RBAClock: Contain RBAC Permissions through Secure Scheduling

Qingwang Chen
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
chenqingwang@iie.ac.cn

Ru Tan
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
tanru@iie.ac.cn

Xinyu Liu
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
liuxinyu@iie.ac.cn

Yuqi Shu
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
shuyuqi@iie.ac.cn

Zhou Tong
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
tongzhou@iie.ac.cn

Haoqiang Wang
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
wanghaoqiang@iie.ac.cn

Ze Jin*
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
jinze@iie.ac.cn

Qixu Liu
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University
of Chinese Academy of Sciences
Beijing, China
liuqixu@iie.ac.cn

***Abstract*—Kubernetes has emerged as the de facto standard for container orchestration. However, existing container scheduling strategies prioritize QoS, leading to the co-location of pods with varying permission levels on the same node. This not only introduces risks of privilege escalation but also facilitates the spread of pods with risky permissions across the cluster, exacerbating the potential for attackers to elevate their privileges.**

In this work, our goal is to mitigate permission disparity among pods on each node, thereby reducing the risk of privilege escalation from co-location attack and curbing the spread of high-risk permissions across the cluster. We introduce a novel metric, *Extraneous Risk Privileges (ERP)*, to quantify additional privileges derived from the combination of RBAC permissions and cluster parameters that are utilized by other pods on the node but not by the target pod itself. The *RBAClock* scheduling framework is designed to minimize ERP increase during pod placement, prioritizing the aggregation of pods with similar risk profiles and isolation of those with divergent privileges. Experimental evaluations across 24 CNCF applications demonstrate that, compared to the default scheduler, *RBAClock* alone achieves an average reduction of 41.46% in aggregated privileges in cluster, 64.63% in privilege escalation risk, and 34.59% in high-privilege nodes proportion, with an 8% performance tradeoff. Notably, our investigation uncovered privilege escalation risks in the Kubernetes services of two major cloud providers, Alibaba Cloud and Tencent Cloud, and demonstrated that *RBAClock* can effectively mitigate these threats.

***Index Terms*—Kubernetes, scheduling, RBAC, co-location**

I. Introduction

Kubernetes [1], also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes has been extensively adopted by enterprises and cloud service providers [2], demonstrating significant commercial value and solidifying its position as the de facto standard for cloud container orchestration.

The resounding success of Kubernetes can be partially attributed to its robust ecosystem, encompassing a vast array of tools and third-party applications. These resources enable enterprises and individuals to effectively manage and fully leverage Kubernetes cluster environments for large-scale service deployments, DevOps practices, and other operational

*Corresponding Author

initiatives. As a primary driving force behind cloud-native technology advancement and adoption, the Cloud Native Computing Foundation (CNCF) hosts more than half of its projects as Kubernetes applications [3]. Notable examples of widely adopted tools include Prometheus [4], Istio [5], and KubeVirt [6], which have been extensively implemented by enterprises and cloud service providers. The rapid proliferation of third-party applications prompts us to consider potential error spaces in permission control.

Uncovering security risks in the access control of applications within Kubernetes. To enable applications to operate within cluster environments, developers must grant appropriate authorizations. In Kubernetes, this entails restricting application access to cluster resources through RBAC (Role-Based Access Control) for service accounts. By assigning distinct RBAC permissions to different service accounts and mounting the ServiceAccountToken credentials into containers, Kubernetes grants pods access to cluster resources. This mechanism enables the control plane to validate permissions when a pod presents its ServiceAccountToken during resource access requests at runtime. Notably, RBAC policy configurations yield divergent security implications: Excessively permissive policies may facilitate lateral movement through critical cluster components following container breaches (e.g., deploying attacker-controlled workloads within the cluster), whereas overly restrictive policies may disrupt legitimate application functions. Consequently, strict adherence to the least privilege principle is imperative. However, due to developer oversight or insufficient familiarity with RBAC policies, many applications risk over-privileged configurations. This vulnerability allows attackers who compromise application containers inside pods to exploit service account credentials, potentially causing extensive cluster damage.

More critically, multiple applications co-located on Kubernetes cluster nodes create systemic risks. Even when a compromised application lacks permissions facilitating attacker objectives, insecure container configurations, vulnerabilities in the container runtime, kernel, or Kubernetes itself may enable container escape exploits. Successful escapes compromise the underlying host, allowing adversaries to harvest ServiceAccountTokens mounted by pods on that node. These tokens facilitate privilege escalation ranging from cluster reconnaissance to full cluster takeover [7]. Such co-location attack vectors are non-trivial: prior research [8] has demonstrated that over half of popular cloud platforms can be fully compromised via container escape attacks to steal credentials with high privilege. These risks pose significant threats to multi-workload Kubernetes clusters, particularly in multi-tenant environments where security isolation is paramount.

The gaps in prior works. Regrettably, co-location attacks targeting RBAC permissions have not received adequate research attention. Most Kubernetes-focused studies have concentrated on the feasibility of co-location attacks [9] and flawed security configurations [10]–[13]. Recent works [7], [14] have explored excessive permissions in third-party applications in isolation. For instance, Yang [7] examined attack vectors exploiting excessive permissions in third-party applications within Kubernetes, including co-location attack. In response, Yue et al. [14] developed a static analysis approach to identify actual resource access patterns in applications and compare them against requested RBAC permissions during deployment, thereby identifying excessive permissions.

However, even in the absence of over-privilege, the permissions required by an application may still suffice to enable attackers who have compromised the application to threaten the cluster. Take CVE-2023-26484 [15] as an illustrative example: although subsequent fixes restricted the scope of secrets access, the *virt-operator* service account retains high privileges necessary for functionality but capable of inflicting severe harm on the cluster. Under such conditions, cluster security transitions from excessive permission auditing to safeguarding high-privilege containers, requiring both flaw-resistant application design and strict isolation of co-located containers in orchestrated environments. Malicious or vulnerable neighbor containers can serve as initial access points for attackers, who may then leverage container escape techniques to compromise high-privileged service accounts on the node, thereby threatening the overall cluster security. Existing research lacks the ability to proactively prevent such neighbor relationships that introduce privilege escalation risks.

Our solution. In this work, we focus on exploring the relationship between pod scheduling and RBAC permissions to enhance the security of Kubernetes clusters. Specifically, we propose a greedy RBAC-permission-oriented pod scheduling algorithm and implement a scheduler prototype named *RBAClock*. *RBAClock* decides which node to schedule a pod to by analyzing the risk privileges derived from the RBAC permissions of the pod to be scheduled. This can minimize the exposure to irrelevant privileges for each pod on a node—privileges that are not owned by the pod itself but can be exploited by attackers to expand their gains and perform lateral movement. The key to achieving this is the introduction of a new *Extraneous Risk Privileges (ERP)* metric, which quantifies the external risk privileges that a pod is exposed to. Our approach does not completely eliminate privilege escalation based on co-location attacks, as this is an extremely challenging task. Instead, we use this metric to guide the scheduler to limit and reduce the probability of such risks occurring as much as possible during the scheduling process.

Challenges. During the process of quantifying permission risks, we encountered two significant challenges below (see more detailed discussion of the challenges in III-C).

• ***C1****: Black-box impacts of custom resources.* Numerous third-party applications introduce Custom Resources (CRs), some of which exhibit relevance with the cluster's built-in resources. For security considerations, permissions involving CR must also be incorporated into risk assessments. However, uncovering these associative relationships through cluster runtime proves inherently difficult.

• ***C2****: Difficulty in permission risk quantification.* Due to the dynamic nature of cluster environments, static analysis approaches in prior research [14], [16] struggle to accurately and comprehensively evaluate the potential risks of RBAC permissions across diverse cluster environments.

To address challenges **C1** and **C2**, we developed a two-fold approach. First, we identify custom resources by analyzing structural characteristics in application code and employ taint propagation analysis between identified CRs and Kubernetes built-in resources. Subsequently, deterministic finite automata are utilized to systematically transform non-deterministic permission hazards into quantifiable risk vectors based on cluster parameters.

Effectiveness of resilience against co-location attacks. To evaluate our approach, experiments were conducted across Kubernetes clusters of varying scales, employing multi-dimensional security metrics to assess overall cluster security. Experimental results demonstrate that, compared to the default Kubernetes scheduler, *RBAClock* alone achieves an average reduction of 41.46% in aggregated privileges in cluster, 64.63% in privilege escalation risk, and 34.59% in high-privilege nodes proportion, with a 8% performance tradeoff. Further, the described threat scenarios across two cloud service platforms were identified and confirmed, which were successfully mitigated using *RBAClock*. These findings underscore the effectiveness of our approach in mitigating privilege escalation under co-location attack scenarios.

The following are the contributions of this paper:

1) *New understanding of privilege escalation risk measurement in Kubernetes.* We propose a new metric based on pod's exposure to extraneous privileges arising from RBAC permissions held exclusively by other co-located pods. This metric is designed to measure potential privilege escalation risks in Kubernetes cluster.
2) *New risk quantification techniques.* To address the challenges of quantifying risks from third-party application Custom Resources (CRs) and non-deterministic permissions, we designed a novel subsystem integrating static taint analysis and deterministic finite automata (DFA). This framework quantitatively models the risk exposure of RBAC permissions under specific cluster conditions.
3) *New mitigation techniques.* Based on the metrics proposed in this paper, we developed a security-oriented Kubernetes scheduling system to mitigate privilege escalation risks, and demonstrated its effectiveness through comprehensible experiments, with the source code publicly accessible on our website [17].

II. Background

A. Kubernetes

Kubernetes is an open-source container orchestration platform whose architecture is divided into the **control plane** and **data plane** [18]. The control plane maintains the cluster's desired state, comprising the API Server [19], which exposes the Kubernetes API for communication; etcd [20], a distributed key-value store for cluster data; the controller manager [21], which continuously monitor and align the actual cluster state with the desired state; and the scheduler [22], which assigns workloads to nodes based on resource constraints and policies.

The **data plane**, responsible for executing workloads, operates on worker nodes and consists of several key components. The kubelet [23], an agent on each node, ensures container execution as defined by the control plane. The container runtime (e.g., docker [24], containerd [25]) runs containers, while kube-proxy [26] manages networking, enabling service discovery and routing. These components collectively ensure the cluster's functionality and connectivity.

At the operational level, Kubernetes organizes applications into **pods** [27], the smallest deployable units, each encapsulating containers, storage, and networking specifications. Together, the control plane and data plane enable Kubernetes to deliver a scalable, resilient, and efficient platform for modern containerized applications.

B. Kubernetes Resource

Kubernetes resources represent abstractions that define the state, configuration, and operational aspects of workloads within a Kubernetes cluster [28]. These resources are specified declaratively using YAML or JSON manifests and processed by the Kubernetes control plane to manage the desired state of applications and infrastructure. Resources are grouped into core categories such as workloads, service, and authentication, and are stored as API objects, which can be modified via a RESTful call to the corresponding API.

For third-party applications in Kubernetes, the built-in resources may not be sufficient to capture the application's specific requirements. In such cases, developers can define custom resources [29] to extend the Kubernetes API and model application-specific objects. Custom resources are defined using CustomResourceDefinitions (CRDs) [30], which allow developers to define new resource types and controllers to manage these resources just like built-in ones. By defining custom resources, developers can extend Kubernetes' capabilities to meet the unique requirements of their applications.

C. RBAC in Kubernetes

Role-based access control (RBAC) is a key security mechanism in Kubernetes that restricts access to resources based on user roles and permissions [31]. RBAC is implemented using roles and role bindings, which define the permissions granted to users or service accounts, a specific type of account associated with a pod through a token file mounted into the pod's filesystem. Roles specify the permissions granted to a user or service account, while role bindings associate roles with users or service accounts.

In Kubernetes, permissions are expressed in terms of verbs and resources. Verbs specify the operations that may be performed, while resources define the objects to which those operations apply.

While RBAC offers a robust mechanism for securing Kubernetes clusters, misconfigurations can result in privilege escalation and unauthorized access. For instance, assigning excessive permissions to a service account may enable an attacker to compromise that account and access sensitive resources. To mitigate these risks, it is essential to define roles and role bindings with precision, restricting users and service accounts to the minimal permissions necessary to perform their duties [32].

III. Systematic Security Analysis for Kubernetes Scheduling

A. Security Flaw of Current Kubernetes Scheduling

The default Kubernetes scheduler prioritizes resource utilization and load balancing but disregards the permissions of pods' associated service accounts [33]. This design introduces a security vulnerability: although namespaces provide logical isolation, pods from different namespaces may be co-located on the same physical node due to performance-driven scheduling. If an attacker exploits a container escape or compromises the node, they can perform privilege escalation by enumerating service account tokens mounted in containers on the node, enabling lateral movement across namespaces or cluster-wide attacks, as shown in Fig. 1. Notably, even if no high-privilege tokens are immediately available, the attacker can persist passively until the scheduler places a privileged pod on the node. Consequently, permission-agnostic scheduling inherently risks node-level privilege escalation, either immediately or in the future.

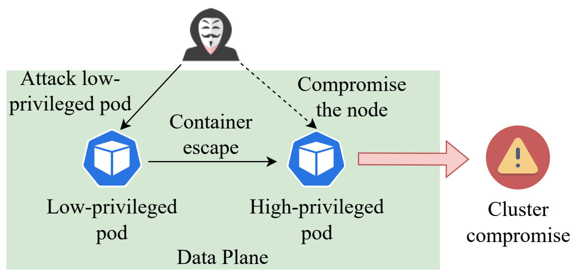

Fig. 1. Compromising cluster by privilege escalation.

B. Threat Model and Goals

Threat Model. We assume a single Kubernetes cluster hosting numerous applications and components. Each application or component may have multiple pods with varying privilege levels, including privileges related to core Kubernetes resources as well as custom resources. Due to the diverse permission requirements across these applications and the uncertainty regarding their impact on various custom resources, there is no universal RBAC profile that can effectively limit resource access without disrupting existing workflows. Consequently, we focus exclusively on pod scheduling managed by the default Kubernetes scheduler and assume no additional RBAC-based access control measures introduced by third-party tools.

Additionally, we account for vulnerabilities and insecure configurations across various host components, including pods, container runtimes, Kubernetes, and the system kernel. Under these circumstances, we assume an attacker could compromise one or more worker nodes through container escapes or other means, thus gaining control of the node and all containers running on it. By doing so, the attacker can exploit all credentials present on the compromised node, including those of any service accounts used by the pods. This access may enable privilege escalation if the compromised credentials grant sufficiently high privileges. Should the attacker find no credentials with adequate permissions to facilitate lateral movement, they may remain on the compromised node until additional pods with the requisite privileges are scheduled. We consider other attack vectors—such as vulnerabilities in containers running on nodes outside the attacker's control, or threats posed by a malicious cloud service provider or system administrator—beyond the scope of this model.

Our goals. Although the control plane contains the most critical privileges for cluster-wide takeover, we assume that the attacker's capabilities are limited to worker node compromise. Given this assumption, our primary objective is to minimize both the number of nodes hosting pods with elevated RBAC privileges and the potential for privilege escalation triggered by co-location attack, thereby mitigating the risks of lateral movement and cluster-wide compromise.

C. Challenges

*1) **C1**: Black-box impacts of custom resources:* A custom resource serves as an endpoint in the Kubernetes API, storing specific types of structured data. They can be parsed by custom controllers, which then perform operations on other types of resources (especially those that come with the default Kubernetes installation, i.e., built-in resources) according to the code logic. This implies that custom resources may provide an avenue for attackers to access internal cluster resources. For instance, in the case of the KubeVirt [6], creating a *VirtualMachineInstance* will lead to the creation of a corresponding pod.

Nevertheless, the logic by which custom controllers parse and operate on other resources is embedded in the application's source code, making the relationships between custom and Kubernetes built-in resources undiscoverable via the cluster runtime. This creates a barrier to evaluating the security implications of custom resources, representing an area of work that has not been systematically explored in existing literature.

*2) **C2**: Difficulty in permission risk quantification:* Kubernetes' RBAC permissions are resource-based; however, resources within clusters are dynamically changing, meaning risk assessment of RBAC permissions must account for all potential resources and their interdependencies. The presence of numerous custom and built-in resources in clusters makes it difficult to quantitatively evaluate RBAC permissions without considering cluster-specific parameters, as these dynamic relationships introduce complexity that static analysis cannot fully capture [14], [16]. For example, when assessing risks associated with *(list/get, secrets)* permissions at namespace level, the evaluation must account for all accessible secrets resources of the service account token type within the scope, as these define the effective permissions available through associated service accounts.

IV. System Design

Under our threat model, we assume that a pod does not incur additional privilege escalation risks when co-located with other pods that utilize identical or functionally similar RBAC permissions affecting the cluster state. Consequently, consolidating pods that require elevated privileges onto fewer nodes reduces the potential attack vectors for cluster compromise. Likewise, grouping pods that do not require extra privileged access mitigates cross-container privilege escalation risks by minimizing privilege disparities among co-located workloads.

To incorporate these considerations into the scheduling process, we introduce a new metric, *ERP*, to quantify the *extraneous risk privileges* associated with each pod, thereby enabling an assessment of the privilege escalation risk within the cluster. Informally, ERP represents the additional privileges that arise from the interaction between cluster parameters and the RBAC permissions utilized by co-located pods on a node. These privileges, which the target pod cannot obtain independently but acquires through its association with other pods on the node, define the potential benefit an attacker might gain by initiating a privilege escalation from the target pod when it is scheduled on that node. Based on this metric, we have designed a security-oriented scheduling algorithm.

Fig. 2 illustrates the overall design of *RBAClock*. We take the source code of the application to be installed as input and extract the data flow relationships between custom resources and Kubernetes built-in resources using *taint analysis*, thereby addressing challenge ***C1*** outlined in Section III-C1. Next, *RBAClock* obtains the RBAC permissions bound to the pod to be scheduled and replaces custom resources with built-in resources. Given the cluster parameters, *RBAClock* establishes a *deterministic finite automata (DFA)* model to derive the risk privilege set from the RBAC permissions, which is then quantified as a vector, thereby addressing challenge ***C2***. During the scheduling phase, *RBAClock* calculates the changes in ERP metrics for each pod when scheduled to different nodes, thereby enabling the selection of a more secure scheduling approach.

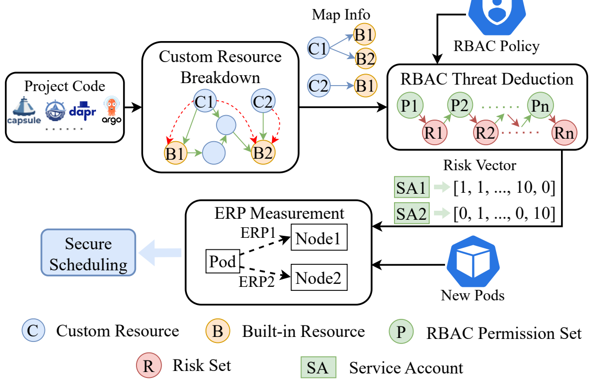

Fig. 2. System overview of *RBAClock*.

A. Custom Resource Breakdown

To investigate the relationship between custom resources and built-in resources, we selected graduated projects from CNCF [3] based on their utilization of custom resources. Our analysis revealed that the source code definitions of custom resources exhibit common structural characteristics. Specifically, for projects written in Go, the struct definitions of CustomResourceDefinitions (CRDs) often include inline fields of specific types or member variables with designated names. Inline fields are employed to store the resource's basic information and metadata, whereas member variables with conventional names delineate the resource's specifications and capture its state. For example, as illustrated in Fig. 3, the *VirtualMachineInstance* struct in KubeVirt contains the field *metav1.TypeMeta*, which specifies the resource's type and API version, and the field *metav1.Object*, which stores metadata such as the resource's name and namespace. Additionally, the conventionally named *Spec* variable refers to a custom struct defining the resource's desired state, while the *Status* variable points to a struct representing the resource's current state. These common patterns facilitate the identification of custom resource struct variables through code analysis.

```
type VirtualMachineInstance struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`
    Spec VirtualMachineInstanceSpec `json:"spec" valid:"required"`
    Status VirtualMachineInstanceStatus `json:"status,omitempty"`
}
```

Fig. 3. Definition of the *VirtualMachineInstance* struct.

```
k8sv1.Pod{
    // .......
    Spec: k8sv1.PodSpec{
        // ......
        NodeSelector:                t.newNodeSelectorRenderer(vmi).Render(),
        Volumes:                     volumeRenderer.Volumes(),
        ImagePullSecrets:            imagePullSecrets,
        DNSConfig:                   vmi.Spec.DNSConfig,
        DNSPolicy:                   vmi.Spec.DNSPolicy,
        ReadinessGates:              readinessGates(),
        EnableServiceLinks:          &enableServiceLinks,
        SchedulerName:               vmi.Spec.SchedulerName,
        Tolerations:                 vmi.Spec.Tolerations,
        TopologySpreadConstraints:   vmi.Spec.TopologySpreadConstraints,
    },
}
```

Fig. 4. Data flow propagation from *VirtualMachineInstance* variable to pod.

By leveraging the structural characteristics of custom resource definitions, *RBAClock* identifies all custom resource variables within the target project's code and designates them as taint sources. Next, *RBAClock* marks the variables corresponding to vulnerable Kubernetes built-in resources as taint sinks, as mentioned in IV-B. Since built-in resources (such as pods and deployments) are also represented as structured variables, data flows from custom resource variables may propagate to members of these built-in resources. For example, in the *renderLaunchManifest* function in KubeVirt, certain aspects of a pod resource are defined by the *VirtualMachineInstance* variable *vmi* during pod creation, as illustrated in Fig. 4. Consequently, *RBAClock* conducts an in-depth search of built-in resources in the code and marks all multi-level members as taint sinks to uncover the data flow propagation between custom and built-in resources.

Following the identification of relationships between custom and built-in resources, *RBAClock* analyzes the RBAC permissions of target service accounts. Kubernetes RBAC permissions are represented as a triple ⟨*apiGroups, resources, verbs, scope*⟩, denoting the allowed operations for resources in specific API groups within the scope. *RBAClock* replace the $apiGroups$ and $resources$ in the triple that pertain to custom resources with corresponding built-in resources, enabling a unified analysis of diverse RBAC permissions in service accounts.

B. RBAC Threat Deduction

As discussed in the challenge ***C2*** of III-C2, certain RBAC permissions pose varying levels of risk under different cluster conditions. To address this issue, *RBAClock* proposes a method based on *deterministic finite automata (DFA)*—a formal model used to represent state transitions—to assess the risks associated with cluster conditions and the RBAC permissions of a target service account. Finally, *RBAClock* quantifies these risks into privilege vectors based on cluster parameters.

First, to summarize the risk categories directly associated with RBAC permissions, we deployed a Kubernetes cluster in a controlled experimental environment and systematically exercised various RBAC permissions by interacting with the Kubernetes API from within application containers. This approach enabled an initial evaluation of the risks corresponding to distinct permissions. The results are summarized in Table I.

RBAClock denotes the RBAC permissions of the target service account as p_1 and retrieves the corresponding risk set r_1 from Table I. Next, when r_1 contains risks that could lead to privilege escalation, *RBAClock* searches the cluster state k for new permissions obtainable via r_1, which are then added to p_1 to form a new set, p_2. This process is represented as $p_1 \rightarrow (r_1) \xrightarrow{k} p_2$, where $p_i \in P$ and $r_i \in R$, corresponding to a state transition in the deterministic finite automaton (DFA).

$$r_i = RiskFromPermissions(p_i, knowledge) \tag{1}$$

$$\begin{aligned} p_{i+1} &= StateTransition(p_i, knowledge, k) \\ &= NewPermissionsFromRisk(r_i, k) \cup p_i \end{aligned} \tag{2}$$

For instance, when a pod possesses permission set p_1 that enables viewing all secrets within its namespace (r_1), the DFA generates expanded set p_2, which combines p_1 with all permissions represented by secrets in that namespace under current cluster state k.

Our assessment proceeds iteratively by querying for risks and evaluating the cluster state until both the permission set and the risk set reach a fixed point. This approach enables a more accurate evaluation of the risks associated with RBAC permissions, thereby guiding the subsequent quantification and mitigation of overall risks to both nodes and the cluster.

Lastly, to quantify the impact of the final risks output from the DFA across diverse parameter clusters, we identified three generic atomic privileges targeting computing resources—*leakage*, *tampering*, and *execution*—through scenario analysis. Risk categories manifest as specific combinations of these atomic privileges within defined scopes, as shown in Table II.

- **Leakage Privilege**. This refers to the ability to read metadata and state information of a target resource, as well as to exfiltrate its content (e.g., source code and credentials stored within a container).

TABLE I
POTENTIAL IMPACT OF DIFFERENT RBAC PERMISSIONS

Impact[a]	Permissions	Scope
1	(list/get, secrets)	= cluster
2	(create/update/patch, workloads[b])	≥ resource-specific
1	(create/update/patch, clusterrolebindings) (bind, clusterroles)	≥ namespace
1	(patch, clusterroles) (escalate, clusterroles)	≥ namespace
4	(update/patch/delete, nodes)	≥ namespace
4	(delete, workloads[b])	≥ resource-specific
5, 6	(list/get, secrets)	= namespace
4, 5	(update/patch, services)	≥ namespace
5	(create, services)	≥ namespace
4	(delete, services)	≥ resource-specific
4, 5	(create/update/patch/delete, networkpolicies)	≥ resource-specific
4	(delete, ingresses)	≥ resource-specific
6	(create, certificatesigningrequests) (update/patch, crt/approval)	= cluster
4	(create/update/patch/delete, validatingwebhookconfigurations)	= cluster
1	(create/update/patch, mutatingwebhookconfigurations)	= cluster
6	(impersonate, serviceaccounts)	= namespace
1	(impersonate, serviceaccounts)	= cluster
1	(impersonate, users) (impersonate, groups)	= cluster
6	(create, serviceaccounts/token)	= namespace
1	(create, serviceaccounts/token)	= cluster
1	(patch, roles) (escalate, roles)	≥ namespace
6	(create/patch/update, rolebindings) (bind, roles)	= namespace
1	(create/patch/update, rolebindings) (bind, roles)	= cluster
1	(create/patch/update, rolebindings) (bind, clusterroles)	≥ namespace
5, 6	(create, secrets) (list/get, secrets)	= namespace
1	(create, secrets) (list/get, secrets)	= cluster
3	(create, pods/exec)	= namespace
1	(create, pods/exec)	= cluster
4	(create, pods/eviction)	≥ namespace

[a] 1-Take over the whole cluster, 2-Take over worker nodes, 3-Take over containers, 4-Compromise cluster availability, 5-Leak sensitive information, 6-Perform privilege escalation.
[b] *workload* is an alias of resources: *pods, deployments, statefulsets, daemonsets, jobs, cronjobs*, and *replicasets*.

- **Tampering Privilege**. This encompasses the capability to alter the metadata and status information of a target resource, potentially leading to service disruption and compromising its content.
- **Execution Privilege**. This primarily involves the ability to execute malicious workflows or payloads, thereby enabling attackers to gain significant control over the target resource.

Therefore, the impact of risk set r_i on the cluster resources can be defined as the vector $priv_i$, as shown in equation 3, where $\{o_1, o_2, \ldots, o_v\}$ represents the set of all computing resources (containers and nodes) in the cluster, and o_j^l, o_j^t, o_j^e denote whether r_i have *leakage, tampering* or *execution* priv-

TABLE II
PRIVILEGES FOR DIFFERENT IMPACT

Impact	Privilege			Scope
	Leakage	Tampering	Execution	
Take over cluster	✓	✓	✓	All resources in cluster
Take over worker nodes	✓	✓	✓	Nodes and containers in data plane
Take over containers	✓	✓	✓	Specific containers
Compromise availability	×	✓	×	Specific Resources
Leak information	✓	×	×	Specific Resources

ileges on resource o_j, taking values 1 or 0.

$$priv_i = (o_1^l, o_1^t, o_1^e, \ldots, o_v^l, o_v^t, o_v^e) \tag{3}$$

For example, if r_i has *leakage* privilege on resource o_1, *tampering* privilege on resource o_2, and *execution* privilege on resource o_3, the corresponding $priv_i$ would be $(1, 0, 0, 0, 1, 0, 0, 0, 1)$.

The vector representations of risks arising from the combination of RBAC permissions and cluster conditions is crucial for evaluating the potential threats posed by these policies. This assessment serves as a foundation for the subsequent analysis of the overall security risks within the Kubernetes cluster.

C. ERP Measurement

After determining potential threats based on the service account of each pod, *RBAClock* will broaden the scope of the analysis to assess the risks stemming from permission propagation across the cluster nodes and the entire cluster.

Consider a node in the cluster. Each pod on the node is associated with its own service account credentials and corresponding permission set. When an attacker infiltrates the node through techniques such as container escape, they may exploit the service account credentials of all pods on that node to maximize gains. **In this scenario, we reasonably assume that the damage inflicted by an attacker depends on the aggregate risk privileges held by all service accounts on the node. These privileges include those for *leakage, tampering*, and *execution* across various cluster resources.** For example, if a compromised node hosts two pods with service accounts, one with the privilege to read cluster secrets and the other with the privilege to write secrets, an attacker could acquire both risk privileges and escalate the overall damage.

More crucially, under this scenario, the attacker's initial intrusion payoff depends on the node hosting the compromised pod. Specifically, when the privileges hold by the attacker's pod significantly diverge from the collective privilege set posed by other co-located pods, container escape attacks yield amplified privilege escalation opportunities, thereby increasing potential cluster damage. **This implies that the expected payoff of privilege escalation is proportional to the divergence between a pod's privileges and the union of privileges held by all co-located pods**.

To quantify risks posed by intra-node privilege discrepancies, *RBAClock* formulates a mathematical model for quantifying privilege differences between co-located pods. Assuming all worker nodes operate independent instances of operating systems, which are numbered $1, ..., N$. Let $priv_i^n$ represent the risk privileges associated with the service account of pod_i on $node_n$, as detailed in IV-B. The potential risk privileges of $node_n$ can be represented as the union of the privileges of the I pods residing on that node, as shown in equation 4.

$$PRIV^n = \bigcup_{i=1}^{I} priv_i^n \tag{4}$$

By applying the *XOR* operation between $priv_i^n$ and $PRIV^n$, *RBAClock* remove the privileges of pod_i^n to obtain the privilege set of other pods on $node_n$, formalized in equation 5. E_i^n signifies the extra privileges a potential attacker of pod_i^n can gain through privilege escalation.

$$E_i^n = PRIV^n \oplus priv_i^n \tag{5}$$

Given that different resources have varying levels of importance, we introduce a weight vector to enhance the precision of threat assessment. Let $W = [w_1, w_2, ..., w_v]$ denote the importance weights associated with resources in the cluster, where $w_i \geq 0$ is an integer:

- $w_i = 0$: Defender does not consider the security of o_i;
- $w_i = 1$: Defender treats o_i's security as equally important as others;
- $w_i > 1$: Higher values indicate greater security priority for o_i.

The weight vector, which administrators freely adjust according to security objectives, is multiplied by E_i^n to quantify the potential gains of acquiring *extraneous risk privileges (ERP)* on the $node_n$ through privilege escalation, as formalized in equation 6.

$$ERP_i^n = W \cdot E_i^n \tag{6}$$

The ERP metric quantifies the privilege homogeneity between a pod and its co-located peers. For scheduling process in Kubernetes, the metric addresses two core questions from dual perspectives:

1) For newly scheduled pods: What potential gains could adversaries obtain from privilege escalation attacks if the pod is scheduled to different nodes?
2) For pre-deployed pods: How does the scheduling decision affect the privilege escalation gains of existing pods' potential attackers?

The ERP_i^n parameter effectively answers the first question, whereas quantifying the second question necessitates node-level ERP—the aggregated value of ERP_i^n across all pods on $node_n$:

$$ERP^n = \sum_{i=1}^{I} ERP_i^n \tag{7}$$

Simultaneously, to characterize the aggregated gains of privilege escalation across the entire cluster, *RBAClock* compute the sum of every ERP^n as a holistic security metric for the cluster:

$$ERP = \sum_{n=1}^{N} ERP^n \tag{8}$$

This completes our quantification of privilege escalation risks at node and cluster levels using ERP, enabling the development of secure scheduling policies enhancing cluster resilience against unauthorized escalation and privilege spread.

D. Secure Scheduling

Leveraging the proposed quantitative metrics and security limitations detailed in III, we designed a secure scheduling algorithm integrated into the Kubernetes scheduler workflow. As pod density increases within the cluster, privilege escalation risks and ERP inevitably escalate. Our secure scheduling algorithm, however, effectively curbs the growth rate of ERP to minimally achievable levels, thereby constraining cluster-wide privilege escalation risks. Accordingly, we propose a simple yet effective methodology: When scheduling new pods, the algorithm selects the node that minimizes the increase of the cluster-wide ERP metric as the scheduling destination.

For each new pod, the Kubernetes scheduling logic operates through two sequential phases:

1) **Filtering:** Kubernetes filters out nodes failing to satisfy the pod's fundamental requirements (e.g., label selectors or resource constraints).
2) **Scoring:** For remaining candidate nodes, the scheduler invokes all enabled scoring plugins. Each plugin computes node-specific scores based on its internal logic. Individual plugin scores are multiplied by their configured weights and aggregated into a final composite score. The pod is then scheduled onto the node achieving the highest score.

To ensure interoperability with existing scheduling mechanisms, we implement our algorithm as a plugin integrated into the scoring phase. The plugin quantifies the incremental increase in the cluster-wide ERP metric when scheduling the pod onto each candidate node. Lower ERP increases correspond to higher scores assigned by our plugin.

Simultaneously, administrators configure tunable weighting parameters for the plugin's output. Higher weights amplify the plugin's influence on scheduling decisions, enabling explicit trade-offs between security and performance objectives. For instance, assigning equal weights to the *NodeResourcesFit* plugin and our scoring plugin concurrently optimizes both node resource utilization and privilege escalation risk mitigation during scheduling decisions.

Fig. 5 illustrates our scheduling strategy utilizing ERP. This simplified example demonstrates the scheduling of three pods with varying privilege profiles (the profile for Pod-1 is 1,2, Pod-2 is 5,7, and Pod-3 is 2,6) in a dual-node cluster with only our scoring plugin enabled. During Pod-1 scheduling, calculations of ERP^n variation revealed zero increase for both Node-1 and Node-2 (Fig. 5(a)), resulting in random assignment to Node-1. Next, when scheduling Pod-2, assigning it to Node-1 results in an increase of 4 in the corresponding ERP^1, whereas assigning it to Node-2 keeps ERP^2 unchanged (Fig. 5(b)). To minimize the increase of ERP, Pod-2 is therefore allocated to Node-2. Finally, for Pod-3, the changes in ERP^1 and ERP^2 are 2 and 4 (Fig. 5(c)), respectively. Accordingly, Pod-3 is assigned to Node-1 (Fig. 5(d)). As a result, the spread of privilege 2 was restricted, while privileges 5 and 7 were effectively isolated from privileges 1, 2, and 6.

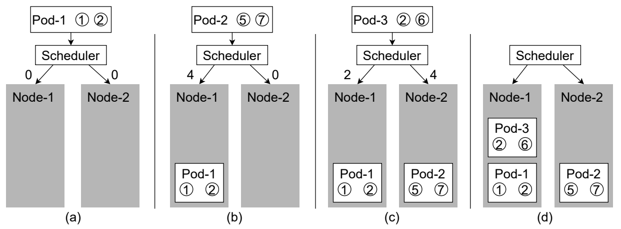

Fig. 5. A scheduling example for new pods based on ERP metric. The number in the circle represent the privileges of the pod, while the number on each arrow represent the ERP^n growth after scheduling.

The scheduling scheme we propose not only mitigates the risk of privilege escalation during scheduling but also limits the scope of diffusion for each risky privilege across the cluster. These benefits stem from the core design principle of the scheduling algorithm: minimizing privilege disparities among pods on each node to the greatest extent possible.

V. Evaluation

In this section, we present a multifaceted evaluation of *RBAClock*. We first assess its effectiveness in reducing ERP and then analyze the security benefits of ERP reduction from multiple perspectives. Finally, we conduct a preliminary evaluation of the performance impact of *RBAClock* on cluster applications. Our results aim to address the following research questions:

- **RQ1:** How much can *RBAClock* reduce ERP?
- **RQ2:** How well can *RBAClock* archieve our security goals?
- **RQ3:** How does *RBAClock* affect the performance of the applications?

A. Experimental Setup

1) Prototype Implementation: We have developed and open-sourced the *RBAClock* prototype on our website [17]. The architecture of implementation is illustrated in Fig. 6, comprising two components: one for threat quantification of service account permissions and the other for designing a scheduling plugin based on our algorithm and integrating it into Kubernetes' online scheduling workflow.

During the threat quantification phase, *RBAClock* uses source code as input and perform taint analysis with CodeQL [34], establishing mappings between custom resources and built-in resources. This is followed by threat reasoning on cluster service accounts, ultimately generating a risk vector for each account. To enhance scalability, we persist all risk vectors in a PostgreSQL database featuring vector storage and query capabilities [35]. This database serves as a repository accessible to the scheduling plugin and potential future components for querying.

In the online scheduling phase, the scheduler plugin incorporating our algorithm is utilized. When a new pod requires scheduling, the MutatingWebhook [36] assigns *RBAClock* as its scheduler, which first filters nodes that do not satisfy the pod's requirements to ensure that the pod can run normally after scheduling. During the subsequent node prioritization stage, the scheduler retrieves the risk vector of the corresponding service account from the database and calculates the ERP changes associated with scheduling the pod to different nodes, ultimately determining the optimal node for placement.

All experiments are performed on a virtual machine cluster consisting of 28 nodes, provisioned by minikube. Each node is evenly allocated 56GB of memory and an Intel i7-12700 processor. The cluster operates on Kubernetes 1.28 with the Debian 12 operating system.

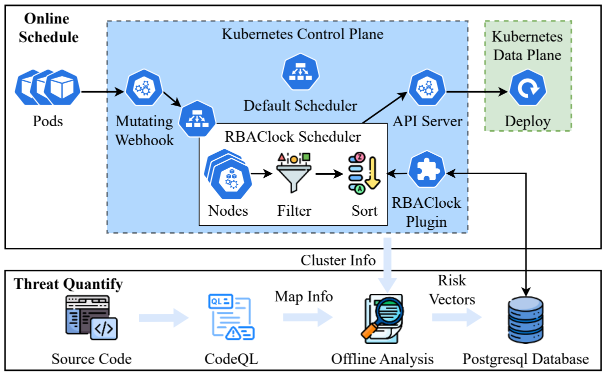

Fig. 6. Implementation of *RBAClock*.

2) Evaluation Dataset: To better simulate production environments, we constructed the evaluation dataset using third-party Kubernetes applications from Cloud Native Computing Foundation (CNCF) projects—the most prominent open-source ecosystem in cloud computing. Because our prototype system implements built-in resource detection only for the Go programming language, 44 projects were selected based on their programming language. Furthermore, after deployment, many applications either conflicted with one another or failed to deploy due to strict environmental requirements. These applications were subsequently removed, resulting in a final selection of 24 applications with 94 pods deployed in the cluster for evaluation, as detailed in Table III.

Prior to conducting experiments, we mined the mapping relationships between custom resources in these third-party applications and Kubernetes built-in resources using the method described in IV-A, caching these mappings into a PostgreSQL database to facilitate rapid resource resolution during threat assessment. The results of mapping relationship extraction for custom resources across third-party projects are shown in Table V (in the Appendix), demonstrating extensive connections between custom and Kubernetes-native resources.

TABLE III
APPLICATIONS USED IN THE EVALUATION

Type	Applications
App Definition and Development	dapr, KubeVela, KubeVirt, Argo, Brigade, Flux, Keptn, OpenKruise
Observability and Analysis	Litmus, Kuberhealthy
Orchestration and Management	Koordinator, Kuadrant, Istio, Crossplane, KEDA, Kured
Provisioning	zot, Bank-Vaults, cert-manager, external-secrets, KubeArmor
Runtime	Antrea, k8up, Kanister

Finally, for parametric initialization consistency, all containers and nodes in the experimental environment are treated as equally important within their groups. In weight vector W, container weights are set to 1 and node weights to 10, since node resources are generally more critical than container ones.

B. Estimation of ERP Reduction (*RQ1*)

To assess how effectively *RBAClock* reduces cluster ERP metrics, we performed multiple rounds of testing on clusters with 2 to 28 nodes. In each test round, we randomized the deployment order of experimental applications and deployed them under both the Kubernetes default scheduler and *RBAClock* respectively, calculating the ERP metric for each deployment. After each experiment, the test cluster was restored to its initial state using a snapshot mechanism.

The experimental results are depicted in Fig. 7 and presented in Table VI (in the Appendix). In contrast to the upward trend observed with the default Kubernetes scheduler, *RBAClock* achieves up to a 100% reduction and an average reduction of 84% in the cluster's ERP metric across varying node counts. Notably, when the number of nodes exceeds six ($Node_{num} > 6$), the magnitude of ERP reduction is more pronounced, as the scheduler benefits from a greater number of scheduling options for optimization, thereby yielding superior outcomes. In experiments with 26 and 27 nodes, the ERP metric of the scheduling results produced by the *RBAClock* dropped to 0, indicating that all working nodes hosted pods with identical risk privileges, effectively minimizing privilege imbalances across the cluster. Conversely, experiments with fewer nodes provided the scheduler with more limited scheduling options, resulting in a smaller magnitude of ERP reduction.

Additionally, both the default scheduler and *RBAClock* exhibited significant fluctuations in ERP values as the number of nodes scales. Similar volatility was observed in the evaluation results of the metrics presented in V-C. Our analysis revealed that the sequence of the experimental application deployments impacts the scheduling outcomes. Specifically, the scheduler made greedy decisions based solely on the specification of the new pod and the existing cluster state, without foreknowledge of subsequent deployments. Consequently, schedulers may converge to local optima. We have detailed the mitigation measures for this issue in VI-B. Nonetheless, our scheduler consistently reduced the cluster's ERP metric across all experiments, while simultaneously demonstrating significantly better security than the default scheduler across all evaluation dimensions, as detailed in V-C.

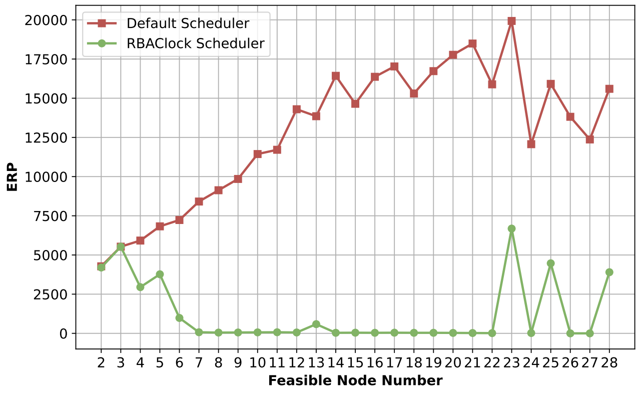

Fig. 7. ERP score across deployments using default scheduler and *RBAClock* respectively with different node counts.

C. Security Benefits (***RQ2***)

RBAClock effectively reduces the ERP metric by striving to schedule pods with similar risk privileges onto a minimal set of nodes while isolating pods with substantial differences in privileges from one another. As outlined in III-B, our objectives are to mitigate the risk of privilege elevation stemming from co-location attacks throughout the cluster (***G1***) and to minimize the number of nodes housing high-privilege pods (***G2***). In order to achieve these security goals, we assess the security of the cluster from three distinct dimensions.

1) Average aggregated risk: Under the threat model, we define the union of all risk privilege vectors of pods (as described in IV-B) on a node as its aggregated risk and compute the ℓ_1 norm of this union vector to quantify the maximum potential gain for an attacker compromising the node. When attackers lack prior knowledge of node configurations, the average ℓ_1 norm of the aggregated risk privilege vectors across all nodes represents the expected potential gain from compromising an arbitrary node in the data plane. This metric allows us to perform a comprehensive assessment of the security of cluster nodes.

Fig. 8 illustrates the average aggregated risks of the cluster scheduled by the default scheduler and *RBAClock* across all experimental rounds in V-B. When the node count exceeds three, *RBAClock* reduces average aggregated risky privileges by up to 66.09% and an average of 41.64% compared to the default scheduler, demonstrating that *RBAClock* reduces the diffusion of risk privilege sets across the cluster by co-locating pods with similar privileges, thereby effectively minimizing the expected gain for attackers compromising nodes. We further observe that as the number of nodes increases, average aggregated risk tends to rise, as the growing cluster size allows pods with risk privileges to exert a more far-reaching impact.

2) Possible privilege escalation: To validate *RBAClock*'s effectiveness in mitigating privilege elevation (as outlined in our security goal ***G1***), we evaluate potential privilege elevation

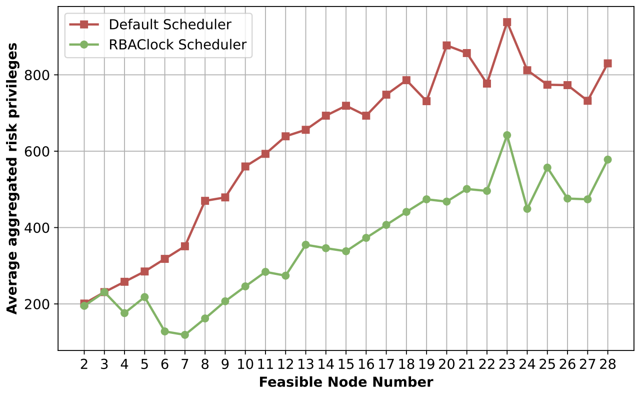

Fig. 8. ℓ_1 norm of average aggregated risk vectors across deployments using default scheduler and *RBAClock* respectively.

opportunities for attackers within the cluster. To ensure generality, we consider all pods as potential origins of privilege escalation and define any pod with privileges not held by the origin pod as a potential target. Specifically, if pod_b's risk privileges are a subset of that of pod_a, no elevation from pod_a to pod_b will occur. We computed the number of escalation paths on each node, which quantifies the extent to which pods within the cluster are exposed to privilege escalation threats.

Fig. 9 illustrates the number of privilege escalation paths of the cluster scheduled by the default scheduler and *RBAClock* across all experimental rounds in V-B. We observe that *RBAClock* effectively reduces privilege escalation opportunities in the cluster by up to 100% and an average of 64.63% across all experiments, primarily by co-locating pods with similar privileges and isolating pods with substantial differences across distinct nodes. As the number of nodes increased, with the total pod count held constant, both schedulers exhibited a general decline in the number of elevation paths. This trend arises because more nodes result in fewer pods per node on average, shrinking the combinatorial space available for privilege escalation attacks. It is noteworthy that with the ERP reaching zero, container escape opportunities are eliminated for the scenarios with 26 and 27 nodes. However, differences in the scheduling order may cause pods with smaller privilege differences to first occupy all idle nodes. This can degrade the protection when subsequently scheduling pods with larger privilege differences, resulting in an increased number of privilege escalations at 28-node scale (as discussed in V-B). Despite this, *RBAClock* still demonstrates superior security compared to the default scheduling scheme.

3) Proportion of cluster-compromising node: To assess *RBAClock*'s efficacy in minimizing nodes hosting pods with elevated RBAC privileges (as outlined as ***G2***), based on our analysis in Table I, we focus on post-scheduling pods with privileges enabling cluster compromise. These highly privileged pods allow attackers to take over the cluster and exert maximal threat potential on the cluster. We count such pods and compute the proportion of worker nodes hosting at least one of them, which directly quantifies the likelihood of an attacker without prior knowledge compromising the entire

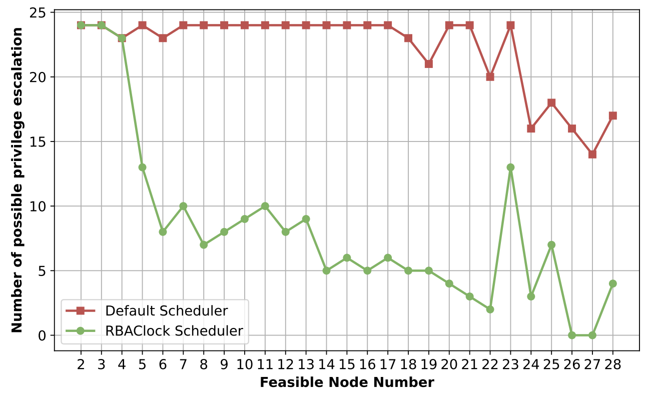

Fig. 9. Number of possible privilege escalation path across deployments using default scheduler and *RBAClock* respectively.

cluster by hijacking a single worker node [8].

Fig. 10 illustrates the proportion of nodes hosting pods with cluster takeover privileges in the cluster scheduled by the default scheduler and *RBAClock* across all experimental rounds in V-B. *RBAClock* reduces the proportion of nodes hosting pods with risky privileges by up to 56.67% and an average of 34.59%, attributed to its strategy of aggregating these pods onto a minimal set of nodes. As the number of nodes increases, both schedulers exhibit a general downward trend in the proportion of nodes with high-privilege pods. Notably, the default scheduler's pod distribution led to nearly every node hosting high-privilege pods until the node count reached 24, after which a downward trend emerged. In contrast, *RBAClock* consistently minimizes this proportion, with increasing node numbers providing greater opportunities for optimization.

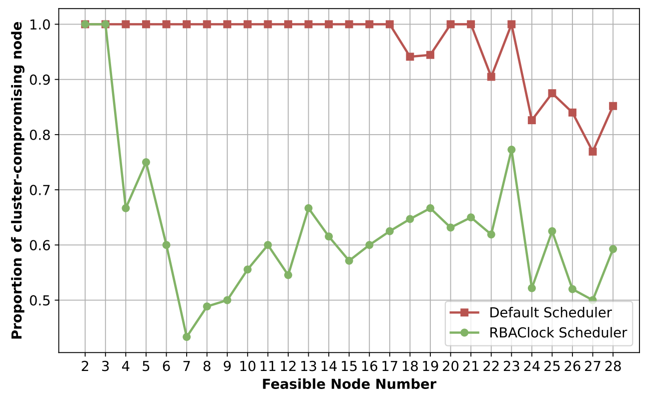

Fig. 10. Proportion of cluster-compromising node across deployments using default scheduler and *RBAClock* respectively.

D. Performance Impact (*RQ3*)

Generally, it is not recommended to use *RBAClock* alone for cluster scheduling management. Instead, it should be combined with the existing performance-oriented scheduling algorithm to balance post-scheduling security and performance. Nevertheless, it is still worthwhile to assess the performance impact of *RBAClock* on deployed cluster applications. However, due to the functions and positioning of the experimental subjects, most of them are not performance-sensitive. For instance, KubeVirt is used to manage virtual machines on Kubernetes. For other applications, such as Istio, their performance cannot be measured without specific upper-level microservices. As a result of these factors, it is infeasible to perform a unified and reasonable assessment of the experimental subjects using the traditional stress-testing scheme.

TABLE IV
PERFORMANCE OF APPLICATIONS WITH DIFFERENT PROPORTIONS OF HIGH-PRIVILEGED POD (HP) AFTER *RBAClock* SCHEDULING

Application	Baseline	20% HP	40% HP	60% HP	80% HP
Nginx	47409.09	45937.93	42419.61	42501.41	46922.99
Apache	37116.08	36114.21	34083.11	35483.66	35912.21
Mongodb	22805.75	21601.75	20884.75	20792.02	21933.02
Mysql	35944.24	33812.05	32724.74	32903.11	33991.05

To resolve such issues, a compromised performance measurement scheme was proposed. Four performance-sensitive applications (Nginx, Apache, MongoDB, and MySQL) were selected as experimental subjects. The deployed pod count for each subject was increased to 100. By assigning different RBAC permissions to each pod replica of the same application, the practical use cases of *RBAClock* can be simulated. For simplicity, each pod replica is assigned one of two permission levels: permissions enabling cluster takeover privilege and ordinary, low-level permissions. By adjusting the proportion of high-privileged pod (HP) in the replica count, the impact of *RBAClock* on pod distribution in practical use cases can be simulated, thereby evaluating its performance degradation.

For each application, 20%, 40%, 60%, and 80% of the pods were configured as high-privilege in each experimental round, followed by stress testing after scheduling. Table IV presents the evaluation results, where "baseline" denotes the performance of applications scheduled by the default scheduler. As the ratio of high-privilege to low-privilege pods approached parity, *RBAClock*'s scheduling decisions increasingly diverged from those of the default scheduler, leading to measurable performance tradeoffs. The maximum impact was approximately 10% for Nginx and Apache, and 8% for MySQL and MongoDB.

E. Case Study

*1) **Alibaba Cloud**:* Alibaba Cloud Container Service for Kubernetes (ACK) [37] exhibits a critical security weakness: insufficient isolation between control plane and data plane workloads. Some system pods with highly privileges can be co-located with user workloads on the same node. This architectural flaw enables privilege escalation attacks, which allows attackers to gain control over all worker nodes by creating malicious containers that run on every node. To mitigate the risk, low-privilege pods are systematically scheduled away from other pods by *RBAClock*, minimizing co-location and thereby reducing the risk of privilege escalation through node-level exposure. We have responsibly disclosed this finding to Alibaba Security Response Center [38] and obtained their confirmation.

*2) **Tencent Cloud:*** In Tencent Kubernetes Engine (TKE) [39], there are pods co-located with other workflows on several nodes across the data plane and responsible for networking. These pods hold dangerous privileges, including the ability to impersonate any user and group. While they do not expose any attack surface to external networks, attackers can still leverage container escape from other workloads on the same node to gain control of them. This poses a significant risk, as an attacker compromising these pods could perform privilege escalation, enabling them to impersonate users in *system:masters* group, which holds full administrative privileges in the cluster, thus taking over the whole cluster. In contrast, *RBAClock* effectively mitigates this risk by ensuring that these pods are not co-located with other low-privileged workloads, thereby preventing potential privilege escalation through node-level exposure. We have responsibly disclosed this finding to Tencent Security Response Center [40] and obtained their confirmation.

VI. Discussion

A. Limitation

1) Deamonsets: In Kubernetes, the daemonset resource ensures that all (or a subset of) nodes run a copy of a pod [41], thereby providing essential infrastructure services such as storage, networking, and monitoring. Scheduling additional replicas that violate the daemonset specification is not permitted; consequently, *RBAClock* excludes these replicas from ERP calculation and scheduling. Our research indicates that these replicas generally do not possess high-risk privileges capable of compromising the entire cluster. However, if misconfigured by developers, pods within daemonsets with excessive privileges can facilitate an attacker's takeover of the cluster. To address this risk, *RBAClock* incorporates an alerting mechanism that detects and flags risky privileges in all pods, thereby enabling defenders to promptly remediate overly privileged daemonset configurations upon discovery.

2) Relation between CR and Builtin: In our analysis of custom resources, we employed taint analysis to identify the initial mappings between custom and Kubernetes built-in resources within RBAC configurations. However, the data flow dynamics between these resource types may extend beyond simple mappings. Custom resources may require intricate code-based validation that constrains the attack vectors exploiting such resources, which introduces minor false positives in *RBAClock*. Importantly, any false positives generated by this process primarily affect performance rather than the security of scheduling decisions. To achieve a more precise risk assessment, we plan to further investigate the latent attack surface introduced by custom resources in future research.

3) Data Sensitivity in Scheduling: RBAClock currently leverages pod RBAC permissions as a scheduling criterion. However, production containers may contain sensitive intra-container data (e.g., API keys or personal information), making them attractive attack targets regardless of their privilege level. Although it is often assumed that high-privilege containers are more likely to host sensitive data, privilege level and data sensitivity are related but not equivalent; containers may require high privileges for operational reasons without necessarily storing sensitive data. At present, neither *RBAClock* nor default Kubernetes schedulers account for intra-container data sensitivity in placement decisions. Addressing this limitation by integrating data sensitivity evaluation into the scheduling framework is an important direction for future work, which we plan to pursue.

B. Volatility of Scheduling Outcomes

As discussed in our evaluation section, the scheduler's greedy decisions are sensitive to application deployment order, potentially trapping scheduling outcomes in local optima and increasing result uncertainty. While *RBAClock* consistently reduces the cluster's ERP metric and outperforms the default scheduler across all security benchmarks, its security metrics exhibit notable variability in certain experimental scenarios. To address this, we propose a clustering-based strategy for pod redistribution: leveraging risk privilege vectors of existing pods as input, we apply algorithms like *K-means* for classification and validate results using *silhouette scores* [42]. Optimal clustering enables the allocation of homogeneous pod groups to pre-allocated node clusters, enhancing security isolation. We have integrated this functionality into *RBAClock*; however, such a strategy may disrupt established workflows, thereby necessitating cluster administrators to define the scope of scheduling targets. We plan to explore more refined security isolation frameworks in future research.

C. Balance between Performance and Security

While performance remains a critical scheduling criterion for cloud providers, our approach does not seek to replace existing scheduling algorithms. Rather, we augment the scheduling framework with a security dimension that explicitly incorporates pod RBAC permissions into scheduling decisions, effectively mitigating privilege escalation risks. Our solution offers deployment flexibility: either as a standalone scheduler or integrated with existing schemes, enabling providers to dynamically optimize the security-performance tradeoff during scheduling while maintaining cluster security guarantees.

D. Container Escape Prevention

Containers are widely adopted in cloud environments due to their lightweight resource usage, rapid deployment, and ease of scaling compared to traditional virtual machines. This efficiency and flexibility make containers attractive for large-scale, dynamic workloads in Kubernetes clusters. However, containers inherently share the host operating system kernel, which means their isolation is weaker than that provided by virtual machines. Virtual machines employ hardware-level isolation and maintain separate kernels, offering significantly stronger security boundaries under our threat model. Thus, administrators can opt for virtual machines (e.g., Kata Containers [43]) over containers to enhance security. Additionally, the risk of container escape can be reduced by eliminating insecure configurations, applying timely updates to vulnerable components, and adopting hardened container runtimes (e.g., gVisor [44]).

VII. Related Work

A. Kubernetes Security

The extensive use of Kubernetes has prompted growing security concerns, with prior research identifying vulnerabilities in Kubernetes across diverse attack vectors. Zeng et al. [45] conducted a comprehensive vulnerability analysis of key cloud native components (e.g., Docker, Kubernetes, CNI, and Istio), categorizing risks, potential attack paths, and mitigation strategies—though their work did not address security challenges arising from RBAC permissions. Spahn et al. [46] deployed a high-interaction Kubernetes honeypot, uncovering substantial real-world attacks targeting internet-exposed container orchestration systems. Shamim et al. [12], [47] systematically explored attacks on manifests violating security best practices. Yang et al. [7] demonstrated that under specific conditions, attackers can exploit excessive permissions in third-party applications to compromise clusters, yet failed to propose effective mitigation strategies. Building on this work, Gu et al. [14] developed an automated method to detect excessive RBAC permissions in third-party applications, revealing widespread risks across numerous tools.

To tackle the specific risks in Kubernetes, both academia and industry have developed diverse protection approaches. Rahman et al. [48] empirically identified 11 categories of insecure configurations in open-source Kubernetes manifests and created SLI-KUBE, a static analysis tool enabling practitioners to detect and remediate these issues. Karn et al. [49] introduced a framework leveraging system calls and interpretable machine learning to detect cryptomining in containerized environments. Lin et al. [50] proposed CDL, a distributed classification-learning framework that identifies malicious activities in Kubernetes clusters through anomaly detection using system call feature vectors. Tien et al. [51] developed a neural network-based anomaly detection system tailored for Kubernetes. Haque et al. [52] presented a knowledge-graph-based method to automate the detection and resolution of security configuration errors. In industry, static analysis tools such as Kubescape and Checkov [53]–[62] generate alerts using cluster state and threat models, while runtime protection tools like Falco and KubeArmor [63]–[66] monitor and block anomalies in real time via system call analysis. Despite these advancements, the critical challenge of RBAC-driven privilege escalation remains unaddressed by existing solutions.

Our work extends the findings of Yang et al. [7] and Gu et al. [14], positing that even when adhering to least-privilege principles, application permissions can still pose cluster threats. We introduce a framework that quantifies permission impacts and leverages scheduling-based optimization to mitigate the risk of RBAC privilege escalation, addressing a critical gap in existing defenses.

B. Co-location Threat in Cloud

Research on co-location attacks in cloud computing environments broadly categorizes into two domains: attack methodologies and defensive strategies. Attack-focused studies employ diverse techniques to achieve co-location of malicious VMs or containers, enabling side-channel exploitation [67]–[70]. For instance, Shringarputale et al. [69] leveraged hardware side channels in cloud containers to substantially boost attack success rates; Fang et al. [70] optimized attack costs via scheduler affinity configurations, while Zhang et al. first validated VM co-residence through side-channel analysis [71] and later executed cross-tenant fine-grained attacks in PaaS environments [68], these works demonstrate that attackers exploit both hardware flaws (e.g., LLC cache, memory buses) and scheduling policy weaknesses, often augmenting their approaches with machine learning to enhance attack obfuscation.

Defensive approaches in this domain center on detection, migration, and resource allocation optimization [72]–[80]. Zhang et al. [75] devised VM migration strategies using the Vickrey-Clarke-Groves (VCG) mechanism to mitigate attacks. At the resource scheduling layer, Azar et al. [76] engineered anti-co-location VM placement algorithms. Additionally, Le et al. [74] addressed container-based co-location threats with a system scheduling-aware scheduler, minimizing excessive system call exposure through optimized container placement. Our research draws significant inspiration from these works, adapting their methodologies to secure scheduling optimization for RBAC permissions in Kubernetes scenario.

VIII. Conlusion

In this paper, we propose *RBAClock*, a pod scheduling framework based on RBAC permissions. By exploring the relationships between custom resources of third-party applications and Kubernetes built-in resources, and by inferring the ultimate impact of permissions on the cluster based on its parameters, *RBAClock* quantifies the risks associated with RBAC permissions. Subsequently, *RBAClock* aggregates pods with similar risk profiles and isolates permissions with significant risk discrepancies, thereby reducing both the potential for privilege escalation within the cluster and the number of high-privilege nodes. The results of experiments verify the effectiveness of *RBAClock* in mitigating the risk of privilege escalation and enhancing overall cluster security.

Acknowledgments

This project is supported by National Natural Science Foundation of China (Grant No.92467201) and the Youth Innovation Promotion Association CAS (Grant No.Y2023047)

References

[1] CNCF, "Kubernetes," 2024. [Online]. Available: https://kubernetes.io/
[2] Flexera, "State of the cloud report," 2021. [Online]. Available: https://www.flexera.com/about-us/press-center/flexera-releases-2021-state-of-the-cloud-report
[3] CNCF, "Cncf native landscape," 2024. [Online]. Available: https://landscape.cncf.io/?group=projects-and-products&view-mode=grid
[4] ——, "Prometheus," 2024. [Online]. Available: https://prometheus.io/
[5] I. Authors, "Istio," 2024. [Online]. Available: https://istio.io/
[6] CNCF, "Kubevirt," 2024. [Online]. Available: https://kubevirt.io/
[7] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, and J. Ma, "Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3048–3062.

[8] Y. Avrahami and S. B. Hai, "Kubernetes privilege escalation: Container escape == cluster admin?" [Online]. Available: https://www.blackhat.com/us-22/briefings/schedule/#kubernetes-privilege-escalation-container-escape--cluster-admin-26344

[9] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, "Co-residency attacks on containers are real," in *Proceedings of the 2020 ACM SIGSAC conference on cloud computing security workshop*, 2020, pp. 53–66.

[10] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration," *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 420–431, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:245424736

[11] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, pp. 1–36, Oct 2023. [Online]. Available: http://dx.doi.org/10.1145/3579639

[12] S. I. Shamim, "Mitigating security attacks in kubernetes manifests for security best practices violation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug 2021. [Online]. Available: http://dx.doi.org/10.1145/3468264.3473495

[13] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation," *ArXiv*, vol. abs/2405.19954, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:270123623

[14] Y. Gu, X. Tan, Y. Zhang, S. Gao, and M. Yang, "Epscan: Automated detection of excessive rbac permissions in kubernetes applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 11–11.

[15] N. V. Database, "Cve-2023-26484," 2023. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-26484

[16] P. A. Networks, "Rbac policy," 2024. [Online]. Available: https://github.com/PaloAltoNetworks/rbac-police

[17] Anonymous, "Rbaclock," 2024. [Online]. Available: https://sites.google.com/view/rbaclock/home

[18] Kubernetes, "Kubernetes components," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[19] ——, "kube-apiserver," 2024. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[20] etcd Authors, "etcd," 2024. [Online]. Available: https://etcd.io/

[21] Kubernetes, "kube-controller-manager," 2024. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

[22] ——, "Kubernetes scheduler," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[23] ——, "Kubelet," 2024. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[24] Docker, "Docker," 2024. [Online]. Available: https://www.docker.com/

[25] CNCF, "Containerd," 2024. [Online]. Available: https://containerd.io/

[26] Kubernetes, "Kube proxy," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/#kube-proxy

[27] ——, "Pod," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[28] ——, "Kubernetes resource," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[29] ——, "Custom resource," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[30] ——, "Custom resource definition," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[31] ——, "Rbac," 2024. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[32] ——, "Rbac best practices," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/security/rbac-good-practices/

[33] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, p. 87, 2023.

[34] GitHub, "Codeql," 2024. [Online]. Available: https://codeql.github.com/

[35] pgvector Authors, "pgvector," 2025. [Online]. Available: https://github.com/pgvector/pgvector

[36] Kubernetes, "Mutatingwebhookconfiguration," 2024. [Online]. Available: https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/extend-resources/mutating-webhook-configuration-v1/

[37] A. Cloud, "Alibaba cloud container service for kubernetes," 2024. [Online]. Available: https://www.alibabacloud.com/en/product/kubernetes

[38] Alibaba, "Alibaba security response center," 2024. [Online]. Available: https://security.alibaba.com/top.htm

[39] T. Cloud, "Tencent kubernetes engine," 2024. [Online]. Available: https://www.tencentcloud.com/products/tke

[40] Tencent, "Tencent security response center," 2024. [Online]. Available: https://en.security.tencent.com/

[41] Kubernetes, "Daemonset," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[42] K. R. Shahapure and C. Nicholas, "Cluster quality analysis using silhouette score," in *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, 2020, pp. 747–748.

[43] K. Authors, "Kata containers," 2025. [Online]. Available: https://katacontainers.io/

[44] Google, "gvisor," 2025. [Online]. Available: https://gvisor.dev/

[45] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, "Full-stack vulnerability analysis of the cloud-native platform," *Computers & Security*, vol. 129, p. 103173, 2023.

[46] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, "Container orchestration honeypot: Observing attacks in the wild," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 381–396.

[47] M. Shamim, F. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *Cornell University - arXiv,Cornell University - arXiv*, Jun 2020.

[48] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–36, 2023.

[49] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE transactions on parallel and distributed systems*, vol. 32, no. 3, pp. 674–691, 2020.

[50] Y. Lin, O. Tunde-Onadele, and X. Gu, "Cdl: Classified distributed learning for detecting security attacks in containerized applications," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 179–188. [Online]. Available: https://doi.org/10.1145/3427228.3427236

[51] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches," *Engineering reports*, vol. 1, no. 5, p. e12080, 2019.

[52] M. Haque, M. Kholoosi, and M. Babar, "Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration."

[53] Armo, "Kubescape," 2024. [Online]. Available: https://kubescape.io/

[54] Bridgecrew, "Checkov," 2024. [Online]. Available: https://www.checkov.io/

[55] Datree, "Datree," 2024. [Online]. Available: https://www.datree.io/

[56] Checkmarx, "Kics," 2024. [Online]. Available: https://github.com/Checkmarx/kics

[57] A. Security, "Kube-bench," 2024. [Online]. Available: https://github.com/aquasecurity/kube-bench

[58] StackRox, "Kube-linter," 2024. [Online]. Available: https://github.com/stackrox/kube-linter

[59] Zegl, "Kube-score," 2024. [Online]. Available: https://github.com/zegl/kube-score

[60] Shopify, "Kubeaudit," 2024. [Online]. Available: https://github.com/Shopify/kubeaudit

[61] CyberArk, "Kubiscan," 2024. [Online]. Available: https://github.com/cyberark/KubiScan

[62] A. Security, "Kube-hunter," 2024. [Online]. Available: https://github.com/aquasecurity/kube-hunter

[63] KubeArmor, "Kubearmor," 2024. [Online]. Available: https://kubearmor.io/

[64] Sysdig, "Falco," 2024. [Online]. Available: https://falco.org/

[65] ——, "Falco talon," 2024. [Online]. Available: http://github.com/falcosecurity/falco-talon

[66] Bytedance, “Elkeid,” 2024. [Online]. Available: https://github.com/bytedance/Elkeid

[67] H. M. Makrani, H. Sayadi, N. Nazari, K. N. Khasawneh, A. Sasan, S. Rafatirad, and H. Homayoun, “Cloak co-locate: Adversarial railroading of resource sharing-based attacks on the cloud,” in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, Sep 2021, p. 1–13. [Online]. Available: http://dx.doi.org/10.1109/seed51797.2021.00011

[68] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in paas clouds,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 990–1003. [Online]. Available: https://doi.org/10.1145/2660267.2660356

[69] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, “Co-residency attacks on containers are real,” in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, Nov 2020. [Online]. Available: http://dx.doi.org/10.1145/3411495.3421357

[70] C. Fang, H. Wang, N. Nazari, B. Omidi, A. Sasan, K. N. Khasawneh, S. Rafatirad, and H. Homayoun, “Repttack: Exploiting cloud schedulers to guide co-location attacks,” in *Proceedings 2022 Network and Distributed System Security Symposium*, ser. NDSS 2022. Internet Society, 2022. [Online]. Available: http://dx.doi.org/10.14722/ndss.2022.23149

[71] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, “Homealone: Co-residency detection in the cloud via side-channel analysis,” in *2011 IEEE Symposium on Security and Privacy*, May 2011. [Online]. Available: http://dx.doi.org/10.1109/sp.2011.31

[72] C. Fang, N. Nazari, B. Omidi, H. Wang, A. Puri, M. Arora, S. Rafatirad, H. Homayoun, and K. N. Khasawneh, “Heteroscore: Evaluating and mitigating cloud security threats brought by heterogeneity,” in *Proceedings 2023 Network and Distributed System Security Symposium*, Jan 2023. [Online]. Available: http://dx.doi.org/10.14722/ndss.2023.24996

[73] M. Thabet, B. Hnich, and M. Berrima, “Investigating, quantifying and controlling the co-location attack’s conditional value at risk of vm placement strategies,” *Future Generation Computer Systems*, vol. 149, pp. 464–477, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002868

[74] M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, “Securing container-based clouds with syscall-aware scheduling,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 812–826. [Online]. Available: https://doi.org/10.1145/3579856.3582835

[75] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang, *Incentive Compatible Moving Target Defense against VM-Colocation Attacks in Clouds*, Jan 2012, p. 388–399. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30436-1_32

[76] Y. Azar, S. Kamara, I. Menache, M. Raykova, and F. Shepherd, “Co-location-resistant clouds.” *IACR Cryptology ePrint Archive,IACR Cryptology ePrint Archive*, Jan 2014.

[77] J. Han, W. Zang, S. Chen, and M. Yu, *Reducing Security Risks of Clouds Through Virtual Machine Placement*, Jan 2017, p. 275–292. [Online]. Available: https://doi.org/10.1007/978-3-319-61176-1_15

[78] S. Yu, X. Gui, F. Tian, P. Yang, and J. Zhao, “A security-awareness virtual machine placement scheme in the cloud,” in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 1078–1083.

[79] J. Han, W. Zang, M. Yu, and R. Sandhu, “Quantify co-residency risks in the cloud through deep learning,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1568–1579, Jul 2021. [Online]. Available: http://dx.doi.org/10.1109/tdsc.2020.3032073

[80] Y. Han, J. Chan, T. Alpcan, and C. Leckie, “Using virtual machine allocation policies to defend against co-resident attacks in cloud computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 95–108, 2017.

TABLE V
CUSTOM RESOURCES FROM EXPERIMENTAL APPLICATIONS WITH LINKS TO KUBERNETES BUILT-IN RESOURCES

Custom Resource	Application	Builtins across
Applications	KubeVela	secrets
VirtualMachines	KubeVirt	secrets, services, validatingwebhookconfigurations
VirtualMachineInstances	KubeVirt	pods, secrets, services, jobs, serviceaccounts, networkpolicies, clusterrolebindings, clusterroles
VirtualMachineInstanceMigrations	KubeVirt	jobs
AppProjects	Argo	secrets
KeptnTasks	Keptn	jobs
BroadcastJobs	OpenKruise	pods
ContainerRecreateRequests	OpenKruise	pods
StatefulsetLikeTests	OpenKruise	pods
AdvancedCronjobs	OpenKruise	jobs
ElasticQuotas	Koordinator	pods
Reservations	Koordinator	pods,nodes
AuthPolicies	Kuadrant	services, deployments
RateLimitPolicies	Kuadrant	services, deployments
IstioKinds	Istio	pods
ProviderRevisions	Crossplane	secrets, clusterroles
CompositeresourceDefinitions	Crossplane	clusterroles
FunctionRevisions	Crossplane	secrets
ClusterCloudEventSources	KEDA	jobs
ClustertriggerAuthentications	KEDA	jobs
ScaledJobs	KEDA	jobs
TriggerAuthentications	KEDA	jobs
CloudEventSources	KEDA	jobs
Certificates	cert-manager	secrets
ClusterIssuers	cert-manager	certificatesigningrequests, secrets
Issuers	cert-manager	certificatesigningrequests, secrets
Challenges	cert-manager	pods, ingresses
PushSecrets	external-secrets	secrets
ExternalSecrets	external-secrets	secrets
ResourceImports	Antrea	services
Backups	k8up	deployments
Actionsets	Kanister	pods
RepositoryServers	Kanister	services

TABLE VI
SECURITY METRICS COMPARISON BETWEEN *RBAClock* AND THE DEFAULT SCHEDULER UNDER VARYING NODE SCALES

Node Num	Default Scheduler				RBAClock			
	ERP	Aggregated Risk	Privilege Escalation	Privileged Node Proportion	ERP	Aggregated Risk	Privilege Escalation	Privileged Node Proportion
28	15600	830	17	0.852	3900	578	4	0.593
27	12370	732	14	0.769	0	474	0	0.5
26	13815	773	16	0.84	0	476	0	0.52
25	15916	774	18	0.875	4469	557	7	0.625
24	12068	812	16	0.826	21	449	3	0.522
23	19930	938	24	1	6683	642	13	0.773
22	15886	777	20	0.905	14	496	2	0.619
21	18490	857	24	1	21	501	3	0.65
20	17770	877	24	1	28	468	4	0.632
19	16729	731	21	0.944	35	474	5	0.667
18	15303	786	23	0.941	35	441	5	0.647
17	17028	748	24	1	42	407	6	0.625
16	16367	693	24	1	35	373	5	0.6
15	14645	719	24	1	42	338	6	0.571
14	16430	693	24	1	35	346	5	0.615
13	13852	656	24	1	587	355	9	0.667
12	14297	639	24	1	56	274	8	0.545
11	11711	593	24	1	70	284	10	0.6
10	11439	560	24	1	63	246	9	0.556
9	9850	479	24	1	56	207	8	0.5
8	9130	470	24	1	49	162	7	0.489
7	8410	351	24	1	70	119	10	0.433
6	7231	318	23	1	982	128	8	0.6
5	6826	285	24	1	3769	218	13	0.75
4	5920	258	23	1	2948	176	23	0.667
3	5530	231	24	1	5530	231	24	1
2	4276	201	24	1	4276	201	24	1