified Images. In: SOSE. pp. 150–155. IEEE (2023)
54. Mambretti, A., Convertini, P., Sorniotti, A., Sandulescu, A., Kirda, E., Kurmus, A.: Ghostbuster: Understanding and overcoming the pitfalls of transient execution vulnerability checkers. In: SANER. pp. 307–317. IEEE (2021)
55. Martin, A., Hernandez-Ramos, J.L., López, J.: A survey on container security: Challenges and directions. IEEE Communications Surveys & Tutorials **22**(2), 1192–1230 (2020)
56. Matos, D.R., Pardal, M.L., Carle, G., Correia, M.: RockFS: Cloud-Backed File System Resilience to Client-Side Attacks. In: Middleware. pp. 107–119 (2018)
57. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux Journal (239), 2 (2014)
58. Microsoft Learn: Exempt resources from recommendations. https://learn.microsoft.com/en-us/azure/defender-for-cloud/exempt-resource, accessed: 2026-04-10

59. Mills, A., White, J., Legg, P.: Longitudinal Risk-Based Security Assessment of Docker Software Container Images. Computers & Security **135**, 103478 (dec 2023). https://doi.org/10.1016/j.cose.2023.103478, https://doi.org/10.1016/j.cose.2023.103478
60. Mills, A., White, J., Legg, P.: Longitudinal Risk-Based Security Assessment of Docker Software Container Images. Computers & Security **129**, 103151 (2023)
61. Minna, F., Massacci, F.: SOK: Run-Time Security for Cloud Microservices. Are We There Yet? Computers & Security **127**, 103119 (2023)
62. Mubanda, D., Mandela, N., Mbinda, T., Ayesiga, C.: Evaluating Docker Container Security through Penetration Testing: A Smart Computer Security. In: ICCSAI. pp. 415–419. IEEE (2023)
63. OISF: Suricata. https://suricata.io/, accessed: 2025-11-17
64. Palit, T., Moon, J.F., Monrose, F., Polychronakis, M.: Dynpta: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In: S&P. pp. 1919–1937. IEEE (2021)
65. ProjectDiscovery: Nuclei. https://github.com/projectdiscovery/nuclei, accessed: 2025-11-25
66. Rahman, A., Williams, L.: Assessing adoption of devsecops practices in industry. In: S&P Workshop (2016)
67. Rapid7: Metasploit Framework. https://www.metasploit.com/, accessed: 2025-12-04
68. Reece, M., Rastogi, N., Lander, T., Dykstra, J., Mittal, S., Sampson, A.: Defending Multi-Cloud Applications Against Man-in-the-Middle Attacks. In: ACM SACMAT. pp. 47–52 (2024)
69. Remigio, R., Bertani, A., Polino, M., Carminati, M., Zanero, S.: The good, the bad, and the binary: An lstm-based method for section boundary detection in firmware analysis. In: International Workshop on Security. pp. 16–35. Springer (2023)
70. Rosa, G., Zappone, F., Scalabrino, S., Oliveto, R.: Fixing Dockerfile Smells: An Empirical Study. Empirical Software Engineering **29**(5), 108 (2024)
71. Rushdi, A.M.: Reliability of K-Out-Of-N Systems. In: Fundamental Studies in Engineering, vol. 16, pp. 185–227 (1993)
72. Schloegel, M., Klischies, D., Koch, S., Klein, D., Gerlach, L., Wessels, M., Trampert, L., Johns, M., Vanhoef, M., Schwarz, M., et al.: Confusing value with enumeration: Studying the use of {CVEs} in academia. In: USENIX Security. pp. 2887–2906 (2025)
73. Schmitt, J.: SAST vs DAST: What They Are and When to Use Them. https://circleci.com/blog/sast-vs-dast-when-to-use-them/ (Apr 2022), accessed: 2025-11-25
74. Security, A.: Trivy. https://github.com/aquasecurity/trivy, accessed: 2025-10-05
75. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access **7**, 111177–111196 (2019)
76. Shu, R., Gu, X., Enck, W.: A Study of Security Vulnerabilities on Docker Hub. In: CODASPY. pp. 269–280. IEEE (2017)
77. Shu, X., Yao, D., Ryder, B.G.: A Formal Framework for Program Anomaly Detection. In: RAID. pp. 270–292. Springer (2015)
78. Snyk: Ignore issues. https://docs.snyk.io/manage-risk/prioritize-issues-for-fixing/ignore-issues, accessed: 2026-04-10
79. Snyk: Snyk. https://snyk.io/product/container-vulnerability-management/, accessed: 2025-10-05

80. Spring, J., Hatleback, E., Householder, A., Manion, A.: Prioritizing Vulnerability Response: A Stakeholder-Specific Vulnerability Categorization (SSVC). Tech. rep., Carnegie Mellon University, Software Engineering Institute (SEI) (2023)
81. of Standards, N.I., Technology: NVD Dashboard — CVEs Received and Processed. https://nvd.nist.gov/general/nvd-dashboard (2025), accessed: 2025-11-17
82. Sysdig: Falco. https://falco.org/, accessed: 2025-11-17
83. The MITRE Corporation: CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/ (2023), accessed: 2025-10-05
84. Thummapudi, K., Lama, P., Boppana, R.V.: Detection of Ransomware Attacks Using Processor and Disk Usage Data. IEEE Access **11**, 51395–51407 (2023)
85. Trend Micro: Trend Micro Risk Score. https://resources.trendmicro.com/rs/945-CXD-062/images/MORE%20THAN%20A%20NUMBER-YOUR%20RISK%20SCORE%20EXPLAINED.pdf, accessed: 2025-11-25
86. Vassallo, C., Proksch, S., Jancso, A., Gall, H.C., Di Penta, M.: Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab. In: ACM ESEC/FSE. pp. 327–337 (2020)
87. Vulhub: vulhub/vulhub: Pre-Built Vulnerable Environments Based on Docker-Compose. https://github.com/vulhub/vulhub, accessed: 2025-11-25
88. Xu, Z., Fang, P., Liu, C., Xiao, X., Wen, Y., Meng, D.: Depcomm: Graph Summarization on System Audit Logs for Attack Investigation. In: S&P. pp. 540–557. IEEE (2022)
89. Zhang, K., Wang, S., Chen, H., Li, X.: bpftime: Realtime ebpf-based system event monitoring for security applications. In: ACM AsiaCCS. pp. 587–600. ACM (2022)
90. Zhao, L., Li, P., Chen, F.: Understanding Docker Image Layering and Build Caching Mechanisms. In: CloudCom. pp. 190–197. IEEE (2020)
91. Zhu, T., Yu, J., Xiong, C., Cheng, W., Yuan, Q., Ying, J., Chen, T., Zhang, J., Lv, M., Chen, Y., et al.: APTSHIELD: A Stable, Efficient and Real-Time Apt Detection System for Linux Hosts. TDSC **20**(6), 5247–5264 (2023)

A Janus System Design

A.1 Implementational Challenges

Implementation of Pre-Deployment Prevention. The main implementation challenge is coordinating heterogeneous scanners over frequently changing container layers while maintaining speed and consistency. Images are exported and parsed to extract SHA256 layer IDs, which drive both layer classification and incremental scanning. Each layer is classified by inspecting its internal file structure, enabling metadata-based SBOM analysis for base layers and content-based scanning for package layers. Incremental scanning then reuses stored reports for repeated layers and processes only new ones. SBOM normalization ensures consistent package and version matching across scanners.

Implementation of Post-Deployment Mitigation. The main challenge in post-deployment mitigation is correlating high-volume, multi-source events into compact exploit signatures while supporting millisecond-level runtime enforcement. Controlled exploit runs generate system-call and network traces, which are merged into a unified stream with container metadata. An audit-trail generator

orders and groups these events to assist expert-guided root-cause analysis. A sequence abstractor then converts the traces into multi-step exploit signatures by constructing minimal state machines that preserve essential event types, arguments, and cross-source relations. At runtime, signatures are loaded into eBPF, where lightweight inspectors attached to kernel probes and traffic hooks perform shallow metadata filtering. Only filtered candidates undergo deep argument checks and state transitions. Consensus across signatures determines whether an exploit is active, triggering the configured enforcement action. This pipeline enables precise, low-overhead detection of complex exploit chains without modifying the kernel, runtime, or applications.

A.2 System Design

Fig. 14 presents the high-level design of JANUS, consisting of two major modules: pre-deployment prevention and post-deployment mitigation. The pre-deployment module performs layer-aware vulnerability analysis by classifying image layers, reusing scan results across versions, selectively applying metadata- and content-based scanners, and automatically patching vulnerable packages before screening images for deployment. The post-deployment module constructs multi-step exploit signatures by analyzing system and network events from controlled exploit runs, abstracting them into compact state machines, and enforcing them at runtime via eBPF-based shallow and deep inspectors. Together, these modules form a unified workflow that minimizes residual risk before deployment and blocks exploit escalation after deployment, maintaining continuous vulnerability mitigation across the container lifecycle.

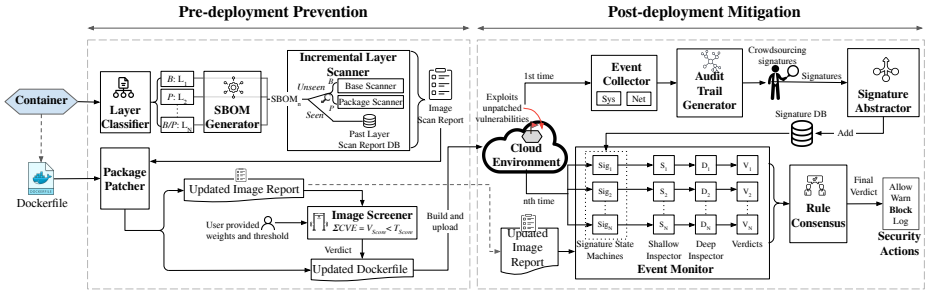
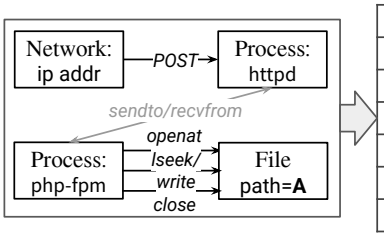


Fig. 14: Implementation overview of JANUS.

B Case Study: Signature Creation for Individual CVEs

This case study illustrates how JANUS creates effective signatures for individual CVEs, not just exploit chains. We analyze CVE-2021-34074, an arbitrary file-upload vulnerability in Pandora FMS (≤7.54) that allows attackers to exploit

relative-path handling in the file-manager module to upload malicious files and achieve remote command execution. Existing IPS tools struggle here: Suricata cannot reliably detect the exploit because `'..'` patterns generate excessive false positives and upload responses always return `200 OK`, while Sysdig flags common file-creation system calls that also occur during normal operation, making benign and malicious behavior difficult to separate.



#	Type	Event	Ret/Header	Args1	Args2
1	net	POST		/file_manager.php	path=**A**
2	sys	stat		path=**A**	
3	sys	openat	fd=**B**	name=**A**	O_CREAT
4	net	lseek		fd=**B**	
5	sys	write		fd=**B**	
6	sys	close	0	fd=**B**	

Fig. 15: CVE-2021-34074 with provenance graph (left) and event signature identified by JANUS (right).

JANUS overcomes these gaps by correlating network and system events. Suspicious `POST` requests related to file uploads are first identified in network logs, after which JANUS traces the resulting sequence of file-handling system calls (`stat`, `openat` with `O_CREAT`, `lseek`, `write`, and `close`) while tracking file paths and descriptors (Events 2–6). This multi-source, multi-step signature captures attacker intent more precisely, reduces false positives, and reliably detects this vulnerability where existing IPS solutions fall short.