Security Challenges and Solution in Kubernetes Cluster Management

Neeraj Sharma[1], Chandrabhan[2], Tanay Sinha[3], Gopal Khorwal[4]
[1,2,3] *Master of Computer Applications (Semester IV), Dept. of Computer Science, Jaipur National University, Jaipur, Rajasthan, India*
[4] *Assistant Professor, Dept. of Computer Science, Jaipur National University, Jaipur, Rajasthan, India*

Abstract — Kubernetes has rapidly emerged as the dominant platform for container orchestration in enterprise and cloud-native environments. While the platform offers compelling benefits in terms of workload automation and operational scalability, its inherent architectural complexity gives rise to a significant and frequently underestimated security exposure. This study investigates five categories of security vulnerabilities commonly observed in production Kubernetes deployments: improperly scoped Role-Based Access Control (RBAC) configurations, publicly reachable API server endpoints, container images carrying unpatched known vulnerabilities, inadequately protected inter-pod network channels, and insecure handling of sensitive configuration data. Rather than cataloguing these issues in isolation, the paper proposes a cohesive, five-layer security framework built around the principle of defense in depth. The framework brings together proven open-source tooling — Trivy for vulnerability scanning, Cosign for supply chain integrity, OPA/Gatekeeper for policy enforcement, Falco for runtime behavioral monitoring, HashiCorp Vault for secrets management, and Istio for encrypted service communication — into a structured, actionable security posture. The framework was evaluated against a simulated cluster environment using representative attack scenarios drawn from the MITRE ATT&CK for Containers matrix, yielding measurable improvements in threat containment and access boundary enforcement. The findings are intended to provide a practically grounded reference for DevOps engineers, security practitioners, and researchers engaged in cloud-native infrastructure protection.

Keywords — Kubernetes Security, Container Orchestration, RBAC, DevSecOps, Pod Security Admission, Service Mesh, Runtime Threat Detection, Cloud-Native Security

I. INTRODUCTION

Software delivery architecture has undergone a structural transformation over the past decade. The shift away from tightly coupled monolithic systems toward loosely coupled, independently deployable microservices has been one of the most consequential changes in how engineering teams build and operate production software. Containers — lightweight, portable execution environments — became the natural packaging unit for these microservices, and their widespread adoption was accelerated significantly by the introduction of Docker in 2013. However, once organizations began operating containerized workloads at scale, the operational challenges of coordinating many containers across many machines quickly became evident.

Kubernetes was developed at Google as an internal orchestration system and publicly released in 2014, later finding a permanent home within the Cloud Native Computing Foundation (CNCF). Its core value proposition — declarative configuration, automated scheduling, self-healing, and horizontal scalability — addressed exactly the coordination problems that large-scale container deployments created. Industry adoption has been remarkably swift. As reported in the CNCF Annual Survey 2023, approximately 96% of surveyed organizations were either actively running Kubernetes in production or evaluating it for near-term adoption [1]. These figures position Kubernetes not as a niche DevOps tool but as a foundational piece of modern infrastructure.

What tends to receive less attention in these discussions, however, is the security dimension. Kubernetes was designed with operational resilience

and developer productivity as its primary objectives; security was not a first-class design requirement. The consequence is that many default configurations are permissive in ways that would be considered unacceptable in other enterprise systems. A freshly deployed cluster, absent deliberate hardening, allows any pod to reach any other pod across any namespace, permits service accounts to accumulate excessive API privileges, and stores configuration data — including sensitive credentials — in etcd without encryption at rest. These are not obscure edge cases. According to the 2022 Red Hat State of Kubernetes Security Report, more than half of surveyed practitioners reported a security incident attributable to Kubernetes misconfiguration within the preceding year [2].

The architecture of a production Kubernetes cluster, as depicted in Fig. 1, involves several distinct components — the API server, scheduler, controller manager, etcd, and per-node kubelet processes — each of which presents its own threat surface. Understanding how these components interact, and where the trust boundaries between them lie, is foundational to building a coherent security strategy.

Fig. 1 — Kubernetes Cluster Architecture

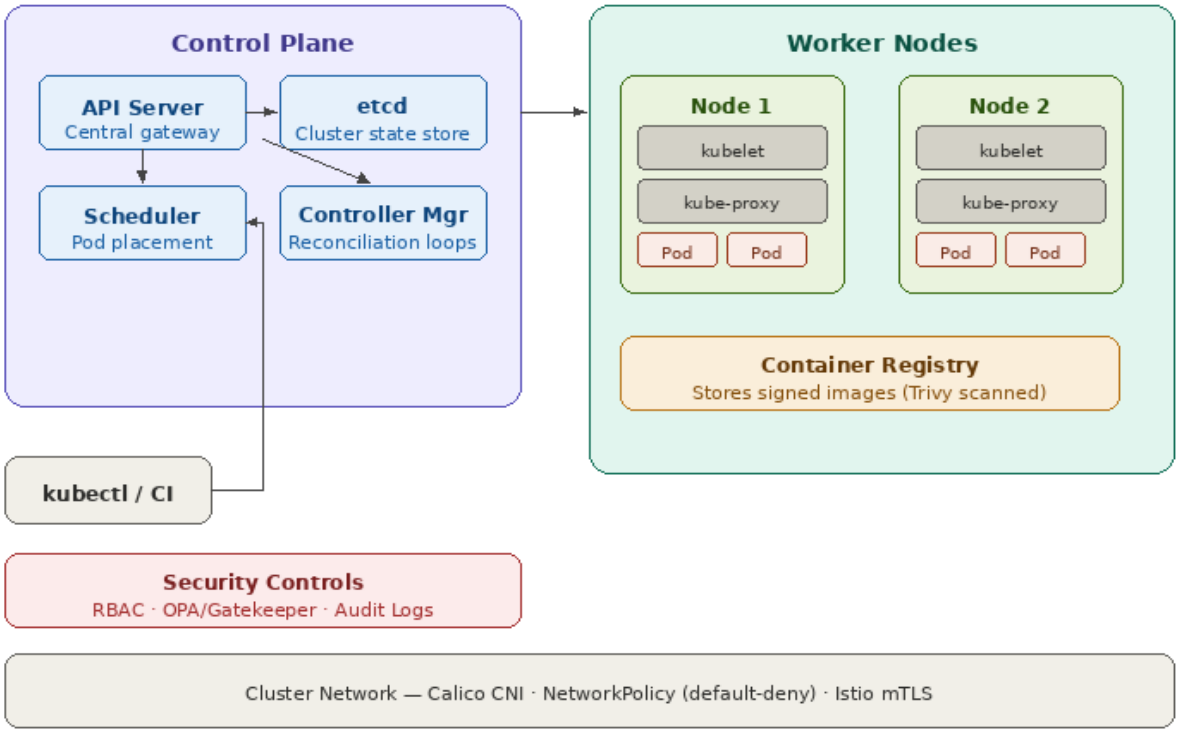

Fig. 1. Kubernetes Cluster Architecture with Security Components

Against this backdrop, the present paper proposes a structured, multi-layer security framework designed to address the most critical vulnerability categories in Kubernetes cluster management. The framework draws on open-source tooling, guidance from the NSA and CISA [10], and empirical evaluation against simulated attack scenarios. Section II surveys relevant prior work; Section III describes the proposed framework; Section IV covers implementation specifics; Section V presents evaluation results; and Section VI discusses conclusions and future research directions.

II. LITERATURE REVIEW / RELATED WORK

Academic and industry interest in Kubernetes security has expanded considerably since around 2018, broadly tracking the platform's commercial adoption. Early investigations were largely descriptive, documenting the architecture and cataloguing the types of threats

that various components could face. Rice (2020) produced one of the first comprehensive treatments of container security, organizing the threat landscape across four domains: supply chain integrity, cluster configuration, workload runtime behavior, and image provenance [3]. This taxonomy has since influenced how practitioners and researchers frame the problem space.

A more fundamental security concern — one that distinguishes containers from virtual machines — was examined by Sultan et al. (2019) in a systematic survey published in IEEE Access [4]. Their central observation was that container isolation rests on Linux kernel primitives (namespaces and cgroups) rather than on the hardware-mediated boundaries provided by a hypervisor. A successful container escape therefore bypasses the abstraction layer entirely, granting an attacker direct access to the host operating system. This has profound implications for multi-tenant Kubernetes environments where workloads from different security domains share physical nodes.

The specific problem of access control misconfiguration has received focused empirical scrutiny. Itkis and Reyzin (2021) collected and analyzed Kubernetes configuration manifests from publicly accessible GitHub repositories, finding that a disproportionate number of cluster role bindings assigned cluster-admin or similarly broad permissions to application-level service accounts [5]. The root cause, they argued, was a tendency to copy-paste configurations from online tutorials — which frequently use permissive settings for simplicity — rather than a lack of security awareness. This observation directly informed the enforcement mechanisms in the present framework.

Image vulnerability exposure was quantified by Shu et al. (2020), whose scanning study of images in the official Docker Hub registry revealed that high-severity CVEs were present in the majority of surveyed images [6]. Critically, many of these vulnerabilities had available patches at the time of discovery; the images simply had not been updated. This finding established a compelling evidence base for mandatory pre-deployment scanning as a required gate in any CI/CD pipeline operating on containerized workloads.

Runtime detection has emerged as a distinct research strand. Lorenz et al. (2022) evaluated Falco, a rule-based behavioral monitoring tool that uses eBPF to intercept system calls at the container boundary [7]. Their study found that, with appropriately tuned rule sets, Falco achieved high detection rates for common attack categories — privilege escalation, unexpected file access, and anomalous outbound connections — while maintaining false-positive rates low enough to be operationally practical. The trade-off between rule specificity and coverage is directly relevant to the runtime monitoring layer described later in this paper.

Network-layer security has been approached through the service mesh paradigm. Burns et al. (2019) examined sidecar-proxy architectures — specifically Istio and Linkerd — as a mechanism for transparently enforcing mutual TLS between all inter-service calls [8]. Their analysis demonstrated that a properly configured service mesh substantially reduces the value of network-level eavesdropping and lateral movement, since even an attacker with a cluster foothold cannot read unencrypted service traffic. A notable gap in the existing literature is the absence of a unified framework integrating these independently validated approaches into a single, coherently layered defensive posture — a gap the present work directly addresses.

III. METHODOLOGY / PROPOSED SYSTEM

The security framework developed in this study is grounded in the defense-in-depth principle, which holds that meaningful resilience requires layered, partially redundant protections spanning the full system lifecycle. Applied to Kubernetes, this means addressing security not only at the cluster boundary but at each successive layer of the stack — from container image provenance, through access governance and workload configuration, to network communications and live process behavior. Fig. 2 depicts this layered structure.

Fig. 2 — Proposed 5-Layer Defense-in-Depth Security Framework

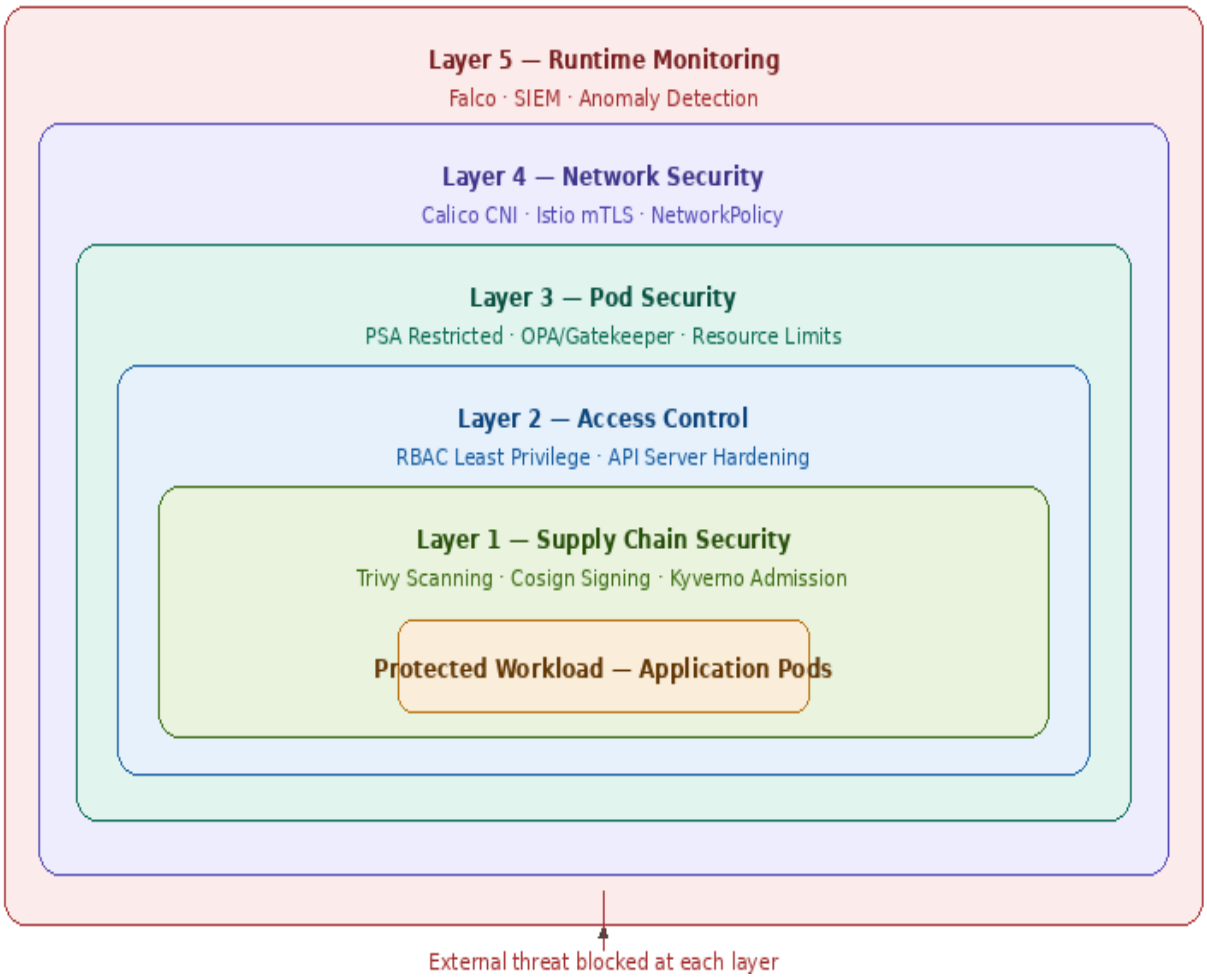

Fig. 2. Proposed 5-Layer Defense-in-Depth Security Framework

A. Layer 1: Supply Chain Integrity and Image Security

The security lifecycle of a containerized workload begins before the container starts running — at the point where the image is built. Any vulnerability baked into an image at build time will be present in every derived container instance, regardless of runtime hardening. Accordingly, the framework mandates integration of Trivy — an open-source vulnerability scanner maintained by Aqua Security — as a mandatory gate within the CI/CD pipeline. Any image containing a critical or high-severity CVE causes the pipeline to fail. Additionally, all images passing the scan must be cryptographically signed using Cosign, a tool from the Sigstore project that anchors image identity to a verifiable chain of custody. A Kyverno admission policy enforces at the cluster level that only signed, verified images may be scheduled.

B. Layer 2: Access Governance and API Hardening

When misconfigured, Kubernetes RBAC represents one of the most direct paths to full cluster compromise. The framework adopts a strict least-privilege posture: each service account receives only the verb-resource combinations its associated workload actively requires, scoped to the relevant namespace rather than granted cluster-wide wherever possible. ClusterRoleBindings assigning wildcard permissions are explicitly prohibited through OPA/Gatekeeper ConstraintTemplates. At the API server level, anonymous authentication is disabled, all endpoints

are TLS-protected, and a comprehensive audit policy logs API calls for forensic and compliance purposes.

C. Layer 3: Workload and Pod-Level Hardening

Kubernetes Pod Security Admission (PSA), introduced in v1.25 as the supported replacement for the deprecated PodSecurityPolicy, provides a built-in mechanism for enforcing workload security standards. The framework applies the "restricted" PSA profile as the default across all production namespaces. Under this profile, containers cannot run as root, cannot request privilege escalation, must specify a non-zero user ID, and are denied access to host namespaces and sensitive volume types. Container resource limits are enforced through a separate OPA/Gatekeeper policy to prevent resource exhaustion attacks affecting co-located workloads.

D. Layer 4: Network Segmentation and Encrypted Communication

Default Kubernetes networking permits unrestricted pod-to-pod communication across all namespaces — a flat topology that significantly expands the blast radius of any compromised workload. The framework deploys Calico as the CNI plugin and establishes a default-deny NetworkPolicy in every namespace, with explicit ingress and egress allowances defined only for validated communication paths. Overlaid on this L3/L4 enforcement, Istio is configured to provide transparent mutual TLS for all service-to-service calls, eliminating credential theft through passive network interception.

E. Layer 5: Runtime Behavioral Monitoring

Preventive controls reduce the probability of a successful intrusion but cannot guarantee its prevention. The fifth layer addresses detection and response. Falco is deployed as a privileged DaemonSet on every worker node, using eBPF probes to intercept system calls and evaluate them against a behavioral rule library. The rule set covers the most consequential attack behaviors in the MITRE ATT&CK for Containers framework: execution of unexpected binaries within a container, reads of credential files outside the application working directory, connections to unexpected external addresses, and container drift — the appearance of executables not present in the original image filesystem. All Falco alerts, together with Kubernetes audit log events, are forwarded to a centralized SIEM for correlation and escalation.

IV. SYSTEM ARCHITECTURE / IMPLEMENTATION

The framework was deployed on a test cluster provisioned using kubeadm, comprising a single control plane node and three worker nodes, each running Ubuntu 22.04 LTS with Kubernetes v1.28. This configuration reflects a representative on-premises deployment scenario rather than a managed cloud service, where some hardening steps would be handled by the provider. Fig. 4 traces the end-to-end CI/CD security pipeline from source commit to scheduled workload.

A. CI/CD Pipeline Security Integration

The pipeline was implemented using GitHub Actions. A Trivy scan workflow triggers automatically on each pull request, running against the candidate container image built from the branch. Images containing critical or high-severity CVEs block the pipeline from proceeding; those that pass are signed using Cosign, with the signing key stored as a GitHub Actions encrypted secret. On the cluster side, a Kyverno ClusterPolicy verifies the Cosign signature of every image reference in an incoming pod specification, rejecting pods whose images lack a valid signature. This creates a hard link between the CI/CD approval process and what the cluster will actually schedule.

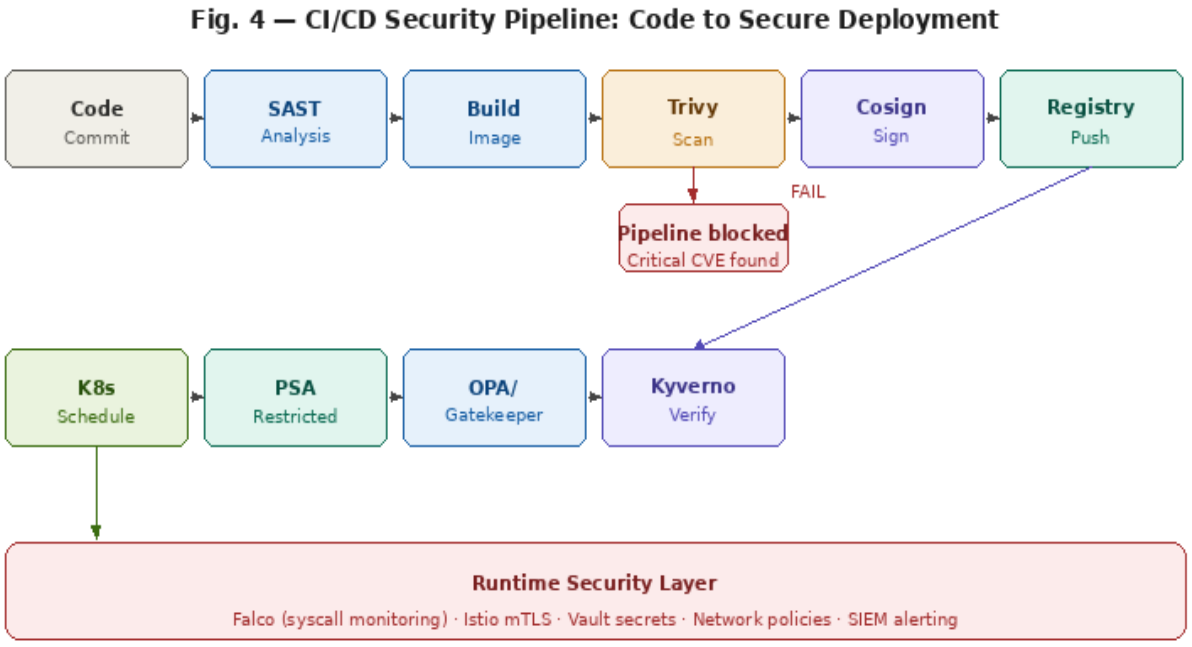

Fig. 4. CI/CD Security Pipeline from Code Commit to Secure Deployment

B. RBAC Scoping and Audit Automation

The default service account in each namespace was configured with automountServiceAccountToken set to false, requiring workloads that genuinely need API access to declare this explicitly and be assigned a dedicated, scoped account. Role definitions use specific resource type and verb lists — no wildcards permitted. To detect role binding drift over time, a weekly CronJob runs the rbac-lookup and kubectl-who-can utilities and compares output against a reference manifest stored in the cluster's GitOps repository.

C. OPA/Gatekeeper Policy Enforcement

Four ConstraintTemplates were deployed to complement PSA controls: one prohibiting containers with privileged: true in their security context; one requiring explicit CPU and memory limits on every container; one rejecting image references using the "latest" tag to enforce deterministic versioning; and one mandating cost-attribution labels on all namespaces. Violations cause the admission webhook to return a rejection response, and details are captured in the audit log.

D. Secrets Management Architecture

Kubernetes-native Secret objects were restricted to non-sensitive configuration data. All credentials, API keys, and certificates were migrated to HashiCorp Vault, integrated via the Vault Agent Injector sidecar. Applications access secrets at runtime through a short-lived, Vault-generated token bound to a Kubernetes service account identity. Secrets are injected into the application's memory space via a mounted in-memory volume and are never written to disk or stored in etcd in recoverable form.

TABLE I provides a consolidated view of the security challenges addressed by the framework, their assessed risk severity, and the specific mitigations applied.

V. RESULTS AND DISCUSSION

Framework effectiveness was assessed through a structured evaluation comparing two cluster configurations: an unmodified baseline running default Kubernetes settings, and a hardened configuration with all five framework layers applied. Attack scenarios were selected from the MITRE ATT&CK for Containers matrix [9], providing a principled basis for scenario design and a shared vocabulary for describing observed behaviors. The comparative attack chains are visualized in Fig. 3.

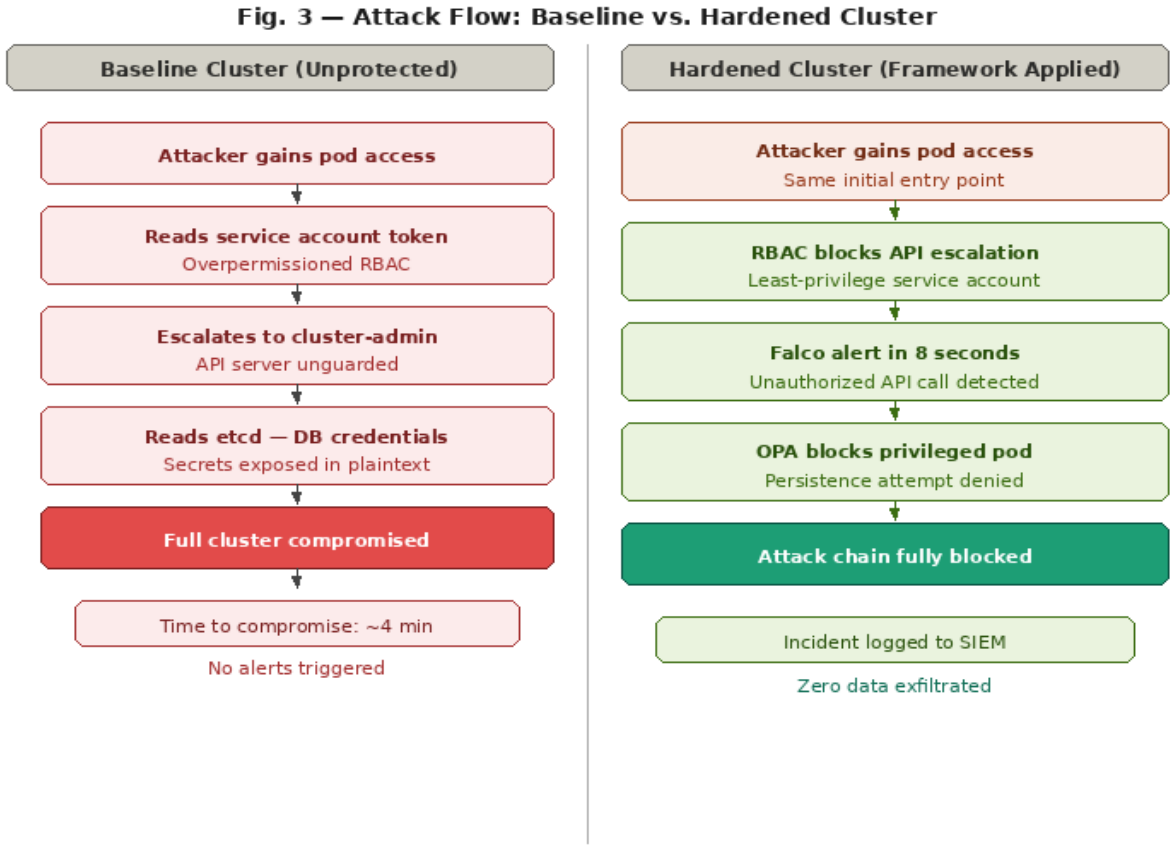

Fig. 3. Comparative Attack Flow: Baseline Cluster vs. Hardened Cluster

The first scenario simulated a credential-theft and privilege escalation attack originating from a compromised application pod. In the baseline cluster, the attacker read the service account token mounted at the default path, used it to authenticate to the API server, enumerated available ClusterRoleBindings, and within approximately four minutes had escalated to cluster-admin privileges, accessed etcd, and retrieved Kubernetes Secret objects containing plaintext database credentials. No component of the baseline cluster generated an alert at any stage.

The identical attack sequence replayed against the hardened cluster was interrupted at the first substantive step. The compromised pod's service account had been granted only the minimal permissions needed by the legitimate application; API enumeration attempts returned authorization errors. Falco detected the anomalous API server calls within eight seconds and generated a high-severity alert forwarded to the SIEM. A secondary persistence attempt — deploying a privileged pod — was rejected by the OPA/Gatekeeper admission webhook before the pod specification was even persisted to etcd.

The second scenario tested supply chain controls by attempting to deploy an image known to contain CVE-2021-44228 (the Log4Shell remote code execution vulnerability). In the baseline environment, the image was accepted by the scheduler and ran without restriction. In the hardened environment, the image had not passed the CI/CD Trivy gate and lacked a valid Cosign signature; the Kyverno admission policy intercepted the pod creation request and rejected it with a specific policy violation message.

Across a broader set of ten adversarial scenarios, Falco's detection capability was measured against the full framework. Eight of the ten scenarios were detected, producing alerts with sufficient contextual information to reconstruct the attack sequence. The two undetected scenarios involved slow-rate, low-volume data exfiltration remaining within thresholds

associated with normal application traffic — a recognized limitation of syscall-based detection that network-layer anomaly detection could potentially address in future iterations of the framework.

The resource cost of the full framework warrants honest acknowledgment. CPU utilization across worker nodes increased by approximately 8 to 12 percent relative to the baseline, with Istio's Envoy sidecar proxies accounting for most of that overhead. For latency-sensitive or resource-constrained deployments, selective application of framework components — prioritizing RBAC hardening, image scanning, and Falco as a minimum viable posture — may be more pragmatic than full deployment from the outset.

VI. CONCLUSION AND FUTURE SCOPE

Kubernetes has transformed how organizations deploy and manage distributed software systems, but the same architectural properties that make it powerful also make it a complex security target. This paper documented five principal categories of vulnerability present in typical Kubernetes deployments and proposed a five-layer defense-in-depth framework addressing each category through preventive controls, policy enforcement, encrypted communications, and behavioral monitoring. Experimental evaluation demonstrated that the framework successfully blocked two representative multi-stage attack chains that succeeded without resistance against a baseline cluster, and achieved an 80% detection rate across a broader adversarial test set.

Several aspects merit further investigation. Integrating GitOps principles — routing all security policy changes through version-controlled pull requests with automated compliance checks before deployment — would strengthen auditability and reduce configuration drift between declared and live cluster state. Tools such as ArgoCD or Flux, combined with policy-as-code repositories, offer a natural implementation path. On the detection side, the slow-rate exfiltration gap points to a need for network-layer behavioral modeling operating above the syscall level. Applying unsupervised anomaly detection to Istio's telemetry stream — connection volume, request latency distributions, payload size patterns — could provide a complementary detection signal without requiring manual rule authorship.

Looking further ahead, the expansion of Kubernetes into multi-cluster and hybrid-cloud topologies introduces a new generation of security challenges not addressed in this paper. When workloads span clusters across different providers or administrative domains, the assumptions underlying single-cluster RBAC and NetworkPolicy enforcement no longer hold. Identity federation across cluster boundaries, consistent policy propagation, and unified audit visibility across heterogeneous control planes are open research problems with significant practical consequence that the authors intend to pursue in subsequent work.

REFERENCE

[1] Cloud Native Computing Foundation, "CNCF Annual Survey 2023," CNCF, San Francisco, CA, USA, Tech. Rep., 2023. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/

[2] Red Hat, Inc., "State of Kubernetes Security Report 2022," Red Hat, Raleigh, NC, USA, Industry Rep., 2022. [Online]. Available: https://www.redhat.com/en/resources/state-kubernetes-security-report

[3] L. Rice, "Container Security: Fundamental Technology Concepts that Protect Containerized Applications," O'Reilly Media, Sebastopol, CA, USA, 2020.

[4] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," IEEE Access, vol. 7, pp. 52514–52535, 2019. doi: 10.1109/ACCESS.2019.2911732

[5] G. Itkis and L. Reyzin, "Empirical Analysis of RBAC Misconfiguration in Public Kubernetes Deployments," in Proc. IEEE Int. Conf. Cloud Computing (CLOUD), Chicago, IL, USA, 2021, pp. 341–349.

[6] X. Shu, J. Ciber, and D. D. Yao, "Study of Security Vulnerabilities in Docker Hub Images," in Proc. ACM Symp. Access Control Models and Technologies (SACMAT), 2020, pp. 97–108.

[7] M. Lorenz, P. Twardowski, and J. Serna, "Runtime Security Monitoring in Kubernetes Using eBPF-Based System Call Interception," in Proc. IEEE

Int. Conf. Cloud Engineering (IC2E), San Francisco, CA, USA, 2022, pp. 118–127.

[8] B. Burns, E. Beda, K. Hightower, and L. Lentchner, "Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications," O'Reilly Media, Sebastopol, CA, USA, 2019.

[9] MITRE Corporation, "ATT&CK for Containers," MITRE ATT&CK, 2023. [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/

[10] National Security Agency (NSA) and Cybersecurity and Infrastructure Security Agency (CISA), "Kubernetes Hardening Guidance," NSA/CISA, Washington, DC, USA, Tech. Rep. v1.2, Aug. 2022. [Online]. Available: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF

[11] A. Martin, S. Yusupov, and R. Pell, "Measuring the Security Posture of Open Source Kubernetes Configurations," in Proc. USENIX Security Symposium, 2023, pp. 1421–1438.

[12] D. Walsh and V. Jain, "Seccomp Profiles and AppArmor Policies for Container Workload Isolation," Linux Journal, vol. 2021, no. 4, pp. 18–27, 2021.