Understanding Resource Injection Vulnerabilities in Kubernetes Ecosystems

Defang Bo[1,2†‡], Jie Lu[3], Feng Li[1,2*†‡], Jingting Chen[1†‡],
Jinchen Wang[1,2†‡], Chendong Yu[1,2†‡], Yeting Li[1,2†‡], Wei Huo[1,2†‡],
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]SKLP, Institute of Computing Technology, CAS, Beijing, China
{bodefang, lifeng, chenjingting, wangjinchen, yuchendong, liyeting, huowei}@iie.ac.cn, lujie@ict.ac.cn

***Abstract*—Cloud-native technologies have revolutionized application development, with Kubernetes emerging as the de facto standard platform for containerization and orchestration. Kubernetes manages applications through API objects called resources, where users declare desired states via resource definitions that are processed by controllers to reconcile system discrepancies. However, this resource-based architecture introduces resource injection vulnerabilities, where controllers perform privileged operations using user-controllable fields without adequate validation. Attackers can exploit these weaknesses by injecting malicious content into resource fields to achieve unauthorized access and privilege escalation.**

In this paper, we conduct the first comprehensive study on 125 resource injection vulnerabilities from 8,306 Kubernetes-related vulnerabilities across common databases. For all studied vulnerabilities, we investigate their vulnerable fields, root causes, privileged operations, exploitation conditions, and fixing strategies. Our study reveals many interesting findings that can guide the detection and mitigation of resource injection vulnerabilities, as well as the development of more secure cloud-native applications.

***Index Terms*—Resource Injection Vulnerabilities, Kubernetes**

I. Introduction

Cloud-native technologies [1] have transformed application development for dynamic, scalable environments through containerization and orchestration. As the de facto standard, Kubernetes [2] is now adopted by over 96% of organizations [3], serving as the foundational platform for deploying and managing cloud-native applications.

Kubernetes manages applications and cluster components through API objects called **resources**—standardized, RESTful data structures that serve as declarative configuration specifications. These configuration objects are centrally managed by the *API Server* and stored in *etcd*. Users define the desired application state (*i.e.*, the desired configuration of deployed applications and infrastructure components) by declaring resources via definitions in YAML or JSON files. When users submit these **resource definitions** to the API Server, it validates and persists the configuration objects, then notifies the corresponding **controllers**—specialized components that continuously monitor their assigned resource kinds and automatically reconcile discrepancies between the actual system state and the desired state. For example, if a deployment specifies three *Pod* replicas, the controller ensures exactly three Pods are running by creating or terminating instances as necessary.

While Kubernetes' resource-based architecture provides significant flexibility, it introduces a new class of security vulnerabilities that we define as ***resource injection vulnerabilities***. These vulnerabilities emerge when controllers perform privileged operations using user-controllable fields from resource definitions without adequate validation. Attackers can exploit these weaknesses by injecting malicious content into resource definition fields, thereby subverting normal controller behavior to achieve various security impacts including unauthorized access and privilege escalation.

Existing research on Kubernetes security spans several directions. Prior work has examined operator bugs through detection [4]–[7] and root-cause analysis [8], investigated pod-level defects [9], and exposed permission vulnerabilities introduced by third-party applications [10], [11]. Among these efforts, security misconfigurations have received the most extensive attention. Empirical studies identify prevalent configuration weaknesses [12], examine mitigation strategies [13], document gaps between community-prescribed settings and real-world practice [14], and reveal under-reporting of security defects in configuration files [15]. These empirical findings have motivated a broad detection ecosystem, including rule-based [16]–[20], graph-based [21]–[23], and LLM-based tools [24]–[26]. However, these misconfiguration studies target errors in configuration content itself—such as hardcoded secrets, excessive RBAC permissions, and policy violations—which stem from administrator or user mistakes. Resource injection vulnerabilities, by contrast, arise from flaws in controller implementation: controllers fail to adequately validate user-controllable resource fields (e.g., image references, volume sources, or service account bindings) before consuming them in privileged operations, enabling adversaries to inject malicious content even when configurations are otherwise valid. Despite extensive work on misconfigurations

*Corresponding author.
†Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China
‡Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00220

and other vulnerability classes, systematic investigation of such controller-level validation flaws remains absent from the literature.

Addressing this research gap is crucial for the cloud-native ecosystem. Without systematic understanding of controller-level validation flaws, the community cannot develop targeted detection and mitigation strategies that complement existing configuration-focused approaches, nor can practitioners assess the prevalence and prioritize remediation of these vulnerabilities in real-world deployments.

In this paper, we conduct the first comprehensive study of resource injection vulnerabilities in Kubernetes ecosystems. Starting from 8,306 Kubernetes-related vulnerabilities collected from common vulnerability databases, we employed LLM-assisted filtering to identify 660 candidates (7.9%) requiring Kubernetes interaction for exploitation. Manual verification revealed that 125 of these (18.9%) constitute resource injection vulnerabilities—a significant proportion warranting dedicated investigation given their potential for cluster-wide compromise. We systematically analyze these vulnerabilities to answer five essential research questions:

- **RQ1 (Vulnerable Resource Fields):** What characteristics of resource fields contribute to these vulnerabilities?
- **RQ2 (Root Cause):** What are the root causes of resource injection vulnerabilities?
- **RQ3 (Privileged Operations and Impacts):** What types of privileged operations can be exploited through resource injection, and what impacts do they lead to?
- **RQ4 (Exploitation Conditions):** What conditions are required for vulnerability exploitation?
- **RQ5 (Fixing Strategies):** How do developers fix these vulnerabilities?

Through our in-depth investigation of resource injection vulnerabilities, guided by the aforementioned five research questions, we have obtained some interesting findings. The key findings from our analysis are summarized as follows.

- Vulnerable fields have two types of similarity patterns, i.e., intra-application similarity where fields with similar functions exhibit identical vulnerabilities (32.7%), and inter-application similarity where applications implementing identical functionalities share similar vulnerable fields (42.2%). These patterns motivate automated detection of potentially vulnerable fields in cloud-native ecosystems.
- 86.4% resource injection vulnerabilities stem from missing validation, suggesting that taint analysis can be used to identify these vulnerabilities. 44.0% of vulnerabilities are due to trust boundary shifts during cloud migration, where privileged operations (such as access to specific files and command execution) previously restricted to local administrators become accessible to cluster users through vulnerable fields. Developers often overlook these boundary shifts, highlighting the need to reassess threat models when migrating applications to cloud environments.
- 93.6% of resource injection vulnerabilities are medium-to-high severity or above, stemming from 217 privileged APIs. Since similar applications share functionally similar APIs (70.5%), motivating a similarity-based approach: identify comparable applications and leverage their known vulnerable APIs to discover similar APIs in new applications.
- More than 70% vulnerabilities involve a single field within one resource via a single API call, allowing fuzzing methods to focus on one field per resource during testing.
- Vulnerability fixes are implemented across three locations using three primary fix strategies. This finding motivates an approach that condenses these fix strategies into few-shot learning examples for the automated identification of sanitizers, which are subsequently utilized in taint analysis.

This paper makes the following key contributions:

- We present the first comprehensive study on resource injection vulnerabilities in Kubernetes ecosystems, opening new research directions for addressing these vulnerabilities.
- Our documented resource injection vulnerabilities serve as a benchmark dataset for future research on mitigating such threats in Kubernetes environments. Our collected vulnerabilities and analysis results are publicly available [27].

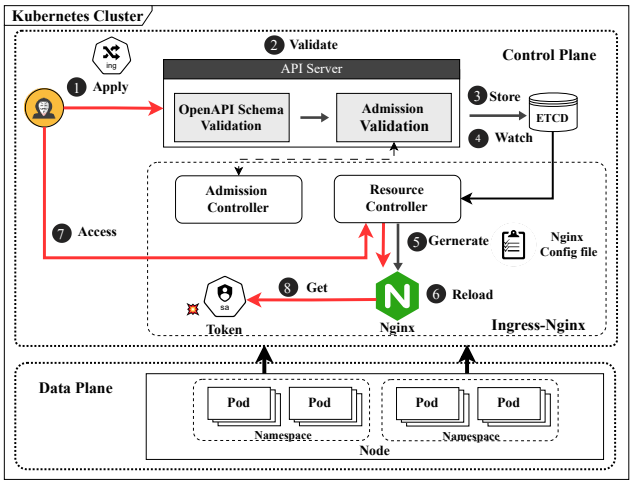


Fig. 1. Example of Ingress Resource Processing Workflow

II. Resource Injection and Threat Model

In this section, we describe Kubernetes' resource processing workflow using an example with two resource injection flaws and define the threat model.

A. Key Components for Processing Resource

As shown in Figure 1, a Kubernetes cluster is logically divided into two primary planes: the control plane and the data plane, each serving distinct operational functions within the cluster architecture.

The control plane manages applications through core Kubernetes built-in components including the API server, controller manager (not shown in Figure 1) and etcd, alongside third-party extensions implemented by application developers. The API server processes all resource operations, serving as the central gateway for creating, reading, updating, and deleting Kubernetes resources, while etcd stores all resources as the cluster's persistent data store. These extensions comprise two critical types: resource controllers and admission controllers.

Fig. 2. Example of Ingress resource definition containing two resource injection vulnerabilities(CVE-2021-25742 and CVE-2021-25745).

Resource controllers are components that manage applications by continuously monitoring their states and adjusting them to match the configuration specified in the resources. Meanwhile, admission controllers intercept API requests to perform validation and modification before persisting resources to etcd, ensuring resource compliance.

The data plane runs application containers through components (not shown in Figure 1) that primarily execute Pods on Nodes. Pods serve as abstractions that provide resource isolation for containers, including CPU, memory, storage, and network resources. Namespaces provide logical isolation between different workloads and tenants.

The control plane and data plane represent a fundamental architectural separation where the control plane defines desired application states through resources, while the data plane executes these applications accordingly. The control plane manages applications by creating, updating, and monitoring resources rather than directly controlling execution processes, allowing declarative specification of what should run and how it should be configured, while data plane components handle the actual scheduling and maintenance. This separation provides clear trust boundaries, isolating control logic from execution workloads and reducing the attack surface by limiting direct access to underlying infrastructure.

B. Resource Definitions

Figure 2 presents an example of a Kubernetes resource definition. The resource definition follows a structured format, demonstrated here using an `Ingress` resource from ingress-nginx [28]—a widely adopted controller that uses nginx as a reverse proxy and load balancer for external traffic routing. This resource is used to modify the routing configuration of a running nginx application.

The `metadata` section encompasses the resource identification and configuration information. This includes the `apiVersion` field (line 1) and `kind` field (line 2) which identifies this as an Ingress resource to notify the corresponding controller. The mandatory `name` field (line 4) uniquely identify this resource as `ingress-exploit`. The `annotations` subsection (lines 5-9) provides supplementary configuration directives, specifically the `nginx.ingress.kubernetes.io/configuration-snippet` annotation (lines 6-8) that contains custom nginx configuration creating a location block for the `/security/` URL path with an alias directive pointing to the `serviceaccount` directory.

The `spec` section (lines 10-25) defines the core routing configuration. Within the `rules` array (line 11), the `host` field (line 12) specifies that traffic to `nginx.com` should be processed by this ingress rule. The `http` subsection (line 13) contains path-based routing rules, where the `paths` array (line 14) defines specific routing behaviors. The `path` field (line 15) and `pathType` field (line 16) serve the same purpose as the `annotations` subsection, modifying nginx configuration with similar functionality. The `backend` subsection (lines 20-24) defines the destination service for request forwarding, but is not discussed in the following section, as it is not related to vulnerabilities.

The `status` section (line 26) records the resource's current state and is maintained automatically by Kubernetes.

C. Resource Processing Workflow in Kubernetes

Figure 1 illustrates the complete processing workflow for applying resources to Kubernetes, using the resource definition in Figure 2 as an example. ❶ Users apply the resource definition to the API server. ❷ The API server first performs schema validation to verify the structural format and required fields of the resource definition, ensuring that essential elements such as `apiVersion` and`kind` are properly specified. Admission controllers invoked by the API server then conduct deeper validation checks on the resource specifications. ❸ Upon successful validation, the resource definition is stored in the etcd database. ❹ The corresponding resource controller, which has registered to monitor that specific resource type, detects the new resource through the watch mechanism of etcd—in this case, the Ingress controller specifically watches for resources with `kind: Ingress`. ❺ The controller generates the nginx configuration and updates it through a sidecar container that shares the same Pod with nginx. ❻ Nginx reloads the configuration file to update its routing rules.

D. Real-world Resource Injection Vulnerabilities

Figure 2 illustrates real-world resource injection vulnerabilities in Kubernetes Ingress-Nginx, where the resource definition contains two fields that have been injected with malicious content, leading to CVE-2021-25742 and CVE-2021-25745. These vulnerabilities demonstrate how attackers can exploit resource injection to compromise cluster security.

1) CVE-2021-25742: The first vulnerability exploits the `configuration-snippet` annotation within the `metadata` section. As shown in the resource definition, the attacker adds a custom location block that maps a public URL path `/security/` to the local service account token directory `/var/run/secrets/kubernetes.io/serviceaccount/` using the `alias` directive.

When the resource undergoes the complete processing workflow (❶–❻) illustrated in Figure 1, the Ingress-Nginx controller fails to sanitize this user-provided annotation and directly incorporates the `alias` directive into

the generated configuration file. After Nginx reloads with the compromised configuration, ❼ the attacker can access `nginx.com/security/`. ❽ Nginx follows the `alias` directive and returns the service account token with admin privileges to the attacker. The attacker can then use this token to compromise the entire cluster.

2) CVE-2021-25745: This vulnerability demonstrates a similar attack vector but exploits the `path` field within the `spec` section. The developers overlooked that this field also has similar functionality and can be injected with malicious nginx directives that are ultimately written to the configuration file without proper sanitization.

E. Threat Model

1) Attacker Assumptions: We consider attackers with legitimate but limited Kubernetes resource deployment permissions who can create or modify specific resources through kubectl, APIs, or client libraries. These attackers include cluster operators with restricted privileges, application developers with namespace-scoped access, and external attackers who gain limited cluster access through container escape or other vulnerabilities.

2) Attack Process: Attackers exploit user-controllable fields in resource definitions to inject malicious content. The attack workflow involves crafting malicious resource definitions, which controllers process without adequate input sanitization. The malicious content is then incorporated into generated configurations that underlying systems execute, allowing attackers to achieve their objectives.

3) Attack Impact: By deploying maliciously injected resource configurations, attackers can cause unintended cloud-native application or controller behaviors, breaking the trust boundaries within Kubernetes. For instance, when certain attributes of Pods in the data plane are injected with malicious content, controllers read this malicious content during management operations and perform corresponding actions, thereby breaking the trust boundary from the data plane to the control plane, leading to privilege escalation, information disclosure, service disruption, and other security violations.

III. Methodology

A. Collecting Resource Injection Vulnerabilities

To identify resource injection vulnerabilities, we employed a three-phase methodology. First, we conducted keyword-based searches across applications and built-in Kubernetes components. Second, we used vulnerability patch analysis enhanced by large language models (LLMs) to filter out false positives. Finally, we manually verified remaining candidates to confirm authentic resource injection vulnerabilities. The vulnerability distribution across phases is shown in Table I.

1) Keyword-Based Search: Our keyword sources comprised two categories: (1) Application names from the Cloud Native Computing Foundation (CNCF) ecosystem [29]. CNCF provides a vendor-neutral repository of cloud-native projects that interact with Kubernetes resource requests—a prerequisite for resource injection vulnerabilities. We manually examined application documentation, source code, and configuration files to identify Kubernetes integration relationships, yielding 519 relevant projects from 677 candidates. These applications spanned six categories covering the complete cloud-native technology lifecycle. (2) Kubernetes-related terminology (including "kubernetes", "k8s", "container orchestration", and "kubectl") to identify vulnerabilities in non-CNCF applications and Kubernetes built-in components.

TABLE I
VULNERABILITY COLLECTION RESULTS

Keyword Category	# Phase 1	# Phase 2	# Phase 3
Application name	7858	507	110
Kubernetes-related terminology	448	153	15
Total	8306	660	125

We searched these keywords across multiple vulnerability databases: MITRE CVE Database [30], NVD [31], CVEDetails [32], and OSV [33]. These yielded 8,306 potential vulnerabilities: 7,858 from application names and 448 from Kubernetes terminology.

2) Preliminary Filtering with Large Language Models: Given the impracticality of manually reviewing 8,306 vulnerabilities, we implemented preliminary filtering using LLMs. The vulnerabilities fell into two categories: self-contained vulnerabilities exploitable without Kubernetes interaction (*e.g.*, CVE-2024-45806, where Envoy allows external clients to manipulate header information for unauthorized data access), and interaction-dependent vulnerabilities requiring Kubernetes communication. For each vulnerability, we collected descriptions and extracted links from CVE descriptions to locate discussions and patches using established methods [34], [35]. We submitted this information to DeepSeek-R1 [36] to identify vulnerabilities belonging to the first category and filtered them out. The prompts are available in our repository [27]. This process reduced our dataset to 660 potential resource injection vulnerabilities.

3) Manual Verification: For 660 vulnerabilities, we manually examined their patches. When patches were unavailable, we compared commits between vulnerable and patched versions. This process involved identifying the specific commits that implemented the actual security fixes. Vulnerabilities for which patches could not be retrieved despite these efforts were excluded from further analysis. For vulnerabilities with accessible patches, we performed detailed source code analysis based on the patch content to determine whether they constituted genuine resource injection vulnerabilities. This rigorous verification process, which required an investment of three person-months, ultimately yielded 125 confirmed resource injection vulnerabilities.

B. Analyzing Resource Injection Vulnerabilities

To address our five research questions, we analyzed the 125 confirmed resource injection vulnerabilities based on their patches and source code. We documented detailed vulnerability scenarios and categorized them by: vulnerable field characteristics, root causes, privileged operations, security impacts, fixes.

We employed an open card sorting methodology for categorization, where categories emerged through collaborative analysis rather than being pre-defined—an approach widely used in empirical bug studies [8], [9], [37]–[39]. Three researchers independently classified all 125 vulnerabilities and cross-reviewed each other's classifications. When disagreements arose, we consulted a fourth expert to facilitate discussion until consensus was reached, following established vulnerability analysis practices [40], [41]. This comprehensive categorization effort required an additional three person-months to complete.

Throughout the analysis of these 125 vulnerabilities, each author identified noteworthy findings that warranted further investigation. We conducted collaborative discussions to synthesize these findings and derive lessons learned and implications for existing approaches to combating resource injection vulnerabilities. Our analysis reveals important temporal patterns in resource injection vulnerabilities. These 125 vulnerabilities span evenly from 2016 to 2025, indicating a persistent rather than transient security challenge.

C. Threats to Validity

Similar to other vulnerability studies [8], [9], [41], potential threats include representativeness of studied applications and vulnerabilities, accuracy of our analysis methodology, and replicability and generalizability.

Generalizability. Cloud-native systems share declarative resource management patterns where controllers process user-defined YAML configurations. Both open-source (Kubernetes, OpenShift) and commercial platforms (AWS EKS, Google GKE) adopt similar controller-based reconciliation mechanisms. The fundamental vulnerability pattern of inadequately validated user-controllable fields manifests across platforms following this paradigm.

Representativeness. Kubernetes is the dominant container orchestration platform. Our study covers 519 CNCF projects across six categories, representing widely adopted cloud-native technologies. The 125 confirmed vulnerabilities from our three-phase methodology provide comprehensive coverage of resource injection patterns.

Analysis methodology. We employed a rigorous three-phase approach: keyword-based search across CVE databases, LLM-based filtering using DeepSeek-R1, and manual verification requiring three person-months. Three researchers independently classified vulnerabilities, with conflicts resolved through discussion. We excluded cases without accessible patches to ensure quality. To ensure LLM accuracy, we conducted small-scale testing before large-scale deployment.

Replicability. Our classification follows systematic criteria based on our threat model. We make our dataset, classification criteria, and analysis results publicly available to enable validation and extension [27].

IV. Vulnerable Resource fields(RQ1)

The 125 vulnerabilities involve 147 vulnerable fields from 55 different resources, with one vulnerability containing multiple fields. We categorize these fields by location and function.

A. Location Characteristics

Vulnerable fields are predominantly located in the *spec* section (74.1%) and *metadata* section (23.8%), while the *status* section contains only a few vulnerable fields (2.1%).

The high vulnerability rate observed in the *spec* section aligns with expectations, as these fields are typically user-configurable and thus more susceptible to security issues. Similarly, the *metadata* section exhibits relatively few vulnerabilities overall, as most of its fields are immutable by design. In contrast, the *status* section contains the fewest vulnerable fields because these status fields are typically read-only and cannot be modified by cluster users. However, some applications grant cluster users elevated privileges to modify status fields, leading to vulnerabilities.

Within the metadata section (26 vulnerabilities total), we identify three vulnerable field categories: (1) Annotations (20 vulnerabilities): Provide flexible key-value storage and are susceptible to injection attacks, such as vulnerabilities in Figure 2. (2) Labels (3 vulnerabilities): While typically harmless, certain controllers like Cilium use labels to generate security policies. Attackers can inject malicious content to bypass restrictions, leading to vulnerabilities like CVE-2023-39347. (3) Metadata fields (3 vulnerabilities): Include system-managed fields like UIDs and ownerReferences that should be immutable. For instance, CVE-2025-1098 exposes an admission controller endpoint in Ingress-nginx that accepts *AdmissionReview* requests directly, bypassing API server validation. This allows users to modify UIDs—fields enforced as immutable by the API server—through this alternate endpoint [42].

Three vulnerable fields in the *status* section stem from erroneously granting cluster users modification permission. For example, in CVE-2020-8554, the `status.loadBalancer.ingress.ip` field should be automatically maintained by cloud provider controllers, but permission misconfiguration inadvertently granted non-privileged users write access to the `status/loadBalancer` field, enabling malicious content injection to hijack traffic.

Finding 1: *74.1% of vulnerable fields are located in the spec section, 23.8% in the metadata section, and 2.1% in the status section.*

B. Functional Characteristics

We categorize the functional characteristics of each field based on its specific utilization within the component implementation. This classification encompasses three distinct categories: application-specific functional fields, traditional functional fields, and resource references.

1) Application-Specific functional fields (88/147, 59.9%): Different applications implement unique functionalities that introduce application-specific vulnerable fields. Despite their diversity, we observe two characteristics: (1) **Intra-application similarity** (48/147, 32.7%)—Fields with similar functions exhibit identical vulnerabilities. As demonstrated by the vulnerabilities in Figure 2, the `configuration-snippet` field

and the `path` field can both be exploited for resource injection attacks; (2) **Inter-application similarity** (62/147, 42.2%)—applications implementing identical functionalities share similar vulnerable fields. For example, both *CRI-O* and *containerd* implement Pod lifecycle hooks via `lifecycle.preStop` and `lifecycle.postStart` fields, both accepting command inputs and causing command injection vulnerabilities (CVE-2022-1708 and CVE-2022-31030). Note that the sum of intra-application similarity (48) and inter-application similarity (62) exceeds the total number of application-specific functional fields (88) because some fields exhibit both similarity characteristics simultaneously.

Finding 2: *Application-specific fields exhibit two characteristics: intra-application similarity (32.7%) and inter-application similarity (42.2%).*

2) Traditional functional fields (43/147, 29.2%): Traditional functional fields remain significant attack vectors in Kubernetes resource injection scenarios, as Kubernetes, built upon traditional software stacks, inherently incorporates conventional functional capabilities from its foundational layers. *Path-Type Fields* (33/147, 22.4%) serve resource location functions but inherit traditional path-based vulnerabilities: file paths enable container escape through path traversal attacks (*e.g.*, `../../../etc/passwd` accessing host filesystem), network URLs facilitate server-side request forgery targeting internal Kubernetes APIs, and external resource references (*e.g.*, container images `registry/namespace/repository:tag`) introduce supply chain risks by allowing malicious image injection. *Command Execution Fields* (10/147, 6.8%) represent input vectors triggering command execution within containerized workloads, including Pod startup commands, container lifecycle hooks, environment variable injection, and init container scripts that can compromise pod security boundaries and cluster integrity.

Finding 3: *Traditional functional fields cause resource injection vulnerabilities: path-type fields (22.4%) and command execution fields (6.8%).*

3) Resource References (16/147, 10.9%): Kubernetes allows resource references, which enable one resource to reference and utilize data from another resource within the cluster. For example, *Pods* can reference *Secrets* for sensitive data and Deployments can reference ConfigMaps for configuration.

However, resource references introduce two critical security challenges that lead to vulnerabilities. First, they create ambiguity regarding validation responsibilities between referencing and referenced resources, as will be discussed in Section V-A2a. Second, resource references can bypass Kubernetes' built-in security policies when developers fail to verify whether attackers have appropriate permissions to access the referenced resources, as will be analyzed in Section V-A2b.

TABLE II
ROOT CAUSES OF RESOURCE INJECTION VULNERABILITY

Root Cause			# Vuln
Missing Validation	Trust Boundary Shift	Local to Cloud	55
		Internal to External	4
	Validation Responsibility Confusion	Over-Reliance on Other Resource	12
		Over-Reliance on Kubernetes Security Control	8
		Complex Resource Fields	29
Wrong Validation	Overlooked Input Content		13
	Overlooked Input Types		1
	Developer Coding Mistakes		3

Finding 4: *10.9% vulnerable fields are referenced.*

V. ROOT CAUSE(RQ2)

As shown in Table II, our analysis identified two main categories of root causes. Missing validations represent the majority at 86.4% (108/125) of all vulnerabilities, while wrong validations account for the remaining cases. Our classification is based on multi-dimensional analysis that examines not only code-level implementation issues but also contextual factors, architectural challenges, and human factors that contribute to these validation failures in cloud-native environments.

Finding 5: *Missing validations constitute the majority (86.4%) of root causes for resource injection vulnerabilities, with wrong validations (13.6%) accounting for the remainder.*

A. Missing Validation

Through analysis of vulnerability patches, corresponding source code, and related user discussions, we identified three primary reasons why developers forget to validate resource fields: trust boundary shifts, check responsibility confusion, and complex resource fields. These patterns account for 86.4% of the identified vulnerabilities in our dataset.

1) Trust Boundary Shifts (59/125, 47.2%): These shifts occur when the operational environment of the software changes, causing previously safe assumptions about data sources to become invalid. In our case, we identified two main categories: trust boundary shifts from local environments to cloud deployments, and transitions from internal to external accessibility. Developers often fail to adapt their validation strategies to account for these new trust boundaries, leaving resource fields unchecked.

a) Local to Cloud (55/125, 44.0%): When applications migrate from local to cloud-native environments, trust boundaries shift, but developers often fail to recognize this change and forget to add necessary validation. Take the vulnerabilities in Figure 2 as an example. In local environments, Nginx configuration files were only accessible to trusted administrators, so input validation was unnecessary. However, after migrating to the cloud, local administrators became untrusted cloud users who modify configurations through resource definitions, while trusted resource controllers with cloud administrative privileges execute these changes to configuration files. Since modifications now originate from untrusted cloud users, validation becomes essential. However, developers failed to recognize this trust boundary shift and did not implement necessary

validation. Similar issues affect other functions previously restricted to local administrators, such as file reading (CVE-2022-40607) and host path mounting (CVE-2023-3955).

b) Internal to External (4/125, 3.2%): As Kubernetes components evolve, interfaces originally designed for internal cluster access are increasingly exposed to external users, creating critical trust boundary shifts. Internal access models rely on mutual trust among cluster administrators, eliminating the need for rigorous input validation. However, when these interfaces are exposed to external users, developers often fail to implement the additional security measures required for this new threat model. CVE-2025-1098, detailed in Section IV-A, exemplifies this issue, where the ingress-nginx admission controller exposes `AdmissionReview` functionality externally, allowing modification of resource UIDs.

Finding 6: *47.2% of vulnerabilities result from overlooking trust boundary shifts: from local to cloud and from internal to external.*

2) Validation Responsibility Confusion (20/125, 16.0%): As discussed in Section IV-B3, when resources reference another one, it becomes ambiguous which controller is responsible for validation. This uncertainty often leads developers to make incorrect assumptions and fail to perform validation.

a) Over-Reliance on Other Resources (12/125, 9.6%): Developers building a controller for a referencing resource might assume that any referenced resource has already undergone thorough content checks by its respective controller. Conversely, the controllers for these referenced resources may implement less rigorous validation, not anticipating that their data fields could be propagated into critical operations. Take CVE-2021-41254 for example, where a *Secret* was referenced by a *ServiceAccount*, which was then referenced by *Kustomization* resources. However, the *Secret* contained malicious shell scripts, and none of the controllers for these three resources performed validations. Consequently, the Kustomization controller executed the malicious script.

b) Over-Reliance on Kubernetes Security Controls (8/125, 6.4%): Application components often assume that Kubernetes built-in security policies provide sufficient protection, leading developers to neglect implementing permission checks. CVE-2024-43803 demonstrates this issue: attackers referenced *Secret* resources in their resource fields, and the controller failed to validate whether users had access permissions for these referenced *Secret* resources, resulting in attackers gaining unauthorized access to resources in other namespaces.

Finding 7: *16.0% of studied vulnerabilities are due to validation responsibility confusion, which comes from over-reliance on other resources and Kubernetes security controls.*

3) Complex Malicious Content Sources (29/125, 23.2%): Kubernetes environments face threats from diverse external sources due to extensive integrations with container registries, Git repositories, cloud APIs, and other third-party systems. This broad attack surface significantly expands the potential origins of malicious content. Moreover, the final malicious inputs often result from complex assembly processes involving multiple components and dynamic transformations, making it extremely difficult for developers to anticipate all possible attack scenarios, thus leading to inadequate validation measures. Take CVE-2022-31036 for example, where attackers submitted symlinks in Git repositories pointing to arbitrary server files, which Argo CD then read and displayed in its interface without proper validation. Similarly, CVE-2021-37914 demonstrated how crafted `-p` arguments in Argo Workflows, after undergoing complex assembly and transformation processes during workflow submission, could overwrite critical fields like `env`, enabling unauthorized command execution.

B. Wrong Validation

The complex external inputs in Kubernetes environments not only lead to missing validation (as discussed in Section V-A3) but also result in wrong validation implementations. This complexity makes it difficult for developers to anticipate all attack vectors and implement appropriate validation. Consequently, even when developers validate fields, they may implement inadequate validation that fails to address potential attacks.

1) Overlooked Input Content (13/125, 10.4%): Developers often fail to comprehensively analyze the diverse content structures within input fields when implementing validation mechanisms. CVE-2022-24731, which represents a bypass of the previously patched CVE-2022-24348, exemplifies this vulnerability pattern through path traversal exploitation. The original validation mechanism limited recursion depth (*e.g.*, `maxDepth=10`) but failed to implement adequate root directory boundary checks on resolved paths. This design flaw allowed attackers to construct multi-layer path traversal sequences (*e.g.*, `app/charts/../../../../etc/passwd`) that ultimately resolved to paths outside the intended repository root directory, thereby enabling unauthorized access to sensitive system files.

2) Overlooked Input Types (1/125, 0.8%) and Developer Coding Mistakes (3/125, 2.4%): Certain resource fields can simultaneously accept different input types, causing parsing functions to misinterpret inputs and bypass validation (CVE-2022-24348). Simple development oversights, such as incorrect implementation of filtering logic, can also lead to resource injection vulnerabilities (CVE-2023-6476, CVE-2020-8945).

Finding 8: *Wrong validations are due to overlooked contents(10.4%), types(0.8%) and developer mistakes(2.4%).*

VI. Privileged Operations and Impacts(RQ3)

In this section, we analyze privileged operations and their corresponding security impacts.

TABLE III
PRIVILEGED OPERATION TYPES DISTRIBUTION

Privileged Operation Type	# Traditional API	# Application API	Total
Command Execution	15	21	36
File System Operation	31	22	53
Network Operation	9	11	20
Kubernetes Resource Access	16	23	39
Security Configuration Generation	1	31	32
Format Parsing	18	19	37
Total	**90**	**127**	**217**

A. Types of Privileged Operations

An API is defined as a sink point where taint data executes dangerous operations. Following established methodologies [43], [44], we classify certain APIs in Kubernetes and applications as privileged operations based on their potential security implications. We extracted APIs from each of the 125 vulnerabilities, identifying a total of 217 API instances (not deduplicated, as the same API may be involved in multiple vulnerabilities). These APIs are categorized into six types according to their functionality, as shown in Table III.

The majority of privileged operations involve file system operations (24.4%), Kubernetes resource access (18.0%), and format parsing (17.1%). The prevalence of file system operations stems from path-type fields (22.4% of vulnerable fields, as shown in Finding 3), which require file system APIs for processing. Kubernetes resource access encompasses resource CRUD operations and resource relationship handling (as discussed in Section IV-B3). Format parsing APIs process complex formats such as JSON and YAML from resource field contents, where improper parsing can result in denial-of-service (DoS) attacks. Additionally, command execution (16.6%), security configuration generation (14.7%), and network operations (9.2%) constitute other categories of privileged operations.

These privileged operations fall into two categories: traditional APIs (41.5%) and application-specific APIs (58.5%). Traditional APIs directly invoke standard libraries, such as os.Open() for file operations and exec.Command() for command execution. Application-specific APIs are either fully custom-developed functions or wrapper functions that encapsulate traditional APIs. When an application-specific API invokes a traditional API, it is classified as application-specific. We found that functionally similar applications share similar APIs, with 70.5% satisfying this pattern.

Finding 9: *Privileged operations comprise six types in two categories: traditional APIs (41.5%) and application-specific APIs (58.5%) for particular functions. Similar applications share functionally similar APIs (70.5%).*

B. Compromised Trust Boundaries

The analyzed resource injection vulnerabilities demonstrate systematic breaches across six critical trust boundaries within Kubernetes environments, as shown in Figure 3a.

1) Data Plane (Container) to Host: Container escape through resource injection in the data plane, bypassing container isolation mechanisms. For example, CVE-2024-3154 injects systemd properties (such as `ExecStart`) through malicious Pod annotations, circumventing container isolation.

2) Data Plane to Control Plane: Gaining access to or modifying control plane content stored in the data plane through resource injection of the data plane, such as the vulnerabilities in Figure 2.

3) Inter-Control Plane Components: Exploiting malicious content injected by one component to probe or manipulate other components, such as the SSRF attack launched by CVE-2022-29164.

4) Multi-Tenant Isolation: Breaking namespace isolation through malicious resource references, such as CVE-2024-43803 discussed in Section V-A2b.

5) External to Internal Cluster: Attackers exploit resource injection to expose internal cluster components to external access. For example, CVE-2021-32783 involves creating malicious `ExternalName` services linked to Ingress routes, enabling unauthorized external access to internal resources.

6) Control Plane to Data Plane: Compromised controllers distribute malicious workloads and configurations to the data plane. For example, CVE-2022-21701 demonstrates this attack vector: attackers create malicious *Gateway* objects, which the *Istiod* controller processes and uses to deploy associated resources such as Pods, thereby propagating malicious workloads across the data plane.

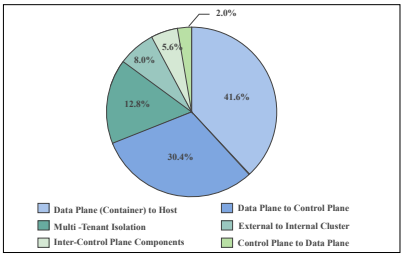
Finding 10: *Resource injection vulnerabilities compromise six distinct trust boundaries, with the majority of attacks targeting data plane to host isolation (41.6%) and data plane to control plane (30.4%).*

C. Impacts

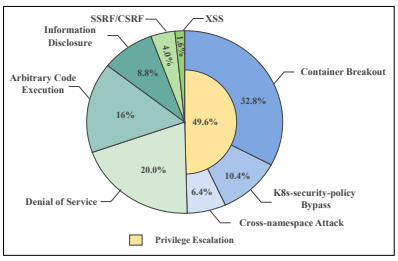
93.6% of vulnerabilities are medium severity or above based on their CVSS v3.0 score. To provide deeper insight into security implications, we conducted a comprehensive impact analysis by categorizing vulnerabilities based on their ultimate security consequences rather than intermediate effects. For instance, vulnerabilities that initially manifest as information disclosure of administrator tokens are classified according to their final exploitable impact, such as privilege escalation. The distribution of these ultimate vulnerability impacts is presented in Figure 3b.

Privilege escalation is the most common vulnerability impact, accounting for 49.6% of cases and encompassing container breakout, k8s-security-policy bypass, and cross-namespace attacks. Container breakout is the primary attack vector (32.8%) leading to such escalation. Denial of service attacks comprise 20.0% of cases, while information disclosure accounts for 8.8%. The remaining vulnerabilities include arbitrary code execution (16.0%) and web-based attacks such as XSS and SSRF/CSRF (5.6% combined).

Finding 11: *93.6% of vulnerabilities are medium severity or above. 49.6% vulnerabilities cause privilege escalation, with container breakout being the primary attack vector (32.8%) leading to such escalation.*



(a) Trust Boundary



(b) Impact

Fig. 3. Distribution of Trust Boundaries and Impacts

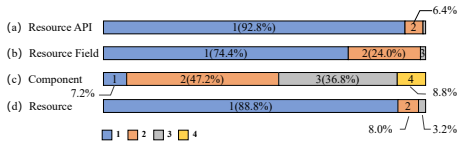


Fig. 4. Distribution of Exploitation Conditions

VII. Exploitation Conditions(RQ4)

We study the input conditions required to exploit a resource injection vulnerability, including the characteristics of malicious content in vulnerable fields and trigger complexity.

A. Malicious Content Characteristics

Our analysis reveals two distinct patterns in how malicious values manifest in Kubernetes environments.

1) Static Configuration Injection (98/125, 78.4%): This pattern involves direct specification of malicious values in configuration fields, where attackers explicitly define harmful parameters that are used as-is without transformation.

2) Dynamic Parameter-based Generation (27/125, 21.6%): This pattern involves seemingly benign user inputs that transform into malicious values through runtime processing. Such vulnerabilities manifest primarily in template rendering, command construction, and expression evaluation contexts, where exploitation typically involves breaking out of intended processing contexts. A notable example is CVE-2021-37914, which has been discussed in Section V-B1.

Finding 12: *78.4% exploits involve static configuration values and 21.6% require runtime transformation.*

B. Exploitation Complexity

To assess the complexity of resource injection vulnerability exploitation, we analyzed four key metrics: (1) the minimum number of required resource API calls, (2) the minimum number of resource fields to be injected, (3) the minimum number of resources involved, and (4) the minimum number of components to be activated. Figure 4 shows the distribution of these conditions.

The analysis reveals a distinctive asymmetric complexity pattern. Three metrics demonstrate remarkably simple trigger requirements: 92.8% of vulnerabilities require only a single resource API call, 74.4% need injection of just one resource field, and 88.8% involve only one resource type. This suggests that the initial activation barrier for resource injection vulnerabilities is exceptionally low.

However, the component metric presents a stark contrast to this simplicity. Components are defined as functional units (like controllers) that interact externally through communication mechanisms (like sockets and events). Only 7.2% of vulnerabilities remain confined within a single component, while the majority require cross-component propagation: 47.2% involve two components, 36.8% span three components, and 8.8% require four or more components. This distribution indicates that successful exploitation necessitates malicious data traversal across multiple components before reaching privileged operations.

Finding 13: *Over 70% of exploitations require only a single API call, field, and resource, yet 90% involve malicious content across 2-4 application components.*

VIII. Fixes (RQ5)

Fix Locations. Resource injection vulnerabilities are fixed at three primary locations: resource controllers (77.6%), admission controllers (11.2%), and API server (11.2%).

Fix Complexity. The fix of these vulnerabilities requires significant development effort. On average, each vulnerability necessitates 2.3 patches, affects 8.2 files, and involves 399.8 lines of code changes. The average fix time is 44.09 days (median: 31 days), with 72% employing coordinated disclosure with pre-release fixes. Notably, 32% require multiple fix attempts, though 87.9% of re-fixes complete within one week of the initial patch. Field location significantly impacts remediation timelines: metadata-annotation fields average 10.42 days versus 17.33 days for spec fields. This stems from architectural heterogeneity, backward compatibility constraints, cross-component dependencies, and hierarchical fix requirements within Kubernetes systems.

Fix Strategies. The fixing approaches fall into three categories: (1) **Access Control Restriction** (11/125, 8.8%) – limiting user permissions to specific fields or privileged operations by implementing role-based access controls or field-level restrictions to prevent unauthorized access; (2) **Input Validation and Content Filtering** (109/125, 87.2%) – implementing comprehensive security measures including resource value verification, content sanitization, and secure API integration with built-in validation mechanisms to prevent malicious input from reaching privileged operations; and (3) **Field Deprecation**

(5/125, 4.0%) – disabling or removing vulnerable fields that are susceptible to injection attacks.

Finding 14: *Fixes are complex and implemented across three locations with three primary fix strategies.*

IX. Lessons Learned

In this section, we discuss lessons learned for existing approaches, and opportunities for new research in combating resource injection vulnerabilities. As the first study on resource injection vulnerabilities, we believe that our findings can help improve the security of Kubernetes ecosystems.

A. Lessons for Detecting Vulnerabilies

Through our study, we found that existing static and dynamic tools for Kubernetes face significant limitations when detecting resource injection vulnerabilities. Representative tools like Go-flow-levee [45] and Kubefuzzer [46], specifically designed for Kubernetes, suffer from issues such as inability to handle cross-component analysis and use of random mutation strategies without targeted fuzzing seeds for resource injection scenarios. Therefore, we explore how our findings can be leveraged to enhance current tools.

1) Static taint analysis: Finding 5 shows that the majority of vulnerabilities are caused by missing validation, which suggests that static taint analysis can be used to detect this category of vulnerabilities. The key challenge lies in identifying sources(vulnerable fields), sinks(privileged operations), and sanitizers(input validation), as well as modeling cross-component data flow in Kubernetes.

a) Location and Function-Guided Source Identification: Finding 1 inspires focusing vulnerable field identification efforts on the *spec* and *metadata* sections. Finding 2 suggests using similarity characteristics within and across applications to systematically discover additional vulnerable fields from known sources. This also enables systematically checking functionally equivalent fields against known vulnerabilities for proactive detection. The main challenge is identifying whether fields or applications share similar functionality. Finding 3 indicates that vulnerable fields typically exhibit path-type and command execution functionalities, enabling vulnerable field identification based on these two functionalities. The primary challenge lies in accurately determining whether a field possesses these two functionalities. Finding 4 suggests that when resources reference fields from other resources, both the referencing and referenced fields should be considered more likely to be vulnerable. Despite low proportion, prioritize these fields due to cross-namespace impact and severe isolation violations [47].

b) Type-Guided Sink Identification: Finding 9 indicates that privileged operations comprise 6 types, categorized into traditional APIs and application-specific APIs. For traditional APIs, we can systematically identify sink points by matching APIs that exhibit these 6 types. For application-specific APIs, Finding 9 shows that functionally similar applications share functionally similar APIs, which inspires a similarity-based approach: when analyzing a new application, we first identify previously studied applications with comparable functionality, then leverage their known vulnerable APIs to discover APIs with similar functionality in the new application. The primary challenge lies in determining whether two applications and their APIs have similar functionality.

c) Fix-Guided Sanitizer Identification: Finding 14 revealed that sanitizers exist in three locations: resource controllers, admission controllers, and API servers. The challenge lies in systematically identifying whether sanitizers are present for specific fields. Large language models could automate this detection, using our discovered three fix strategy types (access control restriction, input validation, and field deprecation) as few-shot examples.

d) Component-Guided Data Flow Modeling: Finding 13 indicates that data flow propagates between multiple components. This finding identifies that we can model inter-component communication to trace data flow propagation. By modeling these communication mechanisms, we can explicitly map how variables from one component flow to specific variables in another component, thereby establishing clear data flow relationships across component boundaries.

2) Fuzzing: Fuzzing represents an alternative approach for detecting resource injection vulnerabilities by systematically mutating field values. Our study also provides insights that can improve fuzzing effectiveness for resource injection detection.

a) Exploitation-Guided Fuzzing: Finding 13 shows that resource injection vulnerabilities exhibit simple exploitation conditions (>70% require single API call, field and resource). Therefore, during fuzzing, we can mutate only one field of one resource at a time and use only one API to send this mutated value to the controller.

b) Content-Guided Fuzzing: Finding 12 demonstrates that most fields are static. Finding 3 and Finding 4 suggest that their values can be paths, commands, and resource references. These findings inspire fuzzing approaches that use vulnerable field values as seeds for mutation-based testing. The static nature enables direct injection without complex computation, allowing fuzzers to efficiently generate test cases by mutating values according to path, command, and resource reference formats and directly injecting them into corresponding resource fields.

c) Validation-Guided Fuzzing: Finding 8 reveals flaws in existing validation mechanisms due to overlooked content characteristics, suggesting that directed greybox fuzzing [48] can effectively test validation logic, guided by insights from our collected patches.

B. Lessons for Controller Developers

1) Rebuild threat models: Finding 6 demonstrates that trust boundaries undergo significant shifts when organizations migrate applications to cloud environments or expose specific functionalities to external entities. These shifts expose a critical limitation of Kubernetes' RBAC: while it controls resource-level access, it cannot restrict specific fields within resources [49]. Even with proper RBAC configuration,

attackers with legitimate resource creation permissions can inject malicious content into fields that controllers process with elevated privileges, effectively bypassing RBAC restrictions. Developers must conduct threat modeling to identify whether privileged operations can now be triggered through user-controlled input, and implement field-level validation to prevent privilege escalation through controller manipulation.

2) Implement independent validation: Finding 7 indicates that developers should implement their own access control mechanisms against Kubernetes security policies during development to ensure compliance. When referencing other resources, developers should perform final validation checks themselves rather than relying solely on the security measures implemented by other controller developers.

C. Lessons for Cluster Administrators

1) Monitoring: Finding 10 highlights the need for specialized monitoring tools to detect trust boundary compromises. Deploying cloud-based Endpoint Detection and Response (EDR) technology [50]–[53] tailored for Kubernetes environments is essential. Such EDRs should monitor cross-boundary data flows, including container-to-host access behaviors and abnormal data plane calls to critical Kubernetes APIs. They should also track resource injection in real time to provide comprehensive protection against trust boundary compromises.

2) Component isolation and access control: Based on Finding 10, administrators should avoid deploying control plane and data plane components in the same pod to prevent shared resources (like files and network) access that could compromise the trust boundary. Additionally, following the principle of least privilege, administrators should implement fine-grained access control to ensure components access only necessary APIs and resources. Network policies should prevent control plane components from being exposed to public networks, protecting against external-to-internal breaches.

X. Related Work

A. Kubernetes Ecosystem Security Analysis

Several studies have examined Kubernetes security challenges. MITRE [54] and Microsoft [55] outline attack tactics, while empirical research has analyzed CVEs [56], cloud security challenges [57], and platform vulnerabilities [58]. Attack scenarios including privilege escalation [59], cross-container attacks [60], co-residency attacks [61], and cross-namespace reference vulnerabilities [47] have been investigated. Infrastructure reliability studies have examined operator bugs [8], failure analysis [62], and runtime system issues [9]. However, none of them target resource injection vulnerabilities.

Misconfigurations in Kubernetes can lead to severe security vulnerabilities, making misconfiguration prevention a key focus of security research. Extensive research has addressed this challenge through empirical studies [14], [63], security best practices [13], [64], defect analysis [15], network misconfiguration detection [65], permission-related vulnerability analysis [11], [66], [67], and security hardening efforts such as attack surface reduction [49] and secret protection [68]. However, these studies primarily focus on configuration content errors rather than controller implementation flaws. Resource injection vulnerabilities differ fundamentally by exploiting inadequate input validation in controller code that processes untrusted configuration inputs, requiring analysis of both configuration semantics and controller implementation rather than static configuration analysis alone. Consequently, existing approaches do not specifically target this distinct class of vulnerabilities in the Kubernetes ecosystem.

B. Kubernetes Ecosystem Bug Detection Tools

Most existing Kubernetes bug detection tools focus on misconfiguration issues through pre-deployment checks using rule-based [16]–[20], graph-based [21]–[23], and LLM-based approaches [24]–[26]. For controller verification, Sieve [4] detects reliability issues by perturbing cluster states, finding 46 critical errors. Acto [5] validates operator functionality through automated testing, uncovering 56 bugs. Kivi [6] uses model checking for controller configurations. Anvil [7] formally verifies controller liveness properties. EPScan [11] detects excessive permissions in third-party applications using pod-oriented program analysis. For API testing, KubeAPI-Inspector [69] discovers hidden vulnerable APIs, KubeFuzz [70] fuzzes admission controller chains, and KubeFuzzer [46] uses NLP to guide effective request generation. However, none of these tools specifically target resource injection vulnerabilities.

XI. Conclusion

Kubernetes' resource-based architecture introduces resource injection vulnerabilities when controllers perform privileged operations using user-controllable fields without proper validation, leading to severe consequences. We conduct the first comprehensive study on 125 resource injection vulnerabilities from Kubernetes ecosystems. From our study, we obtain many interesting findings and lessons. We believe that our study can be beneficial for cloud-native security and software engineering researchers from many aspects, *e.g.*, vulnerability detection, automated testing, and secure controller design.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103901, Chinese National Natural Science Foundation (Grants #62202462, #62302500, #62032010, #62202452).

References

[1] Cloud Native Computing Foundation, "The cncf cloud native glossary project," 2023. [Online]. Available: https://github.com/cncf/glossary

[2] Kubernetes, "Kubernetes documentation," accessed: 2025-04-28. [Online]. Available: https://kubernetes.io/

[3] Cloud Native Computing Foundation. (2022) Cncf annual survey 2021. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2021/

[4] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 143–159.

[5] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3600006.3613161

[6] B. Liu, G. Lim, R. Beckett, and P. B. Godfrey, "Kivi: Verification for cluster management," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 509–527.

[7] X. Sun, W. Ma, J. T. Gu, and Z. Ma, "Anvil: Verifying liveness of cluster management controllers," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[8] Q. Xu, Y. Gao, and J. Wei, "An empirical study on kubernetes operator bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680396

[9] J. Yu, X. Xie, C. Zhang, and S. Chen, "Bugs in pods: Understanding bugs in container runtime systems," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680366

[10] 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 (CCS)*, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3576915.3623121

[11] Y. Zhang, Y. Wang, J. Li, X. Liu, X. Guo, and J. Ma, "Epscan: Automated detection of excessive rbac permissions in kubernetes applications," in *Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 11–25. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2025/223600a011/21B7Q3t3dF6

[12] 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.

[13] 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*, 2021, pp. 1689–1690.

[14] S. I. Shamim, H. Hu, and A. Rahman, "On prescription or off prescription? an empirical study of community-prescribed security configurations for kubernetes," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 707–707.

[15] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported'security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021, pp. 9–12.

[16] StackRox, "Kubelinter," https://kubelinter.io, 2024.

[17] Fairwinds, "Polaris," https://www.fairwinds.com/polaris, 2024.

[18] P. Cloud, "Checkov," https://www.checkov.io, 2024.

[19] KIKS, "Checkmarkx," https://kics.io, 2024.

[20] E. Russell and K. Dev, "Centralized defense: Logging and mitigation of kubernetes misconfigurations with open source tools," *arXiv preprint arXiv:2408.03714*, 2024.

[21] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 420–431.

[22] A. Blaise and F. Rebecchi, "Stay at the helm: secure kubernetes deployments via graph generation and attack reconstruction," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 59–69.

[23] G. Dell'Immagine, J. Soldani, and A. Brogi, "Kubehound: Detecting microservices' security smells in kubernetes deployments," *Future Internet*, vol. 15, no. 7, p. 228, 2023.

[24] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation," *arXiv preprint arXiv:2405.19954*, 2024.

[25] F. Minna, F. Massacci, and K. Tuma, "Analyzing and mitigating (with llms) the security misconfigurations of helm charts from artifact hub," *arXiv preprint arXiv:2403.09537*, 2024.

[26] G. Lanciano, M. Stein, V. Hilt, T. Cucinotta *et al.*, "Analyzing declarative deployment code with large language models," *CLOSER*, vol. 2023, pp. 289–296, 2023.

[27] "K8s resource injection vulnerabilities dataset and analysis results," 2025, gitHub repository. [Online]. Available: https://github.com/b0b0haha/K8s-Resource-InjectionVul-DataSet

[28] F5, Inc. (2025) Nginx ingress controller. [Online]. Available: https://docs.nginx.com/nginx-ingress-controller/

[29] Cloud Native Computing Foundation, "Cloud native computing foundation," https://www.cncf.io/, 2025, accessed: 2025-05-30.

[30] MITRE Corporation. (2025) Common vulnerabilities and exposures (cve). [Online]. Available: https://cve.mitre.org/

[31] National Institute of Standards and Technology. (2025) National vulnerability database. [Online]. Available: https://nvd.nist.gov/

[32] CVE Details. (2025) Cve security vulnerability database. [Online]. Available: https://www.cvedetails.com/

[33] Google. (2025) Open source vulnerabilities (osv). [Online]. Available: https://osv.dev/

[34] T. Dunlap, E. Lin, W. Enck, and B. Reaves, "Vfcfinder: Pairing security advisories and patches," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1128–1142.

[35] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.

[36] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[37] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 339–351.

[38] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 539–550.

[39] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-Anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM symposium on cloud computing*, 2014, pp. 1–14.

[40] A. Rahman and C. Parnin, "Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3536–3553, 2023.

[41] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu *et al.*, "Understanding transaction bugs in database systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[42] Wiz Research Team, "Ingress-nginx kubernetes vulnerabilities," accessed: 2025-03-24. [Online]. Available: https://www.wiz.io/blog/ingress-nginx-kubernetes-vulnerabilities

[43] Y. Hu, W. Wang, C. Hunger, R. Wood, S. Khurshid, and M. Tiwari, "Achyb: A hybrid analysis approach to detect kernel access control vulnerabilities," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 316–327.

[44] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "{PeX}: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1205–1220.

[45] Google, "Go flow levee," GitHub repository, 2021, static analysis tool for data flow security. [Online]. Available: https://github.com/google/go-flow-levee

[46] T. Zheng, R. Tang, X. Chen, and C. Shen, "Kubefuzzer: Automating restful api vulnerability detection in kubernetes." *Computers, Materials & Continua*, vol. 81, no. 1, 2024.

[47] A. Chen, Z. Jin, Z. Guo, and Y. Chen, "Breaking the bulkhead: Demystifying cross-namespace reference vulnerabilities in kubernetes operators," *arXiv preprint arXiv:2507.03387*, 2025.

[48] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[49] C. Cesarano and R. Natella, "Kubefence: Security hardening of the kubernetes attack surface," *arXiv preprint arXiv:2504.11126*, 2025.

[50] CrowdStrike, Inc. (2024) What is endpoint detection and response (edr)? Accessed: 2025-05-29. [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/

[51] Sysdig, "Endpoint detection and response (edr) for containers and kubernetes," 2024, accessed: 2024. [Online]. Available: https://sysdig.com/blog/sysdig-edr-container-kubernetes/

[52] ——, *Sysdig Secure Documentation*, 2024, comprehensive security platform for cloud-native environments. [Online]. Available: https://docs.sysdig.com/en/docs/sysdig-secure/

[53] Microsoft, "Microsoft defender for kubernetes - the benefits and features," *Microsoft Learn*, 2024, real-time threat protection for AKS environments. [Online]. Available: https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-kubernetes-introduction

[54] MITRE, "Matrix-enterprise-containers," accessed: 2025-09-29. [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/

[55] Microsoft, "Threat matrix for kubernetes," accessed: 2025-09-29. [Online]. Available: https://microsoft.github.io/Threat-Matrix-for-Kubernetes/

[56] Nordell, Fred, "A systematic evaluation of CVEs and mitigation strategies for a Kubernetes stack," 2022, Student Paper.

[57] Y. Yang, W. Shen, B. Ruan, W. Liu, and K. Ren, "Security challenges in the container cloud," in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2021, pp. 137–145.

[58] 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.

[59] N. Pecka, L. Ben Othmane, and A. Valani, "Privilege escalation attack scenarios on the devops pipeline within a kubernetes environment," in *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering*, 2022, pp. 45–49.

[60] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, "Cross container attacks: The bewildered {eBPF} on clouds," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5971–5988.

[61] 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.

[62] M. Barletta, M. Cinque, C. Di Martino, Z. T. Kalbarczyk, and R. K. Iyer, "Mutiny! how does kubernetes fail, and what can we do about it?" in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 1–14.

[63] A. Rahman, S. I. Shamim, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–32, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3579639

[64] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *2020 IEEE Secure Development (SecDev)*, pp. 58–64, 2020.

[65] J. Bufalino, J. L. Martin-Navarro, M. Di Francesco, and T. Aura, "Inside job: Defending kubernetes clusters against network misconfigurations," *Proceedings of the ACM on Networking*, vol. 3, no. CoNEXT3, pp. 1–25, 2025.

[66] 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.

[67] A. Sissodiya, E. Chiquito, U. Bodin, and J. Kristiansson, "Formal verification for preventing misconfigured access policies in kubernetes clusters," *IEEE Access*, 2025.

[68] M. Rostamipoor, A. Sadeghi, and M. Polychronakis, " KubeKeeper: Protecting Kubernetes Secrets Against Excessive Permissions ," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2025, pp. 322–338. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/EuroSP63326.2025.00027

[69] "KubeAPI-Inspector: A Dynamic Kubernetes API Detection Framework," https://github.com/yeahx/KubeAPI-Inspector, 2025.

[70] "Kubefuzz: Generative and Mutative Fuzzer for Kubernetes Admission Controllers," https://github.com/avolens/kubefuzz, 2023.