KubeKeeper: Protecting Kubernetes Secrets Against Excessive Permissions

Maryam Rostamipoor
Stony Brook University
mrostamipoor@cs.stonybrook.edu

Aliakbar Sadeghi
Stony Brook University
alisadeghi@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—Kubernetes has become the dominant platform for managing containerized applications, but its native Secrets management mechanisms introduce security vulnerabilities, especially in environments where third-party applications may have excessive permissions. In this paper, we present KubeKeeper, a comprehensive solution for protecting Kubernetes Secrets against leakage due to excessive permissions. KubeKeeper automatically encrypts Secrets and ensures that only explicitly authorized Pods can access their decrypted form. This is achieved by integrating with Kubernetes' admission control framework to transparently enforce access policies, without requiring changes to application code and with minimal integration effort into existing cluster infrastructure. We evaluated KubeKeeper on a diverse set of 498 Kubernetes applications and demonstrate that it successfully protects Secrets against all identified excessive permissions, without introducing performance degradation during execution or any significant overhead during Pod creation and deployment.**

***Index Terms*—cloud computing, Kubernetes, excessive permissions, Secrets management**

1. Introduction

Kubernetes is a widely adopted platform for automating the deployment, scaling, and management of containerized applications [51]. Practitioners adopt Kubernetes due to its ability to reduce the repetitive manual processes traditionally involved with container deployment and management [82]. As one of the most popular open-source container orchestration tools, Kubernetes is widely used by major organizations and governments [6], [48].

Despite its strengths, Kubernetes introduces significant security challenges—particularly in the management of sensitive data such as authentication tokens, cryptographic keys, and user credentials. To handle such sensitive data, Kubernetes introduces the concept of *Secrets* [32], a special type of object that helps developers to ensure that confidential data is not included in application code. However, the native Secrets management mechanisms in Kubernetes have several limitations, especially in deployments that include third-party applications [56], [69], [76], [92].

By default, Secrets are stored in `etcd` (Kubernetes' key–value store) in plaintext form. Although encryption at rest is supported, it is not enabled out of the box and lacks critical features, such as key rotation and audit logging. Moreover, Kubernetes' Role-Based Access Control (RBAC) [54] provides coarse-grained controls. A user or Service Account can be granted `get` or `list` permissions on Secrets, but these permissions give access to the full decrypted values. Kubernetes lacks a zero-trust model in which users can store Secrets without having direct access to their plaintext content [56].

Another major issue lies in how Kubernetes ties Secrets to workload deployment. Kubernetes does not prevent Pods from mounting any Secret in the same Namespace. This means that if a user or Service Account has permission to create Pods—or higher-level resources like Deployments or DaemonSets—it can indirectly access Secrets by configuring the workload to mount them, even if it lacks direct read access via RBAC [4], [9], [10], [14]. This gap can be exploited by both compromised applications and overprivileged internal users.

These risks are amplified by third-party applications, which are often deployed with insecure default configurations. These defaults typically grant excessive permissions—such as cluster-wide Secrets access or Pod creation capabilities [92]. These default settings can bypass best practices for access control and unintentionally expose Secrets to unauthorized workloads. Yang et al. [92] analyzed the security of third-party applications and observed that they often receive excessive permissions, which can be exploited to compromise Secrets. A recent vulnerability in the widely deployed `ingress-nginx` Controller (used in over 40% of Kubernetes clusters) further illustrates the severity of this issue. Under certain conditions, an unauthenticated attacker with network access could achieve arbitrary code execution in the Controller's context, leading to the disclosure of any Secrets it could access—by default, all cluster-wide Secrets.

Due to the current limitations in Kubernetes' Secrets management, it is challenging to automatically prevent these excessive permission risks without impacting application functionality. A common recommendation is to delegate Secrets management to an external Key Management Service (KMS) [13], [36], [47], which offers fine-grained access control over encryption key usage. KMS platforms do not fully mitigate the risks associated with excessive permissions. While KMS platforms typically support fine-grained access control, they cannot prevent workloads from mounting Secrets, and security still depends heavily on correct access control configuration—with misconfigurations remaining a common issue.

As a step towards mitigating the Secrets management limitations of Kubernetes, in this paper we propose *KubeKeeper*, a novel approach that provides more fine-grained control over Secrets to prevent unauthorized access due to excessive permissions or insecure default configurations.

KubeKeeper enhances Kubernetes' Secrets management by encrypting Secrets with unique keys, and tightly controlling their decryption. While encrypted Secrets remain accessible under existing access control mechanisms (i.e., RBAC) and within the defined Namespace scope, their decryption is tightly controlled.

Each Secret is created with a specific purpose and designated resources in mind. Therefore, during application deployment, it is clear which resources are needed to use a defined Secret. Additionally, should new resources require access to these Secrets, they can be accommodated by updating the list of authorized resources. KubeKeeper uses an automated approach to identify all authorized resources for each Secret during application deployment. Decryption keys are then granted exclusively to the specific Pods that have been explicitly authorized during a Secret's deployment or update process. This ensures that unauthorized resources, whether through *direct* or *indirect* access, can only retrieve the encrypted version of Secrets.

KubeKeeper is designed to leverage the Admission Control mechanism of the Kubernetes API server. The key components of KubeKeeper are implemented as dynamic Admission Webhooks, which are responsible for encrypting Secrets and distributing encryption keys to authorized resources. A primary goal of KubeKeeper is to provide practical protection without imposing a burden on developers. To that end, it *does not require any changes to application source code*, and only minimal high-level configuration adjustments.

To evaluate KubeKeeper, we analyzed a diverse set of 498 applications (some of which have been used in previous studies [78], [92]), which we extended for our analysis. As part of our evaluation, we developed a tool that scans the YAML configuration files of applications to automatically identify and highlight excessive permissions, offering a more efficient and reliable analysis compared to existing tools [49], [92], which require manual installation on live clusters.

Our results show that 202 out of the 498 applications (41%) have excessive permissions that can lead to unauthorized access to their Secrets. Notably, 1,866 permissions across 84% of these vulnerable applications provide direct access to Secrets, while 3,068 permissions in 79% of them provide indirect access to Secrets by controlling resource deployment. KubeKeeper protects Secrets against all identified instances of excessive permissions. Our performance evaluation results indicate that KubeKeeper introduces no application runtime overhead, and no significant overhead during Pod creation and deployment.

In summary, we make the following main contributions:

- We designed and implemented KubeKeeper, a Kubernetes Secrets protection approach that transparently encrypts Secrets and tightly controls their decryption, to prevent unauthorized access due to excessive permissions or insecure default configurations. KubeKeeper does not require any changes to the source code of applications.
- We developed a tool that builds and scans Kubernetes YAML configuration files to automatically identify excessive permissions that can lead to direct or indirect unauthorized access to Secrets.
- We experimentally evaluated KubeKeeper on a diverse set of 498 Kubernetes applications and demonstrate its successful protection of Secrets against *all* identified excessive permissions, without causing any performance degradation during execution or significant overhead during creation and deployment.

Our implementation, along with the full dataset of collected applications used in our experimental evaluation, is publicly available as an open-source project at https://github.com/mrostamipoor/KubeKeeper.

2. Background

2.1. Kubernetes Architecture

Kubernetes (K8s) [51] serves as a platform that manages the orchestration, scaling, and administration of containers. It enables efficient service and application management through declarative configurations and automation, making it essential in modern cloud computing environments. A Kubernetes cluster [1] consists of a set of Nodes that correspond to either virtual or physical machines. Kubernetes runs workloads by placing containers into Pods that operate on these Nodes. Master Nodes (forming the Control Plane) and Worker Nodes are the main types of Nodes within a Kubernetes cluster.

A Master Node consists of the following components: API Server, Scheduler, Controller, and `etcd`. The API Server is the core of the Kubernetes Control Plane, exposing an API for cluster management and security [2]. Containers programmatically interact with Kubernetes resources via the exposed RESTful API, using a Service Account for authentication. This is a special type of account that provides an identity for containers running in a Pod to interact with the Kubernetes API Server [18]. The Scheduler [26] assigns newly created Pods to available Worker Nodes, while the Controller [19] monitors and adjusts the state of the cluster. A highly available key–value store is provided by `etcd` [44], which keeps all configuration information for the whole cluster.

Worker Nodes contain components such as Kubelet, Kube-proxy, and Pods. Kubelet [24] manages the lifecycle of Pods and containers, translating Kubernetes commands into Docker commands when applicable. Kube-proxy [22] maintains network rules for the internal and external network communication to and from the Pods. The smallest deployable unit of computing in Kubernetes is the Pod [50], which typically consists of one or more containers. Pods have a defined lifecycle, meaning that if a Pod fails due to a Node fault, Kubernetes treats this as final condition, requiring a new Pod to be manually created. To automate this process, Kubernetes provides workload resources [15] that automatically manage Pod lifecycles and maintain the desired cluster state. These include Deployment, ReplicaSet, StatefulSet, DaemonSet, Job, and CronJob [51], each designed to handle specific scenarios and requirements.

2.2. Kubernetes API Server

The Kubernetes API Server plays a central role in managing and securing a cluster by processing and validating all API requests. It uses several modules, including *Authentication*, *Authorization*, and *Admission Control*, that enforce

a multi-layered security policy [3], [12]. Each request is authenticated to verify the user or Service Account identity. Upon successful authentication, the request is subjected to authorization checks, typically through Role-Based Access Control (RBAC) [54]. Kubernetes RBAC is a key security control that ensures users and Service Accounts can only access the resources they need according to their Roles.

The RBAC API defines four types of Kubernetes objects: *Role*, *ClusterRole*, *RoleBinding* and *ClusterRoleBinding*. An RBAC Role contains `rules` that set permissions within a specific Namespace, while a ClusterRole, in contrast, is a `non-namespaced` resource that defines permissions on `cluster-scoped` resources. Namespaces provide a mechanism for isolating groups of resources within a single cluster. Finally, Admission Control modules intercept requests and evaluate them to ensure they comply with specific cluster policies before an object is persisted. Admission Control involves two distinct *Mutating* and *Validating* phases. Mutating Controllers can alter objects associated with the requests they process, whereas Validating Controllers only validate a request based on the defined policy. In addition to compiled-in Admission plugins, custom Admission plugins can be developed as extensions and run as Webhooks configured at runtime.

2.3. Kubernetes Secrets

A Kubernetes *Secret* is an object that stores sensitive data, such as passwords, tokens, and cryptographic keys. Instead of embedding sensitive data objects in Pod specifications or container images, Secrets allow developers to securely expose them to applications in Pods through environment variables, or by mounting them as volumes [32].

Kubernetes Secrets often contain security-critical data, such as Service Account tokens, credentials, and API keys. If leaked, these Secrets can compromise not only workloads within the cluster but also external services. To reduce risk, a Secret is only sent to a Node if a scheduled Pod explicitly requests it, and each Pod can access only its own mounted Secrets. To prevent Secrets from being written to persistent storage when mounting Secrets into Pods, the Kubelet stores a copy of the data in `tmpfs`, which is a temporary file storage facility [32].

3. Secret Management Limitations

Kubernetes' native Secrets management has several limitations that hinder secure, least-privilege access. By default, Secrets are stored unencrypted in `etcd`, and even with optional encryption at rest enabled, Kubernetes does not provide built-in support for key rotation. Permissions like `get` and `list` allow full access to decrypted Secrets via the API server, with no mechanism to enforce encrypted-only access. Even if access is restricted to specific Service Accounts [33], users with Pod creation privileges can simply assign a privileged Service Account to a new workload and bypass these protections.

Kubernetes also lacks mechanisms to control which workloads can mount which Secrets. While RBAC controls who can `create`, `read`, or `delete` resources through the API Server, it does not provide fine-grained control over workload configurations, such as restricting which Secrets a Pod can mount [4], [9], [10], [14].

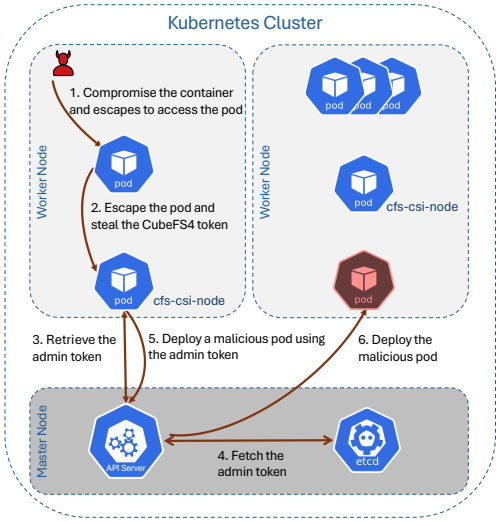

Figure 1: Attackers can control a Worker Node and exploit the CubeFS4 Service Account to further exploit the cluster by stealing the cluster administrator's Secrets and deploying a malicious Pod.

As a result, a significant security concern arises when users or Service Accounts are granted permissions to create Pods within a Namespace, either directly or through higher-level resources like Deployments [32]. While these permissions do not provide direct access to Secrets via the API, they allow users to configure workloads that mount any Secret in the Namespace [7], [32], potentially leading to unauthorized access. This architectural gap means developers and CI/CD pipelines may unintentionally—or maliciously—access sensitive Secrets without direct API permissions. These risks are exacerbated by common misconfigurations. Our analysis, which builds on the dataset from Rahman et al. [78], reviewed 54 Kubernetes clusters and found 88% of them containing one or more Secrets improperly bound to the `default` Namespace instead of specific ones. This allows a Pod within the default Namespace to mount any defined Secrets and gain (potentially unauthorized) access to them.

This challenge is made worse by the complexity of RBAC configuration. In dynamic or large-scale environments, defining fine-grained rules is error-prone and often leads to over-permissioning. This issue becomes even more critical in the context of *third-party applications*, which often come with default settings that grant them more access than necessary, significantly increasing the potential for security breaches. Yang et al. [92] identified numerous instances of third-party applications with excessive critical permissions. The study assumes an attacker who gains control of a Worker Node and attempts to compromise the entire cluster. Given the prevalence of application compromises and container escapes [64], [70], [73], [79], [83], [88]–[91], a malicious actor with access to a containerized application can exploit vulnerabilities in the host OS kernel to escape from the container and compromise the Worker Node. Once the Node is compromised, the attacker can misuse the permissions of third-party applications running on the compromised Node with default settings to bypass the deployed isolation mechanisms and take control of the entire cluster or access Secrets.

TABLE 1: Mapping of Secret-Compromising Permission Types to RBAC Permissions.

Permission Type	Verbs	Resources
Direct Access via Secret Permissions	`get`, `watch`, `list`, `*`	`secrets`, `serviceaccounts/token`
Indirect Access via Secret Manipulation	`patch`, `update`	`secrets`, `serviceaccounts/token`
Indirect Access via Resource Scheduling Control	`create`, `patch`, `update`, `*`	`pods`, `daemonsets`, `deployments`, `statefulsets`, `replicasets`, `cronjobs`, `jobs`
Indirect Access via Node Manipulation	`patch`, `update`, `*`	`nodes`

3.1. Secret Exposure via Excessive Permissions

Different types of excessive permissions can enable a range of attack strategies, each posing a risk of unauthorized access to Secrets. First, an attacker could exploit the excessive permissions of a third-party application on a Worker Node to directly steal cluster administrator privileges. These excessive permissions typically provide direct access to Secrets.

An illustrative example presented by Yang et al. [92], involves CubeFS4 [37], an open-source cloud-native file storage system hosted by the Cloud Native Computing Foundation (CNCF) as an `incubating` project. CubeFS4 includes a critical DaemonSet (`cfs-csi-node`) which deploys a Pod on each Node of a cluster. This critical DaemonSet uses a Service Account named `cfs-csi-service-account`, assigned with the `cfs-csi-cluster-role` via the `cfs-csi-cluster-role-binding`. This role grants it `get` permissions for Secrets resources. Consequently, on a compromised Worker Node, attackers could obtain each Pod's Service Account token (including the DaemonSet's Pod) through the path `/var/run/secrets/kubernetes.io/serviceaccount/`. They could then exploit the excessive permissions of the token to directly obtain the cluster administrator's Secrets, allowing them to escape from the Worker Node and compromise the entire cluster, as illustrated in Figure 1.

Alternatively, the attacker might hijack critical components of the same or another application, which have direct access to Secrets. The attacker must leverage excessive permissions of resources like Daemonsets to force the privileged component to operate on the compromised Node. Again, this indirect method would also enable the attacker to gain administrator permissions. This scenario is similar to the previous example. The only difference is that the attacker exploits excessive permissions of a Service Account token to deploy an authorized resource on the compromised Node, and then uses the token of this authorized resource to perform the further steps mentioned in Figure 1. Based on these strategies, we identify four types of permissions that can lead to direct or indirect access to Secrets. The mapping of these permission types to corresponding RBAC rules is shown in Table 1.

1) Direct Access via Secrets Permissions: Permissions like `get`, `watch`, `list`, or `*` on Secrets allow a Service Account's token to be misused to fetch Secrets through the API server.

2) Indirect Access via Secrets Manipulation: An attacker can modify a Secret's ownership with `patch` or `update` on Secrets or `serviceaccounts/token` to grant attacker-controlled resources access to it.

3) Indirect Access via Resource Scheduling Control: Permissions to `create`, `patch`, `update`, or `*` on workload resources (e.g., Pods, Daemonsets) enable attackers to control where workloads are deployed, and to place resources with access to Secrets on compromised Nodes.

4) Indirect Access via Node Manipulation: Using `update` or `*` verbs on Node resources, attackers can manipulate taints to make all Nodes, except the compromised one, unschedulable, forcing the Kubernetes scheduler to deploy resources with access to Secrets onto the compromised Node [92].

The results of Yang et al. [92] showed that 33.3% of CNCF third-party applications [17], [21], [30] and all applications provided by the top-four cloud vendors [38], [39], [41], [53] were vulnerable to excessive permission attacks [92]. KubeKeeper mitigates these threats by preventing unauthorized access to plaintext sensitive data through both *default configurations* or *excessive permissions*.

4. Threat Model

We consider attackers who aim to gain unauthorized access to Kubernetes Secrets, either by retrieving them through the API server or by accessing them at runtime via deployed workloads. We focus on two main types of attackers: 1) attackers who compromise a containerized application running inside the cluster, and 2) users or Service Accounts that are granted excessive permissions either due to misconfigurations or insecure defaults.

In the first case, the attacker exploits vulnerabilities in a containerized application [70], [73], [79], [90], [91] to gain remote code execution inside a Pod. From there, they may access the Kubernetes API using the Pod's Service Account token, or escape the container using kernel vulnerabilities [64], [88], [89] or misconfigurations [5], [78], [86], [88] to compromise the Worker Node.

In the second case, the attacker may already have access to the API server through a legitimate user account or an over-privileged Service Account, and attempts to access Secrets they were not intended to access. While cloud platforms recommend isolating workloads to specific Nodes to mitigate such risks [16], [35], [68], this isolation can be circumvented. In practice, the misuse of excessive permissions—especially those assigned to third-party applications—can break Node isolation and lead to full cluster compromise and unauthorized access to Secrets [92].

In both cases, we assume that the attacker does not compromise the Kubernetes Control Plane. Permissions that allow privilege escalation (e.g., creating ClusterRoles or modifying RBAC policies) are considered out of scope, particularly when KubeKeeper is implemented using a Webhook Server. Such permissions should never be granted under even the most basic security practices. When

TABLE 2: Comparison between Kubernetes' native RBAC-based access control and KubeKeeper.

Feature	RBAC	KubeKeeper	Remarks
Ease of configuration	×	✓	KubeKeeper establishes a direct link between Secrets and their authorized Pods, eliminating the need for the complex mappings that RBAC requires
Protection of Secrets at rest	×	✓	Unlike Kubernetes' default configuration, KubeKeeper ensures that Secrets are stored encrypted
Protection of Secrets in transit	×	✓	KubeKeeper encrypts Secrets during transmission, offering protection against interception
Delivery of Encrypted Secrets	×	✓	KubeKeeper supports the delivery of Secrets in encrypted form
Fine-grained access control	×	✓	KubeKeeper offers increased protection granularity through per-Pod access controls
Owner-defined access control	×	✓	KubeKeeper allows only the Secret owners to exclusively designate which Pods are authorized
Protection against unauthorized access	✓	✓	Both systems are designed to prevent unauthorized access
Protection against compromised components	×	✓	Even if Kubernetes deployment resources are compromised, KubeKeeper maintains security through data encryption, controlled access, and proactive monitoring
Protection against misconfigurations	×	✓	KubeKeeper reduces the risk of security breaches caused by configuration errors
Low impact on application performance	✓	✓	KubeKeeper does not introduce any performance degradation during execution

KubeKeeper is implemented using a Webhook Server, we consider it part of the trusted computing base. The Webhook Server is deployed in a dedicated Namespace and isolated Node, and Secret-related permissions alone are not sufficient to compromise it.

5. Design

KubeKeeper enhances Kubernetes Secrets management by encrypting Secrets with unique keys and tightly controlling their decryption. While encrypted Secrets remain accessible under existing access control mechanisms, their decrypted forms are restricted. When a Secret is created, it is intended for specific purposes and designated resources. Therefore, during application deployment, it is clear which resources will require access to a defined Secret. Additionally, when new resources require access to these Secrets, they can be accommodated by updating the list of authorized resources. Consequently, decryption keys are made available exclusively to specific Pods that receive explicit authorization during a Secret's deployment or update process. This maintains open access to encrypted Secrets while ensuring strict control over their decryption.

We designed KubeKeeper according to the following key requirements:

- Prevent excessive permissions from enabling unauthorized (direct or indirect) access to Secrets.
- Provide controlled access to encrypted Secrets, restricting decryption only to authorized workloads.
- Enhance and simplify the management of Secrets.
- Maintain compatibility with Kubernetes and avoid any changes to its source code.
- Avoid the need to modify the source code of applications consuming Secrets.

5.1. Advantages Over RBAC

KubeKeeper offers several advantages over Kubernetes' native Role-Based Access Control (RBAC) in terms of both usability and increased protection against additional threats. Kubernetes relies primarily on RBAC to manage access to secrets, requiring manual configuration of Roles, ClusterRoles, and their bindings to Kubernetes resources. This setup becomes particularly challenging and error-prone, as we demonstrate in Section 7.2.3, especially in dynamic environments where roles and responsibilities frequently change. In contrast, KubeKeeper establishes automatically a direct link between Secrets and their authorized Pods, eliminating the need for the complex mappings required by RBAC.

RBAC policies determine who can create, read, or delete resources via the API Server, but they lack granular control over which Secrets a pod can mount [4], [9], [10], [14]. KubeKeeper addresses this limitation by providing more fine-grained access controls specific to each Pod. It also implements an access control mechanism for Secrets owners, allowing only the owner who deploys a Secret to authorize access to the designated Pods.

In addition to fine-grained access control, KubeKeeper introduces an extra layer of security by keeping Secrets always encrypted and actively monitoring their use. This ensures that even if RBAC controls are bypassed and Secrets in `etcd` can be accessed, they will remain secure and unusable, since attackers cannot access the corresponding cryptographic keys. This approach also protects Secrets during transmission, by keeping them encrypted until the moment they are actually used.

Additional security benefits of KubeKeeper include guarding against excessive permissions that could allow attackers to run a separate Pod with access to a Secret within a compromised Node, i.e., *indirect* unauthorized access. From a performance perspective, similarly to RBAC, KubeKeeper does not introduce any measurable overhead during execution. Table 2 provides a detailed comparative analysis between Kubernetes' native RBAC access control and KubeKeeper.

5.2. Overall Architecture

To integrate with Kubernetes, KubeKeeper enhances Admission Control modules to mutate or validate requests related to Secret creation, Secret mounting, and critical resource updates. Figure 2 provides an overview of the proposed architecture. Since KubeKeeper must encrypt Secrets at creation time, receive the list of authorized resources for access, and verify the authority of workloads attempting to access those Secrets, our design leverages key Admission Plugins that intercept requests for *Secret* creation, *Pod* creation, and *Node* updates. Importantly, KubeKeeper integrates with the cluster without altering the standard process of deploying Secrets or workloads. Instead, it introduces additional Plugins during the Admission phase of the API server to enforce encryption and access control.

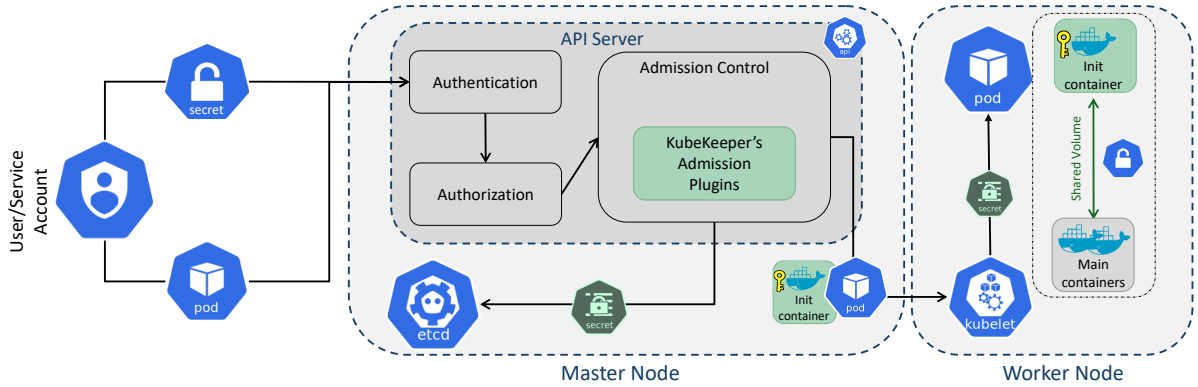

Figure 2: Integration of KubeKeeper within a Kubernetes cluster. The architecture highlights key components and interactions, including KubeKeeper's Admission Plugins for dynamic Secret management.

Moreover, during Pod runtime, KubeKeeper requires an injected `init` container to perform the decryption tasks.

A key goal of KubeKeeper is to provide practical protection without burdening developers, by not requiring any changes to the source code of applications—only minimal, high-level configuration changes are needed. We also use i) the built-in feature of `annotations`, which are arbitrary, non-identifying metadata attached to objects [11], to determine the authorized resources for each Secret; and ii) `labels`, which are key–value pairs attached to Kubernetes objects [27], to inform KubeKeeper's Admission Plugins when to mutate resources. This makes the adoption of KubeKeeper straightforward in existing applications. At the same time, full compatibility with legacy, non-KubeKeeper-protected applications is maintained by not altering Kubernetes' native Secrets management. When an application uses Secrets without the specific label to trigger KubeKeeper's Webhook APIs, they remain unaltered and are protected solely by the native Kubernetes policies.

5.3. Secrets Annotation and Encryption

Secrets are typically created using manifests or Helm charts, both of which are widely adopted for managing Kubernetes resources [45], with sensitive data stored in a Base64-encoded format. It is also possible to use a CLI command (manual approach), which is commonly used when a Secret does not require complex configurations or long-term management. In our approach, we assume that Secrets are created using a manifest file or through Helm charts, ensuring that our solution is compatible with both methods commonly used in production environments [25].

Listing 1: YAML manifest for the deployment of a Secret protected using KubeKeeper.

```yaml
kind: Secret
metadata:
  name: secret-data
  namespace: production
  annotations:
   secret-ownerships:
       "example-pod:Pod:production"
  labels:
    protected-secret: "true"
type: Opaque
data:
sensitivedata: bmV3dGVzdAo=
```

Listing 1 shows a sample YAML configuration file for deploying a Secret. To ensure Secrets are protected by KubeKeeper, Secrets manifests should contain the label `protected-secret` with the value set to `true`, whether using a static YAML file or a Helm chart. We leverage Kubernetes' built-in `labels` [27] attribute within the `metadata` section to annotate these Secrets. This label signals the KubeKeeper Admission Plugin, `SecretMutation`, to encrypt the Secrets. Figure 3 illustrates the workflow followed by KubeKeeper to protect Secrets during deployment.

After a request to deploy or update a Secret is received by the API server and passes authentication and authorization, it is delivered to the `SecretMutation` Admission Plugin. The plugin inspects the request to determine whether the Secret should be protected by KubeKeeper, based on the presence of the `protected-secret` label. If protection is required, the plugin extracts the Secret's Name and Namespace attributes, which are combined to form the Secret's unique identifier (as a ResourceName is unique within its respective Namespace). Using this identifier, the plugin stores information related to the Secret in a key–value database.

Secrets generation requests should also specify the resources authorized to access each Secret. Secrets are created with a specific purpose and designated resources in mind, and a Secret's owner typically knows exactly which resources will require access. As illustrated in Listing 1, we use the built-in `annotations` attribute [27] within the metadata section. This allows the Secret owner to define which resources are authorized to access the Secret. To ensure uniqueness and avoid duplication when identifying an authorized resource, we use a combination of the `ResourceName`, `ResourceKind`, and `ResourceNamespace` as the resource identifier. To eliminate the need for manual intervention, KubeKeeper uses an automated approach to identify all authorized resources for each Secret during application deployment. It processes the application's Kubernetes manifests, extracts all mappings between Secrets and resources, and uses these mappings to populate the annotation value for each Secret, indicating the resources that have mounted it.

If present, the `SecretMutation` Admission Plugin also extracts the `secret-ownerships` annotation, which identifies the authorize resources for the Secret. Then, the plugin either generates a new encryption key for

the Secret or retrieves an existing one from the database using the Secret's identifier. The corresponding value includes the encryption key, ownership information, and the Service Account that created the Secret. By securely storing these details, KubeKeeper supports future updates or modifications to the Secret. This information is accessible through another plugin, `DeploymentMutation`, which ensures that only authorized owners can access the encrypted data. The Service Account is used by the `SecretMutation` Admission Plugin to enforce access control, allowing only the Service Account that created the Secret to update its authorized owners (i.e., `secret-ownerships`).

The plugin proceeds to extract sensitive data within the Secret (e.g., tokens) and encrypts it. Finally, it replaces the original unencrypted data with its encrypted version before sending the protected Secret back to the API server. The encrypted Secret is then stored in `etcd`, consistent with Kubernetes' default behavior. Although Kubernetes can encrypt Secrets at rest if encryption is enabled [43], this feature is not enabled by default. The crucial difference is that KubeKeeper not only ensures encryption at rest by default, but also protects Secrets in transit. Since encryption keys are stored in a separate database secured by the API server, even if attackers compromise `etcd` or gain over-privileged access to Secrets via the API server, they cannot access the decrypted content.

5.4. Workload Authorization Verification

KubeKeeper uses the `DeploymentMutation` Plugin for requests related to workload deployments that require access to protected Secrets. As shown in Listing 2, a deployed resource just needs to add the `protected-secret-access` label and set its value to `true`. Upon receiving a request, the `DeploymentMutation` Plugin verifies it before delivering the encryption key for each Secret. This verification process is illustrated in the top part of Figure 4. The Plugin intercepts only Pod creation events, as all workload deployments (e.g., StatefulSets, Deployments) ultimately result in the creation of one or more Pods. Therefore, instead of hooking different workload resources, we focus solely on intercepting all Pod creations.

Listing 2: YAML manifest file for Pod deployment.

```yaml
kind: Pod
metadata:
  name: example-pod
  namespace: production
  labels:
  protected-secret-access: "true"
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: secret-data
  containers:
    - name: test-container
      image: nginx:1.14.2
      volumeMounts:
        - name: secret-volume
          mountPath: "/etc/secret-volume"
```

Secrets are identified based on the `SecretName` attribute specified in manifest files. For each mounted Secret, the Plugin extracts its unique identifier and retrieves the corresponding encryption key from the `KeyStore` database. If the requesting resource's identifier is not listed among the authorized resources for even a single Secret, the request is denied. Otherwise, the encryption key for each Secret is retrieved and prepared for delivery to the Pod. When the `DeploymentMutation` Plugin is started, it loads a configuration policy file that specifies the Service Accounts and the Namespaces in which they are only allowed to create Pods within their Namespaces. We use this feature to further restrict less secure Service Accounts, including those used by third-party applications, to prevent them from being misused by unauthorized users.

5.5. Selective Encryption Key Distribution

To make the decryption process transparent to applications that consume Secrets, we use an `init` container. Figure 4 (bottom) illustrates the key distribution process, where the Plugin rewrites the Pod specification to support transparent decryption through an injected `init` container. This container is responsible for mounting and decrypting Secrets before the main application containers start, and shares the decrypted Secrets through shared Volumes with the main containers. Therefore, this container must have access to both the encryption keys and the encrypted Secret values. The `DeploymentMutation` Admission Plugin adds an `init` container with a customized specification that provides access to these encryption keys and Secret values and uses a custom container image. Listing 3 shows the Pod manifest from Listing 2 after being processed by the `DeploymentMutation` Plugin.

Listing 3: The *spec* part from Listing 2 after being processed by the `DeploymentMutation` Plugin.

```yaml
spec:
  volumes:
    - name: encrypted-secret-volume
      secret:
        secretName: secret-data
    - name: secret-volume
      emptyDir:
        medium: Memory
  initContainers:
    - name: init-myapp
      image: decrypt-image:v1.0.0
      env:
        - name: ENCRYPTION_KEY_new_secret_data
          value:
              "bTVaQb08+DIn5qkemh83RRkYnLaUpFXITZ"
      volumeMounts:
        - name: encrypted-secret-volume
          mountPath:
              "/etc/encrypted-secret-volume"
        - name: secret-volume
          mountPath: "/etc/secret-volume"
  containers:
    - name: test-container
      image: nginx:1.14.2
      volumeMounts:
        - name: secret-volume
          mountPath: "/etc/secret-volume"
```

5.6. Secrets Decryption in the Init Container

To modify the workload specification, the Plugin scans the Pod's Volume specifications to identify Volumes that

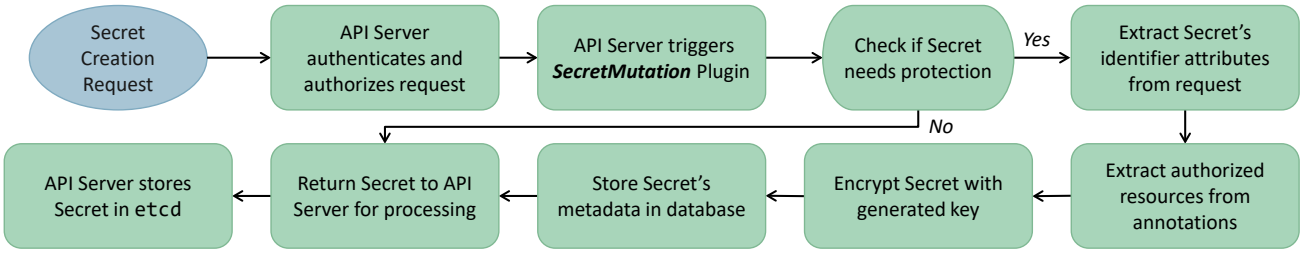

Figure 3: Workflow of KubeKeeper's `SecretMutation` Admission Plugin for encrypting Secrets.

reference sensitive Secrets. First, for each identified Volume, it creates a new Volume that refers to the same Secret, using the prefix `encrypted-` to distinguish it from the original. These prefixed Volumes are used to mount the encrypted Secrets. Additionally, a shared Volume is created using the original Volume name, allowing application containers to access the decrypted Secrets at the expected mount path. However, instead of mounting Secrets directly from the API server, application containers retrieve them from the shared Volumes populated by the `init` container.

Next, the Plugin generates a customized `init` container specification that mounts the prefixed `encrypted-` Volumes. All Secret encryption keys are injected into the `init` container's environment variables (`env` section in Listing 3). These keys are used to decrypt the Secrets before they are made available to the application containers.

Since third-party applications or other resources may have access to both the Secrets and the Pod description (via RBAC permissions), they could potentially read the environment variables and extract the encryption keys directly from the Pod specification. To mitigate this risk, we use envelope encryption [20], in which each data encryption key (DEK) is itself encrypted using a Key Encryption Key (KEK). This KEK is made available to the `init` container during its image creation. A unique KEK is generated for each Node, and the appropriate key is selected based on the Node that will host the Pod. The KEK can either be hard-coded into the `init` container or securely retrieved at runtime from an external Secrets management service.

The init container's primary role is to process the encrypted Secrets and make them available through Shared Volumes for application containers. Upon startup, the `init` container retrieves the encryption keys from the environment variables, decrypts them using the loaded KEK, and then uses them to decrypt the Secrets.

All encrypted Secrets are mounted in directories named `/etc/encrypted-xxx`, where `xxx` refers to the Secret Volume names. The `init` container reads the encrypted values, identifies the appropriate encryption key based on the mapping between Volume paths and Secret names, and performs decryption. The decrypted content is then written to a shared `in-memory` Volume, allowing application containers to access the decrypted Secrets transparently, without requiring any changes to application code.

5.7. Preventing Indirect Workload Deployment

As mentioned in Section 5.4, to further restrict less secure Service Accounts, we only allow them to deploy Pods within their own Namespaces, even if they are permitted to create Pods anywhere in the cluster, including on a compromised Node. An attacker could still deploy a Pod on a compromised Node by modifying other Worker Nodes in a way that makes them unavailable or forces them to repel their workloads. By applying such changes to all other Worker Nodes, authorized Pods with access to protected Secrets could be evicted from those Nodes and forced to run on the attacker-controlled Worker Node [92].

To prevent this attack, KubeKeeper has a Validating Admission Plugin (`NodeValidation`) that intercepts any attempts of forcing a workload to run on a compromised Node, such as setting the `node.kubernetes.io/unschedulable` taint to `NoExecute`. A taint is a property of Worker Nodes that allows them to repel certain Pods [34]. KubeKeeper's Validating Plugin performs real-time checks on Node configuration updates, blocking unauthorized changes before they impact the cluster. If the request is initiated by a less secure Service Account, as defined in the configuration policy file, the request is automatically denied.

6. Implementation

The core component of KubeKeeper is its Admission plugins. While Kubernetes supports compiled-in plugins integrated into the API server, we implement KubeKeeper using dynamic Admission Webhooks for simplicity and ease of deployment. The core logic of an Admission Plugin—intercepting and modifying Secret and Pod creation requests—remains the same regardless of whether it is implemented as a Webhook or compiled directly into the API server. These Webhooks offer equivalent functionality without requiring modifications to the Kubernetes source code, making them ideal for seamless integration with existing clusters [3], [8]. This choice does not limit KubeKeeper's capabilities, as the same logic can be migrated to a compiled-in plugin for tighter integration in future Kubernetes versions.

We implemented KubeKeeper on top of Kubernetes v1.29.4, leveraging its dynamic Admission Control features [42] to build a secure Webhook Server. The Webhook Server was developed using Go v1.22.2, while for our *KeyStore* database component we used BoltDB, a pure Go key–value store [40]. Docker was used for container management, and Helm [45] was used for deploying the Webhook Server and associated resources. Our implementation integrates the Webhook Server into Kubernetes clusters to protect Secrets using several Webhook APIs provided by the Kubernetes Admission controller. Additionally, we implemented a tool using Python to extract the ownership annotations of Secrets.

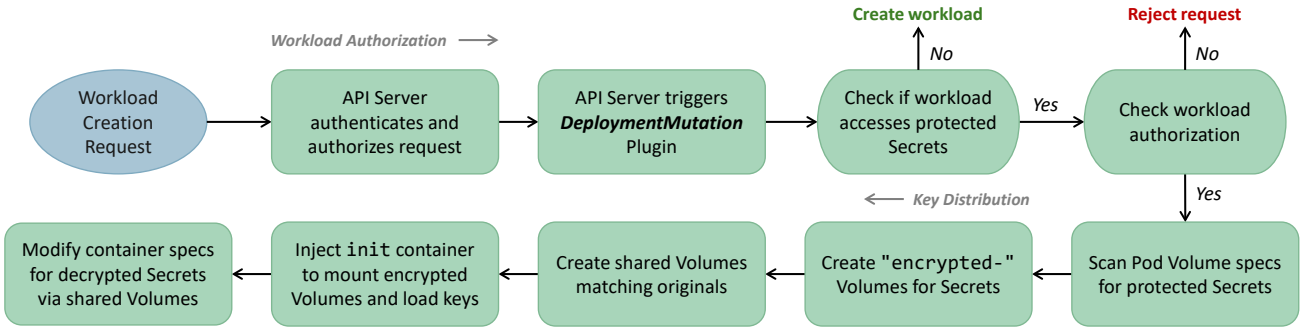

Figure 4: Workflow of KubeKeeper's `DeploymentMutation` Plugin. The top section verifies requests before delivering the encryption keys for each Secret, while the bottom section handles key distribution for Pods.

6.1. Automated Secret Annotations

To automate the process of annotating Secrets, we developed a Python tool that scans Kubernetes application manifests for YAML configuration files. The tool identifies resources that mount Secrets and extracts the `ResourceName`, `ResourceKind`, and `ResourceNamespace` for each as identifiers, and maps them to each mounted Secret. The tool also produces a comprehensive mapping of all Secrets and the corresponding resources that have mounted them. This enables owners of Secrets to use the output for annotating each Secret, ensuring they are safeguarded by KubeKeeper.

6.2. Webhook Server

At the core of KubeKeeper's implementation is a dedicated Webhook Server that enforces the Admission control policies. This Server receives and processes Admission requests from the Kubernetes API Server related to Secret and Pod creation. The Webhook Server intercepts Kubernetes API requests related to Secret creation via the `SecretMutation`, Pod creation via the `DeploymentMutation`, and Node changes via `NodeValidation` APIs. Since the first two APIs require modification of the request object, we use `MutatingWebhookConfiguration`, whereas the Node changes API is deployed as a `ValidatingWebhookConfiguration` to inspect requests and verify their validity.

Our Admission Webhook Server is a service that runs within the cluster and is hosted on a separate Node. As a regular Kubernetes Deployment, it requires a Kubernetes service to communicate with the Validating or Mutating Webhooks. The `WebhookConfiguration` objects instruct the Kubernetes API server to send relevant requests, such as those triggered by Secret or Pod creations, to the Webhook Server, as illustrated in Listing 4.

Webhook Server APIs parse the `AdmissionReview` object from the incoming request. This object contains information about the Service Account that triggered the webhook, as well as the Kubernetes object being created or modified (e.g., a Secret or Pod). The Webhook Server interacts with a BoltDB database to securely store or retrieve encryption keys, and ownership information associated with Kubernetes Secrets. We use *AES-GCM* for encryption and generate a new *256-bit* key for each unique Secret. The server also stores the Service Account that created the Secret to verify and support future updates to the Secret. When it receives a request to update ownership information, it ensures that the request comes from the same Service Account that originally created the Secret.

Listing 4: Simplified YAML manifest for the `SecretMutation` MutatingWebhookConfiguration deployment.

```
kind: MutatingWebhookConfiguration
metadata:
  name: SecretMutation
webhooks:
   namespaceSelector: {}
   rules:
     - operations: [ "CREATE", "UPDATE",
         "PATCH"]
       apiGroups: [ "" ]
       apiVersions: [ "v1" ]
       resources: [ "secrets" ]
   objectSelector:
     matchLabels:
       protected-secret: "true"
   clientConfig:
     service:
       name: Kubekeeper
       path: "/SecretMutation"
```

If changes are required for incoming requests, such as encrypting a Secret value, the Webhook server constructs a JSON patch that is included in the `AdmissionReview` response to instruct Kubernetes how to modify the object before it is persisted. The `deploying-pods` API also adds an `init` container to the Pod descriptions using a similar patch. It also adds a shared Volume for each Secret and injects the corresponding encryption keys into the `init` container as environment variables. These encryption keys are encrypted using a generated KEK (Key Encryption Key) for the target Node that is supposed to host the deploying Pod. The shared Volumes use in-memory storage with the `medium: Memory` attribute to prevent writing the decrypted data to persistent storage, similarly to Kubernetes Secret Volumes.

6.3. Decryption Process in Init Container

The `init` container is a critical component of KubeKeeper, designed to decrypt Kubernetes Secrets before before making them accessible to application containers. The `init` container is custom-built and runs a Go program specifically developed to manage the decryption process.

The program retrieves the necessary encryption keys from environment variables, which are securely provided by the Webhook Server. These keys are further encrypted with a KEK specific to the target Node where the Pod is deployed.

In our current implementation, the KEK is hard-coded within the container images, with access restricted through strong authentication and role-based access control (RBAC). However, in a production environment, we recommend using a Secrets management tool (e.g., HashiCorp Vault, AWS Secrets Manager) to securely inject the KEK into the container at runtime, rather than hardcoding it. This approach is discussed further in Section 8.

7. Experimental Evaluation

In this section, we evaluate the effectiveness of KubeKeeper in protecting Secrets against excessive permissions. We first present our static analysis tool, developed to identify excessive permissions in Kubernetes configurations. We then validate the tool through real-world case studies, demonstrating its capability to accurately detect overprivileged configurations and assessing how effectively our approach safeguards Secrets against these risks.

Our analysis spans three key categories: 1) all third-party applications from CNCF projects [17], [21], [30], 2) third-party applications commonly used in Kubernetes services provided by the top-four cloud vendors, and 3) custom applications sourced from GitHub and GitLab. We ran our performance evaluation experiments on a server equipped with an Intel Xeon E3-1240 CPU and 32GB of RAM, running Ubuntu 20.04.5 and kernel v5.4.0-128.

7.1. Identifying Excessive Permissions

To thoroughly investigate excessive permissions that could expose sensitive data, we developed a tool that scans Kubernetes YAML files to identify excessive permissions for each application. Our tool automates the process by gathering information about all Service Accounts bound to a Kubernetes resource and a Role or ClusterRole with excessive permissions. While existing tools like a script from a previous study [92] and KubiScan [49] can also detect excessive permissions, we chose not to use them for several reasons.

First, they must be deployed on a live cluster. This means that we would have to manually install and run all 498 applications, which would have been extremely time-consuming. Additionally, dynamic analysis may be incomplete, due to the various ways third-party applications can be configured. In contrast, our tool only requires the source code that contains the configuration files, and automatically extracts all Service Accounts and resources with excessive permissions.

Figure 5 illustrates the workflow of our static analysis tool, which takes as input the URLs of GitHub and GitLab repositories and clones them to access the relevant YAML configuration files. When third-party applications use Helm [45] charts to define and install their applications, our tool renders the templates to generate manifests for permission analysis. It identifies Helm charts via the presence of a `values.yaml` file and a `templates/` directory, builds dependencies, and uses Helm to render the manifests. For directories containing Kustomize configurations, it uses `kustomize` to render them. The tool then analyzes the resulting manifest files to assess permissions granted through Roles/ClusterRoles and RoleBindings/-ClusterRoleBindings to Service Accounts, focusing on overprivileged permissions that could lead to unauthorized access to Secrets, following the strategies outlined by Yang et al. [92]. These permissions are listed in Table 1. While the tool considers all ClusterRoles that lead to unauthorized access to Secrets, Roles are deemed overprivileged only if defined in the `default` or `kube-system` Namespaces.

Our tool extracts relevant permissions for each Service Account, highlighting excessive permissions in ClusterRoles and Roles. Furthermore, KubeKeeper needs a configuration file containing all Service Accounts that are only allowed to create resources within their respective Namespaces. Therefore, the tool also identifies all Service Accounts and Namespaces for direct integration into the Webhook Server's configuration, reducing errors.

TABLE 3: Comparison of KubeKeeper's static analysis tool and the dynamic analysis tool by Yang et al. [92] in detecting Secrets-related excessive permissions across *40* CNCF Kubernetes applications. KubeKeeper detects all Secrets-compromising permissions with no missed cases.

Metric	KubeKeeper	Yang et al. [92]
Apps with Excessive Permissions	30	27
Apps with Accurate Extraction	26	4
Apps with Incorrect Reports	0	14
Secret Permissions Missed	0	94

7.2. Excessive Permissions Assessment

7.2.1. Dataset Collection. We analyzed a diverse dataset from previous research [78], [92] to identify excessive permissions that could lead to unauthorized access to Secrets. This analysis assessed security risks across different environments and evaluated the effectiveness of our approach in mitigating these risks. Third-party applications deployed within Kubernetes clusters—particularly those from the CNCF project list [17]—are especially relevant, as they often define their own access controls. The CNCF list includes 176 projects in `graduated`, `incubating`, and `sandbox` categories [21], [30]. A prior assessment revealed that 51 out of 153 CNCF projects (33.3%) had potential security risks [92].

Cloud vendors often use third-party applications to extend their features, which can introduce security risks by requesting excessive permissions [53]. Similarly to previous work [92], we expanded our data set by analyzing open-source applications from Google Kubernetes Engine (GKE) [53], Amazon Elastic Kubernetes Service (Amazon EKS) [38], Azure Kubernetes Service (AKS) [39], and Alibaba Cloud Container Service for Kubernetes (Alibaba Cloud ACK) [41], identifying 65 additional applications across these providers.

Excessive permissions are common in Kubernetes applications, where RBAC configurations are often manually specified by developers who may lack security expertise. To further investigate this issue, we examined an extra set

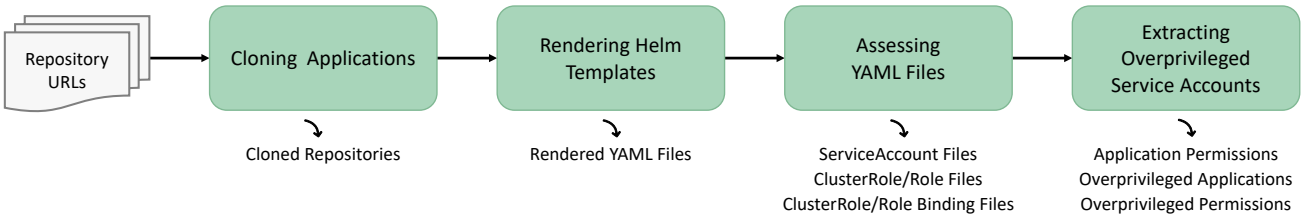

Figure 5: Overview of the workflow of our static analysis tool for identifying excessive permissions.

of 55 Kubernetes applications from previous research [78]. We also examined 202 more applications from GitHub, identified by searching for Helm configuration files. In total, we assessed *498* applications and found that *202* contained excessive permissions leading to unauthorized access to Secrets.

7.2.2. Comparison with Prior Tool. To evaluate the effectiveness of our excessive permissions extraction tool, we conducted a small-scale comparison against the tool by Yang et al. [92]. We selected a representative subset of CNCF applications from the `graduated` and `incubating` categories, and manually deployed each application in a controlled Kubernetes cluster. We then executed their tool on this live setup to extract the set of detected permissions for each application.

In parallel, we applied the KubeKeeper tool to the same set of applications. Unlike the prior tool [92], which inspects only the deployed state of applications, our approach conducts a more comprehensive analysis by rendering all possible configuration paths, including those defined by Helm chart values and Kustomize overlays. During this process, we enable configurations that represent the worst-case privilege scenarios—ensuring that all permissions which could potentially lead to unauthorized access to Secrets are revealed, even if they are disabled by default. As a result, KubeKeeper not only detects all excessive permissions identified by the prior work [92], but also uncovers additional permission paths that were missed in their analysis. For all cases where the tools produced differing results, we manually verified the permissions to determine which tool was accurate.

A summary of the comparison is shown in Table 3. KubeKeeper correctly extracted all excessive permissions in 26 applications and missed no secret-related cases, while the dynamic tool achieved full detection in only 4 cases and missed 94 secret-related permissions across all applications. For 14 applications, the dynamic tool reported excessive permissions that KubeKeeper did not flag. We manually inspected these cases and found that the reported permissions were either correctly confined to appropriate `API groups` relevant to the application or were restricted to their own namespaces. Therefore, we considered these as incorrectly reported permissions. The results demonstrate that our static analysis, when accounting for full configuration paths, provides more complete results.

7.2.3. Effectiveness Assesment. To assess the effectiveness of KubeKeeper, we ran our static analysis tool on all applications in our dataset. The tool identifies and extracts all RBAC permissions mentioned in Table 1 that

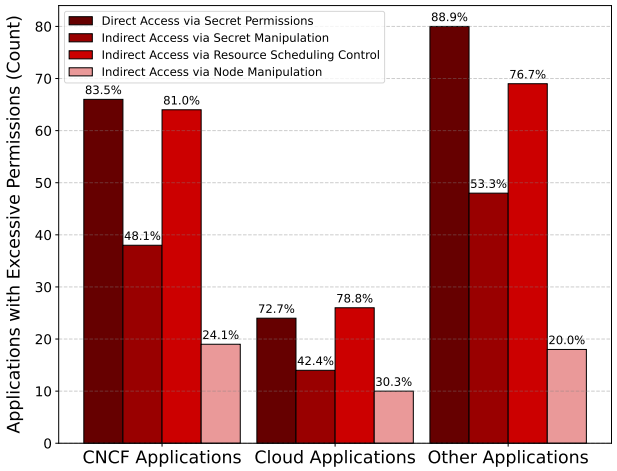

Figure 6: Breakdown of different types of unauthorized access to Secrets for the applications in our dataset.

are bound to a Service Account. Figure 6 shows the distribution of Secret-compromising permissions across various applications within different categories, highlighting the prevalence of overprivileged permissions. It also demonstrates how direct permissions to Secrets are often assigned carelessly to ClusterRoles/Roles—more than 72% of the applications in each category had this permission.

When a Service Account is granted permission to create resources within a cluster or a Namespace, this permission also gives it access to all Secrets within the defined Namespace. This is particularly dangerous because it gives the attacker permission to mount resources with a privileged Service Account. As seen in the figure, nearly 81% of applications in the *CNCF* Applications category and 79% of those in the *Cloud* Applications category received these permissions.

More than 40% of applications in each category have permissions to mutate Secrets, such as defining a token for a Service Account or updating Secret ownerships. Although updating Nodes appears to be a less common permission, the figure shows that approximately 24% of CNCF Applications, 30% of Cloud Applications, and 20% in the rest of the applications have this permission, allowing them to mark a Node as unschedulable or remove it from service.

Table 4 provides an overview of the number of application repositories in each category with excessive permissions, highlighting the prevalence of Secret-compromising vulnerabilities. The results show that 202 out of the 498 applications (41%) possess such permissions. Notably, 84% of these vulnerable applications have direct access to Secrets or excessive permissions related to resource

TABLE 4: Overview of application repositories containing Secrets-compromising excessive permissions across various categories. KubeKeeper protects the Secrets of *all* these applications against unauthorized access.

Application Categories	**Total**	**Number of Applications with Secret-Compromising Permissions**					**Protected by KubeKeeper**
		Count	Direct Access via Secret Permissions	Indirect Access via Secret Manipulation	Indirect Access via Resource Scheduling Control	Indirect Access via Node Manipulation	
CNCF Applications							
Graduated [21]	26	9 (34.6%)	9 (34.6%)	2 (7.7%)	6 (23.1%)	2 (7.7%)	✓
Incubating [21]	36	19 (52.8%)	18 (50.0%)	14 (38.9%)	17 (47.2%)	4 (11.1%)	✓
Sandbox [30]	114	51 (44.7%)	39 (34.2%)	22 (19.3%)	41 (36.0%)	13 (11.4%)	✓
Sum	176	79 (44.9%)	66 (37.5%)	38 (21.6%)	64 (36.4%)	19 (10.8%)	
Cloud Applications							
Google Kubernetes Engine [53]	17	8 (47.1%)	6 (35.3%)	1 (5.9%)	6 (35.3%)	1 (5.9%)	✓
Amazon Elastic Kubernetes Service [38]	25	15 (60.0%)	10 (40.0%)	8 (32.0%)	13 (52.0%)	6 (24.0%)	✓
Azure Kubernetes Service [39]	8	3 (37.5%)	2 (25.0%)	0 (0.0%)	1 (12.5%)	0 (0.0%)	✓
Alibaba Cloud Container Service [41]	15	7 (46.7%)	6 (40.0%)	5 (33.3%)	6 (40.0%)	3 (20.0%)	✓
Sum	65	33 (50.8%)	24 (36.9%)	14 (21.5%)	26 (40.0%)	10 (15.4%)	
Other Applications							
Public Dataset [78]	55	10 (18.2%)	9 (16.4%)	3 (5.5%)	5 (9.1%)	1 (1.8%)	✓
Additional Applications	202	80 (39.6%)	71 (35.1%)	45 (22.3%)	64 (31.7%)	17 (8.4%)	✓
Sum	257	90 (35.0%)	80 (31.1%)	48 (18.7%)	69 (26.8%)	18 (7.0%)	
Total	**498**	**202 (40.6%)**	170 (**34.1%**)	100 (**20.1%**)	159 (**31.9%**)	47 (**9.4%**)	

TABLE 5: Frequency of Secret-compromising permission types across application categories. KubeKeeper protects these applications against *all* four types of excessive permissions.

Excessive Permission Types	**The Frequency of Excessive Permissions**			**Total (Protected by KubeKeeper %)**
	CNCF Applications	Cloud Applications	Other Applications	
Direct Access via Secret Permissions	605 (32.4%)	328 (40.5%)	1,160 (37.8%)	2,093 (100%)
Indirect Access via Secret Manipulation	147 (7.9%)	66 (8.2%)	396 (12.9%)	609 (100%)
Indirect Access via Resource Scheduling Control	1,021 (54.7%)	376 (46.4%)	1,436 (46.8%)	2,833 (100%)
Indirect Access via Node Manipulation	93 (5.0%)	40 (4.9%)	76 (2.5%)	209 (100%)
Total	**1,866**	**810**	**3,068**	**5,744 (100%)**

scheduling control. Table 5 further quantifies the results by detailing the frequency of each type of excessive permission across different application categories. There are 2,093 permissions that provide direct access to Secrets through Secret permissions and 2,833 that do so indirectly by controlling resource deployment. In general, these findings illustrate the prevalence and severity of permission mismanagement in Kubernetes environments.

KubeKeeper is effective in preventing unauthorized access to Secret data through *all* types of excessive permissions across all applications categories, as demonstrated in Tables 4 and 5. For *Direct Access via Secret Permissions*, KubeKeeper ensures that even if a Service Account is granted access to all Secrets, encrypted Secrets remain accessible only through the APIs, as they are fetched from `etcd`. In the case of *Indirect Access via Secret Manipulation* permissions, such as `patch` or `update`, KubeKeeper limits the ability to update deployed Secrets to the Secret's owner Service Account—the one that deployed the Secret. This prevents unauthorized Service Accounts from altering Secret content or ownership annotations.

Permissions that allow the creation or modification of workloads, categorized as *Indirect Access via Resource Scheduling Control*, can be exploited to deploy resources on compromised Nodes. KubeKeeper counters this by restricting decryption keys to pre-authorized Pods, and preventing unauthorized resource deployment by specifying which Service Accounts are only permitted to create Pods within their respective Namespaces. This policy is enforced by the `deploying-pod` Webhook. These Service Accounts, often linked to third-party apps, tend to be less secure. Therefore, if a compromised component attempts to create a workload within another Namespace that is authorized to access sensitive data, KubeKeeper denies the request. However, while the compromised component can still deploy workloads within its own Namespace, these components are not permitted by other Secret owners to access their Secrets.

Attackers may also misuse Node manipulation permissions, categorized as *Indirect Access via Node Manipulation*, to force sensitive workloads onto compromised Nodes. KubeKeeper's `NodeValidation` Validating Webhook API blocks such actions by prohibiting unauthorized Service Accounts from updating the Node properties.

7.2.4. Performance Evaluation. KubeKeeper minimizes its impact on application performance by limiting its major operations mostly during the deployment phase, and specifically during the creation of Secrets and Pods. Unlike solutions like HashiCorp Vault's *vault-k8s* [55], which require ongoing interactions with external systems, KubeKeeper performs encryption and decryption only once during initialization and deployment. Afterwards, applications run without any performance overhead, as Secrets are securely integrated. Webhook calls are triggered only during the initial deployment of Secrets and Pods, using specific labels (`protected-secret` and `protected-secret-access`), without affecting the

creation of other resources.

We experimentally evaluated the overhead during Pod creation and deployment with second-level precision. This level of precision was chosen as Kubernetes also uses second-level timestamps, given that most events occur over a span of seconds or longer. Since Secrets are generated before being used by a resource, their creation does not impact runtime performance and occurs only once prior to Pod deployment.

For Pods, the `DeploymentMutation` Webhook intercepts and modifies the Pod specification during deployment, specifically between Pod creation and deployment. This introduces a one-time overhead during the initialization phase, where the init container is responsible for decrypting Secrets before the main container starts. We measured this delay using timestamps recorded during different stages of the deployment process, such as `StartAt` and `creationTimestamp`. The `creation-delay` was calculated as the time taken from the initiation of the deployment to when the Pod is created by Kubernetes. Additionally, `startup-delay` measures the time from when the Pod is created to the moment it enters the Running or Completed state.

To evaluate Pod creation and deployment latency, we conducted an experiment using a Deployment with 10 replicas, each creating 10 instances of the following popular containers: Nginx, Redis, Node.js, Python, PostgreSQL, Elasticsearch, and Jenkins. We developed a Python script that interacts with Kubernetes APIs to fetch timestamps for the Pods. We measured the latency in two scenarios: first, without accessing KubeKeeper-protected Secrets to measure Kubernetes' baseline delays, and second, when mounting a protected Secret to measure KubeKeeper's impact. We recorded the creation and startup times for all 10 Pods in each case and calculated the average latency. Our results show that KubeKeeper introduces no significant overhead during creation and deployment, with second-level precision indicating that any overhead is *less than one second*.

7.2.5. Security Evaluation. To evaluate the security of KubeKeeper, we analyze how it mitigates specific attack vectors commonly associated with Secret-compromising excessive permissions in Kubernetes environments. Building on prior work that highlights the risks of excessive permissions in third-party applications [92], we focused on the four critical categories of permissions listed in Table 1. We implemented proof-of-concept attacks using a Kind cluster [46], simulating a Kubernetes environment with two Worker Nodes and one Master Node. One Worker Node hosted a vulnerable container alongside third-party applications, mirroring the setup shown in Figure 1.

In our simulated attacks, an adversary gains control of a compromised Node and acquires JWT tokens of all running Pods. These tokens enable the attacker to interact with the Kubernetes API Server, thereby gaining the associated privileges. The attack scenarios align closely with those described in Section 3. We installed third-party applications from CNCF projects, each containing Secret-compromising excessive permissions to simulate the potential attack vectors. Three applications with the first three categories of excessive permissions were chosen from `graduated` projects, while an application from the `incubating` projects was used for the Node manipulation category. We are working with the developers of these applications to address these issues, so their names are not disclosed. All the Service Accounts related to these and their Namespaces were loaded into the Webhook Server during initialization. We deployed one Secret and annotated it to restrict access to only one specific Deployment.

Our evaluation demonstrates that when an attacker uses a compromised JWT token to fetch Secrets via Kubernetes APIs, only encrypted versions are retrieved. Attempts to update the Secret using the compromised token are denied by KubeKeeper's *SecretMutation* Webhook Server API, as the Service Account making the request differs from the Secret's creator. Additionally, attempts to patch a Node or deploy an authorized resource with access to a Secret on the compromised Node are blocked. The *DeploymentMutation* and *NodeValidation* Webhook APIs extract Service Accounts from the requests, compare them with untrusted accounts, and deny the requests. Therefore, KubeKeeper mitigates these risks by enforcing strict, fine-grained access control and encryption, ensuring that even if a Node is compromised, the attacker cannot exploit excessive permissions to access Secrets.

8. Limitations and Discussion

Encryption and Key Management In the current prototype of KubeKeeper, Key Encryption Keys (KEKs) are hardcoded into the init container images. For production, using an external Key Management Service (KMS) is recommended to securely manage decryption without exposing the KEK [20]. In this flow, all `init` containers on a Node share the same Service Account, using its token mounted in the `init` container to authenticate with the KMS. The KMS controls access to KEKs, ensuring the Service Account retrieves only its authorized key. The Webhook API server maps the appropriate KEK based on the Node scheduled to host the Pod.

Updating Secrets Currently, KubeKeeper supports updates to deployed Secrets, particularly for attributes such as `secret-ownerships`, which are crucial for creating new workloads that depend on existing Secrets. The `SecretMutation` API handles update requests by retrieving the relevant Secret from the `KeyStore` database, encrypting its sensitive data, and updating `secret-ownerships` if necessary. If `secret-ownerships` are modified, the request must originate from the account that deployed the Secret.

When sensitive data is updated, future workloads will use the new value. However, if a Secret is already mounted on a Pod, the Kubelet is responsible for updating the value [32]. This can result in some Pods having outdated Secret values until the update is fully propagated. Therefore, it is recommended to manually trigger a rolling update of the deployment when a Secret changes [29].

In KubeKeeper, decrypted Secrets are loaded from shared Volumes rather than directly from Secrets, so updates are not propagated automatically. The current approach involves restarting resources through rolling updates, allowing the init container to fetch and share the updated Secrets. A more efficient approach would use a `Sidecar` container that mounts all encrypted Secrets,

decrypts them when updated by the `Kubelet`, and updates the values in the shared Volumes.

Secret Access and Consumption Secrets can be consumed as environment variables instead of mounted as Volumes [32], but this approach has a significant drawback. Updates require container restarts, during which environment variables may expose Secrets via logs or processes [72], [74]. Currently, KubeKeeper supports Volume-based consumption, using init containers to decrypt Secrets before application startup. It can be extended to support environment variables by detecting Secret references, decrypting them in an init container, and injecting an entrypoint script to export them as environment variables at runtime.

KubeKeeper also restricts `get` access to Secrets, ensuring only authorized workloads are permitted to retrieve or consume them. While this may slightly affect workflows relying on direct API access, it is generally not a limitation. Since Admission Control does not handle `get` requests, supporting this use case would require minor modifications to the API server's Secret fetch logic to enforce KubeKeeper's authorization policy and return decrypted data accordingly.

Scalability Across Platforms While we implemented KubeKeeper on Kubernetes, it can be adapted for use with other container management platforms that share similar architectural frameworks. Platforms such as Red Hat OpenShift [52], Azure Kubernetes Service (AKS) [39], and Google Kubernetes Engine (GKE) [53] also use Kubernetes as their underlying technology. These platforms share the same core architectural components, such as API servers with admission controllers and access control mechanisms for managing Secrets, making the integration of KubeKeeper feasible across these environments.

9. Related Work

Secret Management Approaches Kubernetes' native Secret management is often delegated to an external Key Management Service (KMS) to enhance security [13], [36], [47]. Kubernetes supports "encryption at rest" by allowing Secrets stored in etcd to be encrypted using a Data Encryption Key (DEK), which is itself encrypted by a Key Encryption Key (KEK) managed by a KMS plugin.

While this setup enhances the confidentiality of Secrets at rest, it lacks fine-grained runtime access control—any Pod in the same Namespace with sufficient permissions can mount a Secret in plaintext, introducing risks of excessive or misallocated access, especially from overprivileged third-party applications. KubeKeeper addresses these issues by establishing an automatic direct link between Secrets and their authorized Pods, eliminating the need for the complex mappings required by RBAC. Additionally, KubeKeeper differs from methods like Sealed Secrets [31], which focus on encrypting secrets for storage but decrypt them upon deployment. By keeping secrets encrypted until they are used, KubeKeeper significantly enhances confidentiality and security.

Kubernetes Security Challenges Extensive research has focused on the security of Kubernetes environments, particularly around system vulnerabilities and misconfigurations. Configuration analysis has effectively identified common misconfigurations [66], [78], [84] and violations of security best practices [67]. Studies have also shown how excessive permissions in third-party applications runnning on a cluster can compromise the entire cluster [92]. EPScan is another tool that detects excessive RBAC permissions in Kubernetes applications by combining static configuration analysis with LLM-assisted program behavior modeling at the Pod level [63]. However, it lacks available source code and datasets, limiting the ability to directly compare its findings with our results.

Several other studies have explored attack vectors within Kubernetes clusters. Spahn et al. [86] used a high-interaction honeypot to assess threats to exposed containers and orchestration systems. Additional research has examined privilege escalation attacks [71], co-residency attacks [65], [85], DDoS attacks targeting auto-scaling [57], and DevOps pipeline vulnerabilities [77]. Beyond Kubernetes, LeakLess [81] has addressed data leakage in serverless computing platforms using selective in-memory encryption to protect sensitive data against memory leakage attacks. In contrast, KubeKeeper focuses on Kubernetes-specific infrastructure threats.

Broader efforts to enhance container security have included creating system call profiles [59]–[62], [80], automating AppArmor policy generation [23], [28], [93]–[95], designing Namespaces for autonomous control [87], developing network-level security enforcement [75], and detecting malicious container images [58]. However, these approaches do not mitigate excessive permission risks in Kubernetes. KubeKeeper is, to the best of our knowledge, the first approach designed to prevent unauthorized access to Secrets caused by excessive permissions or insecure default configurations in Kubernetes clusters.

10. Conclusion

We presented KubeKeeper, a robust solution designed to address the security limitations of Kubernetes' native Secret management, particularly the risks associated with insecure configurations and excessive permissions granted to third-party applications. KubeKeeper enhances Kubernetes Secret management through automatic encryption and tightly controlled decryption, ensuring that only explicitly authorized Pods can access decrypted data. By leveraging Kubernetes' dynamic Admission Control, KubeKeeper operates transparently, without requiring any changes to the underlying infrastructure or application code. Our evaluation, covering 498 Kubernetes applications, demonstrated that excessive permission vulnerabilities are indeed prevalent in Kubernetes environments. However, KubeKeeper effectively mitigates these security risks without causing performance degradation during runtime or introducing significant overhead during creation and deployment. These results position KubeKeeper as an effective and scalable solution for improving the security of Kubernetes clusters, offering a practical approach to safeguarding sensitive information in complex, containerized environments.

Acknowledgments. We thank the anonymous reviewers for their constructive feedback. We also thank Jie Chen for helping us with the assessment of excessive permissions. This work was supported by the National Science Foundation (NSF) through award CNS-2104148.

References

[1] Kubernetes Components. https://Kubernetes.io/docs/concepts/overview/components/.

[2] The Kubernetes API. https://Kubernetes.io/docs/concepts/overview/Kubernetes-api/.

[3] A Guide to Kubernetes Admission Controllers. https://Kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/, 2019.

[4] How do I prevent Pods from Mounting Secrets in the Same Namespace? https://stackoverflow.com/questions/61519188/how-do-i-prevent-pods-from-mounting-secrets-in-the-same-namespace, 2020.

[5] StackRox State of Container and Kubernetes Security Report Reveals Rapid Growth across Container and Kubernetes Adoption, Security Incidents, and DevSecOps Initiatives. https://www.prnewswire.com/news-releases/stackrox-state-of-container-and-Kubernetes-security-report-reveals-rapid-growth-across-container-and-Kubernetes-adoption-security-incidents-and-devsecops-initiatives-301136399.html, 2020.

[6] With Kubernetes, the U.S. Department of Defense Is Enabling DevSecOps on F-16s and Battleships. https://www.cncf.io/blog/2020/05/07/with-Kubernetes-the-u-s-department-of-defense-is-enabling-devsecops-on-f-16s-and-battleships/, 2020.

[7] Managing Permissions with Kubernetes RBAC. https://www.paloaltonetworks.com/cyberpedia/Kubernetes-rbac, 2021.

[8] What's the difference between a Kubernetes Admission controller plugin and an Admission webhook?, 2021. Accessed: 2025-03-06.

[9] Run a Pod in a Namespace Without Having Access to its Secrets? https://www.reddit.com/r/Kubernetes/comments/w5g4ro/run_a_pod_in_a_namespace_without_having_access_to, 2022. Accessed: 2025-04-15.

[10] Ability to Create Pods Allows Access to Secrets in the Same Namespace #116188. https://github.com/Kubernetes/Kubernetes/issues/116188, 2023.

[11] Annotations. https://Kubernetes.io/docs/concepts/overview/working-with-objects/annotations/, 2023.

[12] Controlling Access to the Kubernetes API. https://Kubernetes.io/docs/concepts/security/controlling-access/, 2023.

[13] How to Secure Your API Secret Keys From Being Exposed? https://www.spectrocloud.com/blog/effective-secrets-management-in-Kubernetes-a-hands-on-guide, 2023.

[14] Restricting Secret Mounting for Pods. https://discuss.Kubernetes.io/t/restricting-secret-mounting-for-pods/26140, 2023.

[15] Workloads. https://Kubernetes.io/docs/concepts/workloads/, 2023.

[16] Best practices for Advanced Scheduler Features in Azure Kubernetes Service (AKS). https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-advanced-scheduler, 2024.

[17] CNCF Projects Are the Foundation of Cloud Native Computing. https://www.cncf.io/, 2024.

[18] Configure Service Accounts for Pods. https://Kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/, 2024.

[19] Controllers. https://Kubernetes.io/docs/concepts/architecture/controller/, 2024.

[20] Envelope Encryption. https://cloud.google.com/kms/docs/envelope-encryption, 2024.

[21] Graduated and Incubating Projects. https://www.cncf.io/projects/, 2024.

[22] kube-proxy. https://Kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/, 2024.

[23] KubeArmor:Runtime Security Enforcement. https://kubearmor.com/, 2024.

[24] kubelet. https://Kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, 2024.

[25] Kubernetes - Kubectl Create and Kubectl Apply. https://www.geeksforgeeks.org/Kubernetes-kubectl-create-and-kubectl-apply/, 2024.

[26] Kubernetes Scheduler. https://Kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2024.

[27] Labels and Selectors. https://Kubernetes.io/docs/concepts/overview/working-with-objects/labels/, 2024.

[28] Manage AppArmor Profiles for Kubernetes Cluster. https://github.com/sysdiglabs/kube-apparmor-manager, 2024.

[29] Performing a Rolling Update. https://Kubernetes.io/docs/tutorials/Kubernetes-basics/update/update-intro/, 2024.

[30] Sandbox Projects. https://www.cncf.io/sandbox-projects/, 2024.

[31] Sealed Secrets. https://fluxcd.io/flux/guides/sealed-secrets/, 2024.

[32] Secrets. https://Kubernetes.io/docs/concepts/configuration/secret/, 2024.

[33] Service Accounts. https://Kubernetes.io/docs/concepts/security/service-accounts/, 2024.

[34] Taints and Tolerations. https://Kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/, 2024.

[35] Tenant isolation. https://docs.aws.amazon.com/eks/latest/best-practices/tenant-isolation.html, 2024.

[36] Using a KMS Provider for Data Encryption. https://Kubernetes.io/docs/tasks/administer-cluster/kms-provider/, 2024.

[37] A Cloud Native Unstructured Data Storage. https://cubefs.io/, 2025.

[38] Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/, 2025.

[39] Azure Kubernetes Service (AKS). https://azure.microsoft.com/en-us/products/Kubernetes-service, 2025.

[40] BoltDB. https://dbdb.io/db/boltdb, 2025.

[41] Container Service for Kubernetes. https://www.alibabacloud.com/help/en/ack/, 2025.

[42] Dynamic Admission Control. https://Kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/, 2025.

[43] Encrypting Confidential Data at Rest. https://Kubernetes.io/docs/tasks/administer-cluster/encrypt-data/, 2025.

[44] etcd: A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System. https://etcd.io/, 2025.

[45] Helm: The Package Manager for Kubernetes. https://helm.sh/, 2025.

[46] Kind. https://kind.sigs.k8s.io/, 2025.

[47] Kubernetes Secrets: Secure Kubernetes Clusters with the Power of Vault and Dynamic Secrets. https://www.vaultproject.io/use-cases/kubernetes, 2025.

[48] Kubernetes User Case Studies. https://Kubernetes.io/case-studies/, 2025.

[49] KubiScan: A Tool to Scan Kubernetes Cluster for Risky Permissions. https://github.com/cyberark/KubiScan, 2025.

[50] Pods. https://Kubernetes.io/docs/concepts/workloads/pods/, 2025.

[51] Production-Grade Container Orchestration. https://Kubernetes.io/, 2025.

[52] Red Hat OpenShift. https://www.redhat.com/en/technologies/cloud-computing/openshift, 2025.

[53] The Most Scalable and Fully Automated Kubernetes Service. https://cloud.google.com/kubernetes-engine, 2025.

[54] Using RBAC Authorization. https://Kubernetes.io/docs/reference/access-authn-authz/rbac/, 2025.

[55] Vault + Kubernetes (vault-k8s). https://github.com/hashicorp/vault-k8s, 2025.

[56] Argonaut. Secret Management in Kubernetes: Approaches, Tools, and Best Practices. https://medium.com/@argonaut.dev/secret-management-in-Kubernetes-approaches-tools-and-best-practices-f1df77392060.

[57] Ataollah Fatahi Baarzi, George Kesidis, Dan Fleck, and Angelos Stavrou. Microservices made attack-resilient using unsupervised service fissioning. In *Proceedings of the 13th European workshop on Systems Security*, pages 31–36, 2020.

[58] Kelly Brady, Seung Moon, Tuan Nguyen, and Joel Coffman. Docker container security in cloud computing. In *Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0975–0980. IEEE, 2020.

[59] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, pages 139–151, 2021.

[60] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[61] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[62] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[63] Yue Gu, Xin Tan, Yuan Zhang, Siyan Gao, and Min Yang. EPScan: Automated detection of excessive RBAC permissions in Kubernetes applications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 11–11. IEEE Computer Society, 2024.

[64] Shaul Ben Hai and Artur Oleyarsh. Container Escape: New Vulnerabilities Affecting Docker and RunC. https://www.paloaltonetworks.com/blog/prisma-cloud/leaky-vessels-vulnerabilities-container-escape/, 2024.

[65] Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Transactions on Dependable and Secure Computing*, 14(1):95–108, 2015.

[66] Mubin Ul Haque, M. Mehdi Kholoosi, and M. Ali Babar. KGSecConfig: A knowledge graph based approach for secured container orchestrator configuration. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 420–431. IEEE, 2022.

[67] Shamim Shazibul Islam. 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*, pages 1689–1690, 2021.

[68] Google Kubernetes Engine (GKE). Isolate Your Workloads in Dedicated Node Pools. https://cloud.google.com/Kubernetes-engine/docs/how-to/isolate-workloads-dedicated-nodes, 2025.

[69] Picus Labs. The Ten Most Common Kubernetes Security Misconfigurations & How to Address Them. https://www.picussecurity.com/resource/blog/the-ten-most-common-Kubernetes-security-misconfigurations-how-to-address-them, 2024.

[70] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 418–429, 2018.

[71] Artem Linetskyi, Tetiana Babenko, Larysa Myrutenko, and Vira Vialkova. Eliminating privilege escalation to root in containers running on Kubernetes. *Scientific and Practical Cyber Security Journal*, 2020.

[72] Marko Luksa. *Kubernetes in Action*. Simon and Schuster, 2017.

[73] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem—vulnerability analysis. *Computer Communications*, 122:30–43, 2018.

[74] David Mytton. Storing Secrets in env vars Considered Harmful. https://blog.arcjet.com/storing-secrets-in-env-vars-considered-harmful/, 2024.

[75] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A security enforcement network stack for container networks. In *USENIX Annual Technical Conference (USENIX ATC 20)*, pages 81–95, 2020.

[76] Rani Osnat. Kubernetes Secrets: How to Create, Use, and Secure Them. https://www.aquasec.com/blog/managing-kubernetes-secrets/, 2024.

[77] Nicholas Pecka, Lotfi Ben Othmane, and Altaz 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 (ICSSP-GSE)*, pages 45–49, 2022.

[78] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source Kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(4):1–36, 2023.

[79] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z Berkay Celik. Towards improving container security by preventing runtime escapes. In *IEEE Secure Development Conference (SecDev)*, pages 38–46, 2021.

[80] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. Confine: Fine-grained system call filtering for container attack surface reduction. *Computers and Security*, 132:103325, 2023.

[81] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. LeakLess: Selective data protection against memory leakage attacks for serverless platforms. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2025.

[82] Miles S. *Kubernetes: A Step-By-Step Guide for Beginners to Build, Manage, Develop, and Intelligently Deploy Applications by Using Kubernetes (2020 Edition)*. Independently Published, 2020.

[83] Haq Md Sadun, Thien Duc Nguyen, Franziska Vollmer, Ali Saman Tosun, Ahmad-Reza Sadeghi, and Turgay Korkmaz. SoK: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 207–207, 2024.

[84] Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. XI commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices. In *Proceedings of the IEEE Secure Development Conference (SecDev)*, pages 58–64. IEEE, 2020.

[85] Sushrut Shringarputale, Patrick McDaniel, Kevin Butler, and Thomas La Porta. Co-residency attacks on containers are real. In *Proceedings of the 11th ACM SIGSAC Cloud Computing Security Workshop (CCSW '20)*, pages 53–66, 2020.

[86] Noah Spahn, Nils Hanke, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. Container orchestration Honeypot: Observing attacks in the wild. In *Proceedings of the International Symposium on Research in Attacks and Intrusions and Defenses (RAID)*, pages 381–396, 2023.

[87] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: Making linux security frameworks available to containers. In *Proceedings of the 27th USENIX Security Symposium*, pages 1423–1439, 2018.

[88] Yossi Weizman. Secure Containerized Environments With Updated Threat Matrix for Kubernetes. https://www.microsoft.com/en-us/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-Kubernetes/, 2021.

[89] Yossi Weizman. Container Breakout Vulnerabilities. https://www.container-security.site/attackers/container_breakout_vulnerabilities.html, 2024.

[90] Katrine Wist, Malene Helsem, and Danilo Gligoroski. Vulnerability analysis of 2500 docker hub images. In *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*, pages 307–327. Springer International Publishing, 2021.

[91] Ann Yi Wong, Eyasu Getahun Chekole, Martín Ochoa, and Jianying Zhou. On the security of containers: Threat modeling, attack analysis, and mitigation strategies. *Computers and Security*, 2023.

[92] Nanzi Yang, Wenbo Shen, Jinku Li, Xunqi Liu, Xin Guo, and Jianfeng Ma. Take over the whole cluster: Attacking Kubernetes via excessive permissions of third-party applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 3048–3062, 2023.

[93] Hui Zhu and Christian Gehrmann. AppArmor profile generator as a cloud service. In *International Conference on Cloud Computing and Services Science (CLOSER)*, pages 45–55, 2021.

[94] Hui Zhu and Christian Gehrmann. Lic-Sec: An enhanced AppArmor Docker security profile generator. *Journal of Information Security and Applications (JISA)*, 61:102924, 2021.

[95] Hui Zhu and Christian Gehrmann. Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. In *International Conference on COMmunication Systems and NETworks (COMSNETS)*, pages 129–137. IEEE, 2022.