PYRAMID: A Secure, Resource-Efficient, and Pluggable Kubernetes for Multi-Tenancy

Xiang Li
Tsinghua University
Beijing, China
China Telecom eSurfing Cloud (State Cloud)
Beijing, China
lixiang20@tsinghua.org.cn

Weijie Liu*
College of Cryptology and Cyber Science, DISSec, Nankai University
Tianjin, China
weijieliu@nankai.edu.cn

Fabing Li
Ant Group
Xi'an, China
lifabing1349212501@gmail.com

Hongliang Tian
Ant Group
Beijing, China
tate.thl@antgroup.com

Zheli Liu
College of Cryptology and Cyber Science, DISSec, AAIS, Nankai University
Tianjin, China
liuzheli@nankai.edu.cn

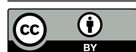
Shoumeng Yan*
Ant Group
Beijing, China
shoumeng.ysm@antgroup.com

Mingyu Gao*
Tsinghua University
Beijing, China
Shanghai Qi Zhi Institute
Shanghai, China
gaomy@tsinghua.edu.cn

Abstract

This work aims to achieve the best of both worlds with two prominent techniques adopted in cloud computing systems: hardware trusted execution environments (TEEs) for data processing security, and Kubernetes (k8s) for efficient container orchestration and resource management for multi-tenancy. A secure, resource-efficient, and pluggable container orchestration system, called PYRAMID, is proposed, which incurs minimal intrusive modifications to the commercial k8s. PYRAMID puts a separate trusted k8s on top of the original k8s cluster and carefully cooperates between the two layers. The workflow within each layer is maximally preserved without significant changes. The untrusted layer manages resource scheduling across different tenants to improve utilization and passes the resource information to the trusted layer to launch actual computations secured by TEEs, with the help of carefully designed interface and protection mechanisms. Evaluation results show that PYRAMID achieves 1.4× higher throughput on the data plane, with comparable control-plane performance to previous work.

*Corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland UK
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769324

CCS Concepts: • **Security and privacy** → **Systems security**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

Keywords: Trusted Execution Environment, Kubernetes, Container, Resource Management, Isolation

ACM Reference Format:
Xiang Li, Weijie Liu, Fabing Li, Hongliang Tian, Zheli Liu, Shoumeng Yan, and Mingyu Gao. 2026. PYRAMID: A Secure, Resource-Efficient, and Pluggable Kubernetes for Multi-Tenancy. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3767295.3769324

1 Introduction

Cloud computing, despite its many attractive advantages like easy deployment and flexible scalability, also raises data privacy concerns when sensitive data have to be outsourced to third-party public cloud platforms. To enable trusted computing in such untrusted environments, hardware vendors have been advocating security-enhanced processors that support trusted execution environments (TEEs), which allow isolated and verified processing on private data not observable by other software components. To better match the demands of virtualization in cloud systems, TEEs at the virtual machine

(VM) level are now the preferred choices, with commercial products including Intel TDX [44] and AMD SEV [4, 49].

On the other hand, as a more lightweight virtualization mechanism than classical VMs, containers have been widely used to efficiently deploy computation tasks on shared computing clusters. Kubernetes (k8s) [15] is the *de facto* framework to facilitate container deployment and management. It allows convenient orchestration along the entire lifecycle of containers, such as automatic restart after a crash, software update rollout, scalable scheduling, resource allocation and reclamation, and so on. Generally speaking, k8s is able to ensure flexible and efficient resource management for large-scale computing clusters, especially in multi-tenancy cases where many clients are concurrently using the system.

However, despite the broad attention to VM TEEs and the ubiquitous use of k8s, the study about how to combine their benefits in an easy and efficient way is relatively lacking. The current approaches remain straightforward, and are mostly designed for single-tenancy settings. For example, Constellation [85] directly puts the whole k8s software stack into VM TEEs to protect everything. While being secure, this *monolithic* approach sacrifices the *resource management efficiency* of k8s in multi-tenancy scenarios, where each tenant is forced to use an independent resource scheduler in separate TEEs, and these schedulers cannot coordinate with each other to manage resources, e.g., lending idle resources of one tenant to another. On the other hand, Confidential Containers (CoCo) [12, 48] only put the computing containers (organized as *Pods* in k8s) into their own TEEs. This allows to use a unified scheduler to manage all the in-TEE Pods across tenants, but such a *minimal* protection principle leaves several critical vulnerabilities that violate *security* (more in Section 3.3). Actually, we find that alleviating these drawbacks in the above two paradigms is highly non-trivial, and may require significant intrusive modifications to many k8s core components, which is difficult for such highly complex and production-grade software, if not impossible.

We aim to design a *secure, resource-efficient, and pluggable* k8s system for multi-tenancy to address the above challenges. Our goals include (1) strong security guarantees, (2) efficient and unified resource management across tenants, (3) no intrusive modifications to the k8s core logic, (4) easy operation and maintenance, and (5) small performance overheads. Our key idea is to *lay a separate TEE-based k8s on top of the original untrusted k8s cluster to reuse its workflow*, so that the complicated k8s workflow can be mostly preserved without significant changes, while still enabling unified resource management in the untrusted layer. Based on this idea, we implement a prototype system, and name it Pyramid.

More specifically, in Pyramid, the k8s scheduler in the untrusted layer is in charge of globally scheduling all resources, and we allow the trusted layer to expose the information about resource budget and resource consumption to this untrusted scheduler. Such information is typically insensitive, as the adversary who controls the physical computing servers can observe and modify it anyway. The untrusted scheduler thus allocates requested resources to *shadow Pods*, which are a novel abstraction in Pyramid as the resource-consuming representatives for the actual computing Pods in the trusted layer. With the allocated resources from the untrusted layer, the Pods in the trusted layer can be launched.

However, using an untrusted scheduler also introduces security problems. For example, while individual Pods are protected, the scheduler can still impact the execution orchestration flow, such as launching two Pods with the same name, to compromise the integrity of the high-level workload (e.g., having two leaders in a distributed computing software). Pyramid thus adopts an *exclusionary guard* mechanism, which lets the node hold a lock for a Pod, so that no other nodes can launch the same Pod, even in the challenging scenario where the adversary can arbitrarily disconnect any server node and manipulate the clock of a node.

We emphasize that we implement Pyramid in an add-on manner with minimum intrusive changes to the commercial k8s framework. For example, through the use of shadow Pods, the untrusted and trusted layers can preserve most of their own highly complicated workflow. The interaction between the two layers and the newly introduced mechanisms are mainly implemented using various existing extension points explicitly or implicitly provided by k8s. For example, the exclusionary guards utilize the k8s Lease objects, which are originally designed for node heartbeats and leader election purposes. Finally, we also introduce a performance optimization called *object divergence*, which separately stores insensitive configurations to mitigate access contention to the persistent metadata storage.

We evaluate two implementations of Pyramid: Pyramid-K directly running on k8s, and Pyramid-V integrated with the virtual cluster (Vcluster) [60]. We compare them against both the original insecure k8s system and the two straightforward design principles of monolithic and minimal, using a cluster of 1 master and 5 worker nodes. On the *control plane*, Pyramid-K and Pyramid-V introduce 1.17× and 1.23× overheads on average compared to the vanilla k8s. Besides, Pyramid-K and Pyramid-V achieve 1.08× and 1.02× speedups over Constellation, and 1.46× and 1.35× over CoCo. If not using object divergence, the overheads of etcd would make Pyramid slightly slower than Constellation. On the *data plane*, Pyramid outperforms Constellation with 1.4× throughput under 16 tenants, due to the ability of unified resource management. Note that Pyramid-K and Pyramid-V behave the same for data planes. Moreover, in a simulated large-scale and heterogeneous k8s cluster, Pyramid is able to save 3.5% CPU and 22.5% memory resources for long-running Pods, and 2.8× for short-lived Pods.

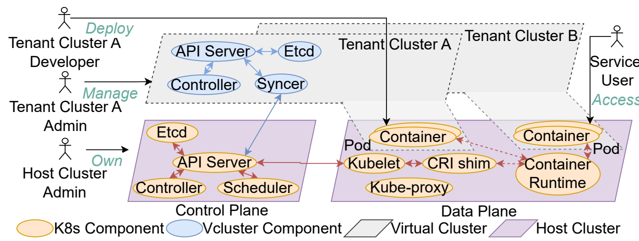


Figure 1. System architecture of k8s and Vcluster [60].

2 Background

2.1 Trusted Execution Environments

Trusted execution environments (TEEs) provide a trusted region inside an untrusted server. Commercial CPUs already support TEEs, including ARM TrustZone [2], Intel SGX [43], AMD SEV [4, 49], ARM CCA [9], and Intel TDX [44]. The trusted region is isolated from the untrusted region by hardware mechanisms. The trusted region could be part of a process, such as an enclave in Intel SGX [6, 41, 67], or a virtual machine (VM), such as a trusted domain (TD) in Intel TDX [44]. TEEs provide attestation mechanisms [6] for clients to verify that the desired code has been loaded into the trusted region. The potentially malicious privileged software like the host operating system (OS) or the VM hypervisor cannot compromise the confidentiality and integrity of the data and code inside the trusted region. Furthermore, in the trusted region, only the volatile states (e.g., in the CPU and memory) are protected, while the non-volatile storage is typically out of scope.

For process-based TEEs like Intel SGX [43], porting complex applications is typically painful and time-consuming [10, 20, 74, 84, 100], though better performance can be achieved for computation-intensive applications through careful implementation [56]. In contrast, VM-based TEEs like Intel TDX [44] and AMD SEV [4] are more friendly to users. Since a whole VM is protected, unmodified applications can directly run in the VM. In terms of engineering, it is easier to build trusted I/O for VM-based TEEs [5, 45], benefiting a wide variety of applications and heterogeneous computing tasks in a lift-and-shift way [30, 31, 51, 54, 62]. VM-based TEEs are thus more preferred nowadays, especially for cloud-native applications using containers. Therefore, in this work, we choose to use VM-based TEEs.

2.2 Container Orchestration: Kubernetes

As we scale up containers, manually managing them would be inconvenient. We need additional features, including self-healing crashed containers, automated rollouts and rollbacks of updates to containers, automatic scheduling of containers among server nodes, and so on. These features are collectively referred to as *container orchestration* [24].

Kubernetes (k8s) [15] is a prevalent and powerful framework to simplify the deployment and management of containers, and has become the *de facto* standard for container orchestration. We introduce the key concepts of k8s below; Appendix A gives a more detailed background.

Instead of a container, the smallest deployable unit in k8s is a *Pod*. It is a collection of closely related containers sharing a network namespace (the same IP address and port space), an IPC namespace, and optionally, the storage volumes. Therefore, containers in a Pod can efficiently communicate.

The **k8s API** is a resource-based (RESTful) API [78] and Pods are the fundamental API resource type. Directly deploying a Pod by creating a Pod object through the k8s command line tool called kubectl [21] could be hard to manage. Instead, high-level workload resources such as *Deployments*, *StatefulSets*, and *Jobs* are deployed through kubectl, and they typically represent a set of homogeneous Pods. K8s automatically manages these Pods, e.g., creating a new one for crash recovery. It is the high-level workload resources that embody the orchestration capability of k8s. Besides, k8s provides many other resources, such as *ConfigMaps* and *Secrets*, which contain configurations and sensitive metadata to be used by the Pods. Apart from the built-in resources, k8s also provides *custom resources* to extend the k8s API [13].

The **k8s architecture** itself is scalable and reliable, comprising several components (Figure 1). The *API server* is the central part of k8s, exposing the k8s API and handling the requests from the client (i.e., kubectl [21]). Then the admission controller in the API server intercepts the request, validates its behavior, and may modify the object in the request based on some rules, for example, filling the default values for missing fields in the object. The rules for validation and modification can be customized through the *admission webhooks*. Finally, the object is persisted to the *etcd*, a distributed and reliable key-value store with strong consistency [47, 88].

The *controller* continuously watches the current state of the cluster, and then makes or requests changes to the cluster, bringing it to the desired state. For example, after a StatefulSet object is received by the API server and stored in the etcd, the StatefulSet controller watching the etcd finds this new StatefulSet does not have Pods yet, and then starts creating the Pods through the API server. The *scheduler* watches for newly created Pods without node assignment and selects nodes for them to run on, i.e., binding a Pod to a node [23, 64] by updating the Pod object through the API server.

All the above components constitute the *control plane* in k8s. Those components can run on any machines in the cluster. We remark that etcd is the ground-truth. It could be locally deployed at the control plane node, or externally deployed as a separate cluster. In the production scenario, the latter is preferred for high availability [38, 46, 50]; even if the whole k8s cluster is reset, there is no metadata loss.

The *data plane* mainly contains the *kubelet*, *kube-proxy*, and *container runtime*. They run on every cluster node. The

kubelet consists of various managers for the whole lifecycle of Pods. It watches Pod object creation or deletion [64], and correspondingly runs or stops the Pod by communicating with the container runtime through the Container Runtime Interface (CRI) [29, 90], a plugin interface to interact with various container runtimes without recompilation. A CRI shim could exist between the kubelet and the container runtime, e.g., cri-containerd [73]. The kube-proxy is a network proxy managing network rules for communicating with Pods from both inside and outside the cluster.

2.3 Kubernetes Multi-Tenancy

Typically, k8s multi-tenancy is categorized into soft and hard multi-tenancy [94]. "Soft" means tenants may partially trust each other, e.g., multiple teams in a company. Namespace-based isolation provides such soft multi-tenancy [89, 94]. "Hard" means tenants distrust each other and require strong isolation; e.g., customers from very different backgrounds and each could be a potential attacker. Typically, *for soft multi-tenancy, multiple tenants share one control plane and one data plane.* Sharing a control plane restricts each tenant to use the cluster-wide API [114]. Sharing a data plane facilitates a malicious tenant to utilize vulnerabilities (e.g., container escape [57]) to compromise the OS kernel and affect other tenants. On the other hand, *hard multi-tenancy requires isolation for both control planes and data planes, i.e., each tenant owns a dedicated control plane and data plane (thus a dedicated cluster)* [89, 94]. Note that a dedicated plane means that an attacker cannot compromise others' planes by maliciously controlling the OS kernel within their own plane. It does not necessitate separate *physical* machines; instead, separate virtual machines are also suitable. Multi-cluster architectures satisfy the requirement [70], providing full functionality for each tenant. However, they suffer from resource underutilization and high management costs, hindering cloud providers from adopting hard multi-tenancy [3, 17, 36]. *Apart from the two models, virtual cluster architectures (e.g., Vcluster) isolate the control plane while sharing a data plane* [60, 71, 114]. Virtual clusters can be built on the cloud-native k8s in an add-on manner (Figure 1). A *Syncer* is used to synchronize workloads between the virtual cluster and the host cluster, and facilitates workload management by sharing the host scheduler and the host cluster data plane (where Pods physically run). Although virtual clusters offer full functionality, they suffer from weak isolation. In this paper, we aim to achieve hard multi-tenancy.

3 Challenges and Goals

Although hard multi-tenancy architectures provide robust isolation *among tenants*, safeguarding tenant clusters *against potentially malicious cloud providers* remains crucial. We can leverage VM-based TEEs (Section 2.1) without heavy modifications on the k8s software stack. Under our threat model (Section 3.1), there are naturally two extreme design principles: *monolithic protection* and *minimal protection*. For monolithic protection like Constellation [85], the whole k8s software stack is put into VM TEEs, turning the VM into a trusted node, and the communication between trusted nodes is protected by TLS [77]. For minimal protection like Confidential Containers (CoCo) [12, 48], only the Pods reside in VM TEEs. Unfortunately, both designs have key drawbacks (Sections 3.2 and 3.3), which motivate our design (Section 3.4).

3.1 Threat Model

We classify several roles in the k8s system as shown in Figure 1. A cloud provider serves as the host cluster administrator owning the host k8s cluster, and tenants operate on tenant clusters built atop it. Since a tenant may be a company, it can be divided into several teams. They differentiate each other through access control in the tenant cluster while they belong to the same tenant. Some of them are tenant cluster administrators, and others may be unprivileged developers. The former can set the permissions of others, define custom API resources, deploy new webhooks and custom controllers, and so on. The latter can deploy workload resources like Pods. We remark that the internal access control inside a tenant is inherited from k8s and not specific to Pyramid. Service Pods are exposed to users outside the cluster.

From a tenant's perspective, while its tenant cluster is trusted, the underlying host cluster (i.e., the cloud) and other tenants are untrusted. Thus the adversary can control the whole software stack of each server, including the host OS and the hypervisor. In that case, a tenant desires a k8s cluster protected from the cloud and other tenants (i.e., hard multi-tenancy). In this paper we focus on this interaction between the tenant and the cloud provider. We aim to achieve confidentiality, integrity, and data freshness for the workloads in the tenant's cluster. We assume the hardware processors on the host are trusted and equipped with VM-based TEEs such as AMD SEV. The code inside the TEEs can be verified through attestation to ensure integrity, while other code can be arbitrarily modified. The adversary can also snoop, tamper with, or roll back data outside the processors, including data in the memory, disks, and network devices.

From a service user's perspective, she trusts the tenant cluster (i.e., the service provider), and the workloads in the tenant cluster support establishing secure channels with the service user, who manages her own credentials like certificates and keys. The tenant Pod never actively leaks user data to the outside of TEEs.

Side-channel attacks [35, 37, 53, 65, 81, 102, 110, 112], TEE design defects [22, 40, 55, 109], denial of service (DoS), and internal security flaws of code inside TEEs are out of scope.

3.2 Monolithic Protection Drawbacks

To combine hard multi-tenancy architectures with VM TEEs, a reasonable monolithic scheme is to provide a vanilla k8s

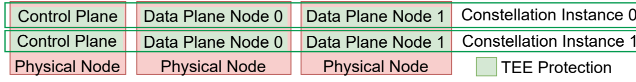


Figure 2. Constellation architecture.

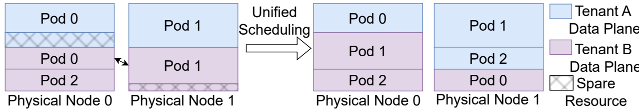


Figure 3. Benefits of unified resource management.

cluster encapsulated with VM TEEs for each tenant, which is the Constellation architecture shown in Figure 2. Recall that hard multi-tenancy does not necessitate that clusters from different tenants reside on separate *physical* machines. Constellation allows TEEs from multiple tenants to run on the same physical machines. Despite sharing physical machines, the issue of resource underutilization for cloud providers still persists. As shown in Figure 3 left, tenant A cannot deploy its Pod 2 because it can only schedule Pods based on its own cluster information, which indicates a lack of resources, though the resource margin of the entire host cluster is sufficient. To solve the issue, the cloud provider should be able to infiltrate the TEE of tenant B and reschedule the two Pods of tenant B across machines to make enough room for tenant A (Figure 3 right). Note that Constellation cluster scaling [86] cannot help, where tenant A just requests for more physical nodes and the cloud would reject it due to lack of resources. Instead, it is desirable to have unified resource management across tenants with a central point of control.

A straightforward solution for unified management is connecting different control planes to the same ground truth, i.e., etcd, which collects all information globally, including resource capacity and consumption from kubelets in all tenants' data planes and resource requests described in Pod specifications. Hence, the scheduler in each control plane can designate a Pod destination based on the information. However, this method has several drawbacks. First, multiple control planes may concurrently query the etcd for resource information, and individually schedule Pods based on the potentially stale information, causing scheduling failures. Second, access control would be difficult because one tenant is allowed to access the states of others. Third, as all control planes have equal rights, it is hard to coordinate and arbitrate among them, e.g., to determine whose Pod to be evicted when a host node is stressed. Finally, the fundamental mechanism for interaction between the control plane and the etcd is List-Watch [47, 83], where an object update in the etcd will inform all the related entities. As the number of tenants grows, the performance will suffer.

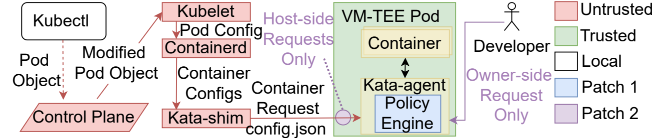


Figure 4. Pod creation flow in CoCo architecture.

3.3 Minimal Protection Drawbacks

One may think minimal protection is enough for security because it is the Pods that have sensitive data. However, the misalignment between the Pod protection boundary and the tenant cluster boundary leads to various attacks, since the complicated k8s framework outside Pods contains heavy code directly or indirectly controlling or viewing Pods' states.

As shown in Figure 4 for the CoCo architecture [12], the format of a Pod varies as the object flows through different k8s components. As the container runtime starts a Pod, it first launches a VM TEE and attests to the tenant to ensure the environment (e.g., the hardware and the OS) is correct to run the Pod. Then the guest VM should get the container configurations (`config.json`) from the untrusted host to start the Pod [12]. Because such configurations are not included in the attestation report, a privileged attacker can arbitrarily snoop and modify the configurations, leading to incorrect behaviors of the Pod, which is somewhat similar in spirit to Iago attacks [25]. Besides, kubectl [21] allows a tenant to execute any command in its running Pods. From the view of a Pod VM, the command comes from the untrusted host may be falsified. We elaborate on various Pod-level attacks in more details in Appendix B.

The CoCo community has recently proposed two patches (Figure 4). The first one adds a policy engine to validate the configuration against tenant-provided policies [48]. However, the policies may be intrinsically incomplete and inflexible for dynamic modifications potentially applied by the custom webhooks and controllers in the control plane, and it adds the burden of Pod protection to the tenants. More importantly, the scheme forbids some requests, significantly sacrificing functionality [101]. The second one categorizes the requests into host-side non-sensitive APIs and owner-side sensitive APIs [11] (published at [101] recently). For example, it classifies in-Pod command execution as owner APIs, and Pod creation and deletion as host APIs. However, the authors do not prove that the broad spectrum of host-side APIs cannot be potential attack points.

Neither patch fits in our setting (Section 3.1), where a "tenant" comprises different roles with different permitted actions, and the tenant administrator must perform maintenance and debug operations using the "host" APIs. Moreover, it is hard to protect Pod-to-Pod communication transparently as in Constellation [85]. As CoCo has been commercialized recently, four vulnerabilities are found [107], of which CVE-2024-21376 and CVE-2024-21400 are explicitly related to k8s

Table 1. Comparison of different secure k8s approaches.

System	Isolation	TCB	RM	Add-on	O&M
Constellation	Strong	Large	Separate	No	Easy
CoCo	Weak	Medium	Unified	Yes	Tricky
PYRAMID	Strong	Large	Unified	Yes	Easy

API manipulation from the control plane [75, 76]. We remark that these vulnerabilities stem from CoCo's exclusion of control plane code from TEEs rather than placing them in TEEs. This exclusion exposes APIs at the TEE boundary to untrusted callers (e.g., cloud attackers) instead of the tenant.

Besides, Pod-centric protection in CoCo *lacks protection for high-level workload resources*. Recall that a StatefulSet imposes a set of constraints on the orchestration of its Pods, which are guaranteed by k8s and utilized by tenants to deploy their Pods. If the k8s control plane is not protected, the Pods may not run correctly as expected by the tenant. For example, in normal execution, upon detecting the creation of a new StatefulSet, e.g., for a MySQL service, the StatefulSet controller would generate a set of Pods with the same image but different name suffixes, i.e., `0` to `N`. Based on the name suffix, either a leader or a follower configuration is copied into the Pod for MySQL to load the configuration. However, if the attacker maliciously launches two Pods with the same name suffixed by `0`, two leader Pods would run simultaneously. At this point, the critical issue is that data integrity is violated due to master conflict, rather than merely the integrity of the resource configuration being affected.

Nevertheless, we remark that CoCo excels in single Pod protection due to a relatively small TCB, and CoCo can reuse attestation reports for both tenants and users. However, it does not guarantee the cluster-level security.

3.4 Goals

Due to the intrinsic vulnerabilities of the minimal protection scheme, we choose to improve the monolithic scheme by resolving the dilemma of isolation and resource utilization. On the one hand, we aim to achieve cluster-level security. Pods should be correctly spawned, and connections among Pods should be protected. Confidentiality and integrity of tenant data and user data should be guaranteed. On the other hand, we should achieve unified resource management among multiple tenant clusters. Since k8s is a complex and product-level framework and has been widely used, we prefer to deploy our design in an add-on manner, without any modification to the source code of vanilla k8s to cooperate with ours.

To summarize, with the comparison to existing solutions shown in Table 1, we aim to build a pluggable secure k8s named PYRAMID, with the following features:

1. Strong isolation guarantees with moderate TCB;
2. Unified resource management (RM);
3. Easy deployment through an add-on manner;

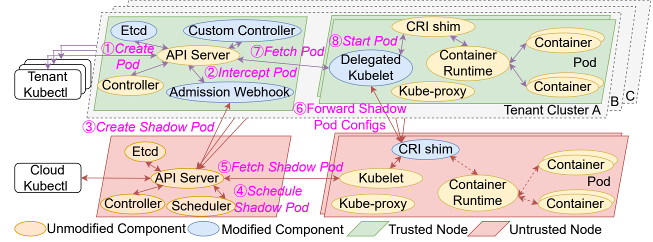


Figure 5. System architecture of PYRAMID.

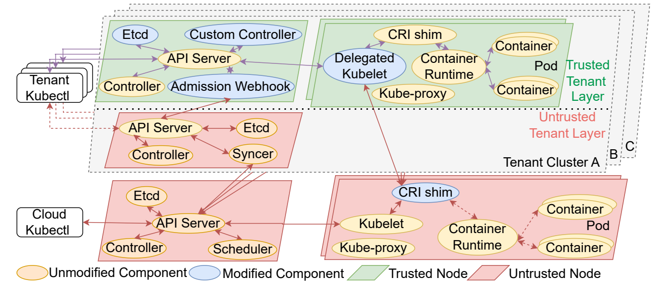


Figure 6. Vcluster-integrated PYRAMID.

4. Full-fledged functionality (including crash recovery) and easy operation & maintenance (O&M);
5. Minor performance overheads.

4 System Architecture

We illustrate our key design points in Section 4.1 and give a brief analysis in Section 4.2. Section 5 further describes the details of our key component designs.

4.1 Overview

To avoid the security issues in Section 3.3, we follow the principle that security fundamentally involves subtraction—minimizing attack surfaces—rather than addition [80]. We begin with the monolithic scheme (i.e., Constellation), where each Constellation instance is a (trusted) tenant cluster, and nodes from multiple tenant clusters can co-exist on the same physical nodes. However, each tenant cluster can only manage its own resources, mainly through two components (Section 2.2): the kubelet in the data plane allocates/deallocates resources on each node for its Pods and maintains their resource usage; the scheduler in the control plane dispatches Pods to nodes based on the resource requirement of Pods and the usage information gathered from all nodes.

To enable unified resource management, our key observation is that *the information on resource budget and resource consumption is generally not sensitive*. On the one hand, it is the cloud provider that decides the real resource allocation. On the other hand, the cloud provider always knows the resource consumption of the workloads running on it. Hence, both integrity and confidentiality about the resource information are meaningless. As a result, we introduce a host

cluster as an *untrusted layer*, on which each tenant cluster in the *trusted layer* is logically deployed. Each tenant cluster should be *deprived of the ability to independently manage resources*, and share the resource requirements of Pods, as well as the usage information, with the host cluster.

Concretely, each time the trusted-layer scheduler of a tenant schedules a trusted Pod, the untrusted-layer scheduler of the host cluster should intercept the procedure, assign a node for that Pod, and return the assignment result to the trusted layer. Then, as the trusted-layer kubelet intends to allocate resources for that Pod, the untrusted-layer kubelet should intercept the procedure and scale up the trusted node with sufficient resources for that Pod. Besides, the untrusted-layer kubelet should maintain the resource usage of trusted Pods for the untrusted-layer scheduler to make better decisions.

However, naively implementing the above design requires significant engineering efforts to modify the scheduler and kubelet, not satisfying the pluggable goal. Instead of intrusively modifying the untrusted k8s core components, PYRAMID *leverages the k8s extension points, which are designed to support various customizations (Section 2.2), and maximally reuses the workflow of the untrusted layer to make the two layers cooperate*, so that the complicated k8s workflow can be mostly preserved without significant changes, while still enabling unified resource management in the untrusted layer.

PYRAMID's architecture and Pod creation workflow. As shown in Figure 5, to deprive the tenant cluster of the ability to schedule Pods, PYRAMID removes the scheduler from the tenant cluster in the trusted layer. Hence, all tenant clusters rely on the scheduler in the untrusted layer to schedule. Concretely, the trusted-layer webhook (an extension point) intercepts the Pod creation request and yields a *shadow Pod* (Section 5.1) request to the untrusted layer. The shadow Pod is a sanitized Pod free of sensitive information in the trusted Pod, serving as a resource-consuming representative for the trusted Pod. After the untrusted-layer scheduler schedules the shadow Pod, the untrusted-layer kubelet fetches the shadow Pod object and allocates resources for it. Hence, the scheduling result and resources are ready for the corresponding trusted Pod. Next, a modified CRI shim (an extension point) attaches the resource to the tenant's trusted node on the same host node, and redirects the shadow Pod configurations (including the scheduling result) to the trusted-layer kubelet in the trusted node. The trusted-layer kubelet cannot start a Pod without the help of the untrusted-layer kubelet; thus we name it *delegated kubelet*. Note that the CRI shim identifies whether the configuration belongs to a shadow Pod and, if so, forwards it to the delegated kubelet of the tenant; otherwise, a normal (untrusted) Pod starts.

Given that the orchestration of trusted Pods involves the scheduler and the kubelet in the untrusted layer, the attacker can launch two Pods with the same identity for a StatefulSet, compromising execution integrity (Section 3.3). We propose *exclusionary guards* (Section 5.2) as a locking mechanism to ensure no security issues could take effect even in the case where the adversary can arbitrarily disconnect any server node and manipulate the clock of a node. As a result, before the delegated kubelet pulls the trusted Pod according to the configuration (mainly Pod `name` and `namespace`) from the untrusted CRI shim, it should try to create an exclusionary guard, and only continues to proceed if successful.

For secure crash-recovery of k8s, the ground truth of cluster states, etcd, should be protected, which increases the overheads. By observing that some k8s objects for operation and maintenance are not necessary to store in the trusted etcd, PYRAMID proposes separate etcds for data of different security levels to further improve performance, as an optimization named *object divergence* (Section 5.3).

Overall workflow. To bootstrap a trusted layer, a set of VM TEEs are launched for control and data planes, and attest to the tenant by providing the measurement of the trusted region, including the hardware environment, bootloaders, OS kernels, and k8s components [79]. After the tenant finishes attestation, it has established secure channels with all VM TEEs and then distributes certificates and keys to the trusted k8s components for inter-component TLS connections [77]. As the tenant cluster scales up, a new trusted node without resource reservation is created on a host node, followed by remote attestation. Recall that a tenant may be a company with administrators and developers (Section 3.1). After an administrator performs attestation, others (in the same company) do not need to install and verify the cluster, because they are the same tenant. For service users, as they trust the tenant (Section 3.1), no attestation is required for users. Instead, the tenant informs the users that the cluster attestation passes. Constellation and CoCo also perform attestation for tenants instead of users [48, 85]. If users does not trust the tenant, the case would be much more tricky. Although the attestation reports can be reused by the users as in CoCo, tenants can still manipulate in-Pod runtime states of users through `kubectl exec` [101]. Banning such commands sacrifices functionality. We leave it to future work.

Without loss of generality, consider the tenant deploys a StatefulSet object through kubectl. The request securely arrives at the trusted API server through the secure channel. Then the StatefulSet is stored in the etcd. After the trusted StatefulSet controller sees a new object, it generates its affiliated Pods, following the aforementioned Pod creation workflow. We remark that the trusted objects never leave the trusted layer and they only flow through secure channels between trusted k8s components, so their confidentiality, integrity, and freshness are naturally ensured.

StatefulSet deletion is relatively simple. As the webhook intercepts the Pod deletion request, it requests to delete the shadow Pod. Simultaneously, the delegated kubelet watches a Pod deletion and stops the trusted Pod. The exclusionary guard will be deleted only when the Pod is *logically* stopped (Section 5.2). Finally, the untrusted kubelet also watches a

shadow Pod deletion, and reclaims the resource. Note that the deletion procedure in the trusted layer does not rely on any components in the untrusted layer, so it is secure.

Vcluster integration. Besides directly on k8s, PYRAMID can also run on other multi-tenant architectures such as Vcluster (Section 2.3). As shown in Figure 6 (which combines Figure 5 and Figure 1), compared to PYRAMID on k8s, it introduces an additional *untrusted tenant cluster*. Importantly, this extension does not affect the security of the trusted tenant clusters on top of it (comparing Figure 5 and Figure 6), and thus the hard multi-tenancy isolation among trusted tenant clusters is the same as described before and remains valid. The untrusted tenant cluster in Vcluster merely affects how the untrusted-layer host cluster is implemented, which saves us a lot of engineering efforts (e.g., natively handling object name conflicts from different tenants) to support multi-tenancy. Note that the untrusted tenant layer does not possess a data plane, so the CRI shim is on the host data plane for Pod configuration forwarding.

4.2 Analysis

We defer the security and performance analysis to Section 6 and show that the other goals in Section 3.4 are achieved in this section. For unified resource management (**Goal 2**), PYRAMID shares a scheduler, which cooperates with the delegated kubelets across multiple clusters. Moreover, the layered architecture embodies the add-on feature and is compatible with multiple architectures (**Goal 3**). The PYRAMID implementation pays special attention to minimizing the required changes to the k8s core components. Figure 5 has highlighted the modified parts in PYRAMID in blue. The scheduler is shared between the two layers, and the removal of the scheduler in the trusted layer is done by simply terminating a running binary. Except for the delegated kubelet in the trusted layer, we implement all the other modified components by only utilizing the extension points of the k8s framework. We modify ~1200 lines in total, including ~750 for extension points. Besides, we do not sacrifice any functionalities, such as cluster-wide object management, `kubectl exec` and `kubectl logs`, owing to the whole protection of k8s, facilitating operation and maintenance (**Goal 4**).

5 Key Techniques

As mentioned in Section 4.1, PYRAMID leverages a novel abstraction called *shadow Pods* for unified resource management, together with *exclusionary guards* to defend against malicious orchestration. These mechanisms try to maximally utilize various extension points explicitly or implicitly provided by k8s, entirely avoiding modifying the k8s core components in the untrusted layer, and minimizing the intrusive modification of those in the trusted layer. We introduce them in this section, as well as the *object divergence* technique which further improves performance.

5.1 Shadow Pods

Listing 1. An example of a Pod object: nginx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-1 # Pod name
spec:
  containers:
    - name: hello # container name
      image: myimage # image to be pulled
      command: ["printenv"] # command to be executed as the container
           starts
      args: ["MY_POD_NAME"] # args for the command
      env: # accessible env vars in the container
        - name: MY_POD_NAME # env var name
          valueFrom: # value source of env var
            fieldRef: # downward API
              fieldPath: metadata.name # Pod field
        - name: secret
          value: "this is a secret!"
      volumeMounts: # where to mount volumes
        - name: foo # volume name
          mountPath: "/config" # within the image
          readOnly: true
  volumes:
    - name: foo # volume to be mounted
      configMap: # type of volume: configMap
        name: myconfigmap # dependent configMap
```

Listing 2. The shadow Pod corresponding to Listing 1

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-1 # nonsensitive
spec:
  containers:
    - name: dummy
      image: k8s.gcr.io-pause:3.1 # dummy image
```

It is the Pods that actually consume resources rather than the high-level workloads like a StatefulSet, which is eventually transformed into Pods. Pods are also the unit of scheduling. Therefore, PYRAMID does not shadow high-level objects, but only intercepts Pod requests and generates shadow Pods (Section 4). A shadow Pod is a desensitized resource-consuming representative managed in the untrusted layer for the corresponding Pod in the trusted layer.

Shadow Pods should ensure that (1) all private information is removed or replaced with insensitive data; and (2) no unexpected fault is triggered during the lifecycle of the shadow Pod (e.g., if the trusted Pod depends on other objects in the trusted layer and the shadow Pod preserves the dependency, a resource-not-found fault would occur since no such objects exist in the untrusted layer); and (3) shadow Pods themselves take minimal resources (e.g., if a trusted Pod requires an image and the image field is unchanged for the shadow Pod, unnecessary storage occupation would occur). Therefore, we first comprehensively analyze the fields of Pods to categorize them to apply different processing strategies. Moreover, we recognize the potentially referred objects

Table 2. Summary of rules to adjust fields during shadow Pod generation. Other fields not listed are kept.

Action	Field
Removed	.spec.<Container>.command/args/workingDir, .spec.<Container>.env/.envFrom, .spec.<Container>.lifecycle/securityContext, .spec.<Container>.stdin/stdinOnce/tty, .spec.serviceAccountName, .spec.automountServiceAccountToken, .spec.securityContext
Modified	.metadata, .spec.<Container>.name/image/ports, .spec.<Container>.volumeMounts/volumeDevices, .spec.volumes/<Scheduling>
Kept	.spec.<Container>.resources, etc.

that also need shadows for shadow Pods to run correctly. Finally, we try to avoid double resource consumption.

The fields in a Pod specification are extremely diverse [95]. They include information about containers, involved volumes, service accounts, hostname, etc. First, we observe that we can safely preserve most fields in the shadow Pods and these fields do not depend on other shadow objects. For example, .spec.<Container>.resources specifies the budget of CPU and memory, and we keep it for the untrusted kubelet to account for the resources taken by the trusted Pods. Second, some fields that depend on other shadow objects can be directly removed without causing faults (e.g., fields about configMaps and ServiceAccounts), because their relevant untrusted execution results are not needed by the trusted layer, i.e., the trusted layer generates and consumes the dependent objects within itself. Third, a limited number of fields need removal or replacement for confidentiality reasons and proper resource management. For example, the .spec.<Container>.env field containing the sensitive values of environment variables should be removed; the required .spec.<Container>.image field designating the image to be pulled should be replaced with a dummy one (e.g., k8s.gcr.io/pause). We also add key-value pairs in .metadata.annotations to indicate the Pod is a shadow Pod, thus the container runtime behind the CRI shim will not pull images in the untrusted layer. Finally, note that for volumes such as local [96], which does not store sensitive metadata as ConfigMap does, the untrusted layer should account for their consumed resources and hence the corresponding volumes should not be removed. Table 2 summarizes the field adjustment rules, which need updates as API changes. Note that shadow Pod generation (e.g., from Listing 1 to Listing 2) is automatically performed by a trusted webhook following the aforementioned rules.

We remark that the substitution and removal of fields in shadow Pods put some restrictions on the specification of the trusted Pod object. The tenant should pay attention to the cases where a field value depends on a value in the same Pod object (i.e., downward APIs) and is *consumed* before the workflow comes to the CRI shim; otherwise, it may crash the untrusted kubelet. For example, the untrusted kubelet may try to operate on a non-existent directory if .spec.volumes.hostPath.path depends on a field in the same object. We can add a webhook at the API server to check the tenants' trusted Pods to forbid this usage. Fortunately, for common downward API usage, the dependent values are consumed inside the (trusted) Pod instead of the untrusted kubelet, and before the consumption, values are resolved from scratch after the delegated kubelet fetches the Pod object. Therefore, the to-be-used values are independent of the wrong values resolved in the untrusted layer.

After the shadow Pod with properly set fields arrives at the untrusted kubelet, resources including CPUs, memory, and persistent storage are allocated, and the related configurations are passed to the CRI shim. Pyramid's custom CRI shim does not run a shadow Pod. Instead, it forwards the Pod configurations to the delegated kubelet in the trusted layer. Besides, it dynamically expands memory and virtual CPUs for the trusted node [58, 69, 113], and it also attaches the allocated resources, like persistent volumes, to the trusted node. We remark that the applications should handle the encryption of data stored in volumes, unless trusted storage hardware equipped with TEE-IO is used with VM TEEs (Section 2.1). This requirement also applies to CoCo [12] and Constellation [85]. When a node is under resource pressure, the untrusted kubelet can evict both untrusted Pods and trusted Pods via shadow Pods, and reclaim the resources. Apart from the call to run and stop a Pod, for other CRI invocations, the custom CRI shim always identifies requests related to the shadow Pod and acts as an interceptor.

5.2 Exclusionary Guards

From the received Pod configurations, the delegated kubelet in the trusted layer will mainly consume (1) the name and namespace fields indicating the Pod intended to launch, and (2) the path of allocated resources (e.g., volume). We have stated that resource-related information (2) is insensitive. As for (1), the untrusted layer may fool the delegated kubelet to run a wrong Pod. Given that only trusted objects flow in the trusted plane (Section 4.1), the delegated kubelet may mistakenly fetch an unintended Pod instead of a misconfigured Pod. Although a Pod object is uniquely stored in the etcd, the attacker can still spawn two Pods with the same name on different nodes since each delegated kubelet pulls the Pod object independently, compromising execution integrity (Section 3.3). Even though the high-level workload resources like StatefulSets and their controllers already impose a set of constraints for orchestration, the constraints are partially invalidated due to the untrusted scheduling.

Our key insight is that, the problem can be solved if a lock is held by the node during the Pod's whole life, preventing

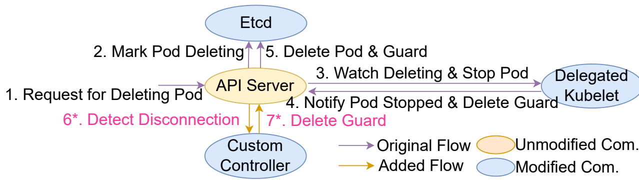


Figure 7. Pod deletion workflow. Operations marked with * are activated if the node is unreachable.

other nodes from acquiring the lock and creating another Pod for the same configuration. Two same Pods thus cannot remain alive simultaneously. The lock is released when the Pod is stopped and deleted. Note that before a Pod object is deleted, typically the Pod should have been already stopped, unless using forced termination that is not recommended [14]. However, a node may crash or be maliciously disconnected. The former case is common, and Pod recovery from crashed nodes requires the lock to have been previously released. In the latter case, the Pod still runs on the node. Since we cannot distinguish between the two situations, we cannot simply revoke the lock for the node; otherwise, the attacker may successfully launch two identical Pods. Note that Constellation is also vulnerable to this attack due to its inability to stop the unreachable Pods.

The k8s framework natively provides the *Lease* objects, which are originally designed for node heartbeats and leader election when multiple schedulers or controllers exist [93]. Utilizing leases with accurate clocks seems effective in solving the issue. Concretely, the lease for a Pod would be released after the underlying node is intentionally or accidentally disconnected for a certain period of time. However, since TEEs do not provide trusted clocks [8, 39][1], relying on timed leases may cause security issues [99]. Moreover, from a performance aspect, periodical renewing leases would heavily stress the etcd.

We propose *exclusionary guards*, locks without timed release. Hence, Pods do not need to periodically renew their guards. Pods with different specifications have different guards. Deletion of the Pod accompanies the release of the corresponding guard. For unreachable Pods (thus cannot be deleted), our key idea is that, as long as the unreachable Pod cannot affect other (trusted) Pods via shared storage and network, the Pod *logically* stops. The shared storage is typically cloud storage or a network file system (NFS), which is also accessed through the network. Therefore, *for Pods under control, just rotating their credentials (e.g., certificates) for network communication can achieve the goal.* Guard creation and normal deletion are performed by the delegated kubelet.

As the node becomes unreachable, including network partition or node failure, we observe that in k8s when the node controller detects it, it marks the node status as Unknown, and creates an Eviction object for the API server to evict the Pod. Therefore, Pyramid deploys a custom controller, which watches the corresponding Eviction object and the Unknown state of its underlying node. The controller can conclude that the node is unreachable and rotate the credentials for connected Pods belonging to the same high-level workload. For credential rotation, generally, current CNI plugins (e.g., cilium) support rotation [27], which involves restarting the connected Pods to activate new configurations, while old Pods hold the old credentials. Some cases are free of Pod restart. For example, by utilizing network policy [97], changes to some fields (e.g, ingress) can prevent access from old Pods. Once the rotation finishes, the controller releases the guards to launch fresh Pods. Figure 7 shows the workflow. The same applies if the attacker makes the node unhealthy (e.g., by blocking healthy checks) while keeping all other network traffic. If a single Pod is identified as disconnected while the underlying node behaves normally, the Pod can be finally stopped and deleted through the original workflow of k8s, because attackers cannot maliciously disconnect a Pod inside a node without crashing the node.

[1]Intel SGX previously provided trusted time, but deprecated it later for security reasons [39].

5.3 Security-Enhanced Etcd

Merely placing the etcd into TEEs does not guarantee security, as disk storage remains outside the protection scope [105]. While several distributed secure storage designs exist [18, 105], implementing a k8s-compatible trusted etcd using these approaches is complex. We design a trusted etcd following our main principle of no intrusive modifications to the original etcd. We leave dedicated etcd designs to future work.

We utilize the `dm-crypt` and `dm-integrity` features in the Linux kernel to transparently encrypt and authenticate persistent data. To guarantee data freshness, we leverage a trusted counter service [7, 66] to protect the backend store. In etcd v3, the backend store (the Raft state machine) contains an index tree that maps keys to corresponding value revisions, as well as a persistent store that maps each revision to the changes of key-value pairs in that revision. The Raft log, as the ground truth, is append-only and can be protected using a hash chain [42]. As a result, we find that if we bind the latest trusted counter with (1) the newest index of the committed log entry, (2) the tail of the hash chain (up to the last committed entry), and (3) the current revision before applying the committed log entries to the state machine, we can guarantee freshness. The modification of etcd only involves ~10 lines of code.

Accessing trusted etcd now becomes more expensive. We observe that not all API resources need to be stored in the trusted etcd. Examples include resource specification APIs like `ResourceQuotaSpec` (setting quota restrictions per namespace) and `PersistentVolumeClaim` (requesting for and claiming to a persistent volume), and operational and maintenance APIs like `Event` (recording cluster events in a best-effort manner) [92]. Therefore, we propose *object divergence*

to store these insensitive objects to another untrusted etcd for better performance, without relying on Pod developers to designate. We utilize the built-in API server configuration `--etcd-servers-overrides` to route different API resources to different etcd clusters. To avoid deploying another untrusted etcd cluster for the trusted layer, Pyramid reuses the etcd in the untrusted layer and avoids potential conflicts with untrusted objects by setting the `-etcd-prefix` string flag [91], which puts all insensitive API resources in the trusted layer on the directory with a prefixed path.

6 Evaluation

6.1 Experimental Setup

Platform. We use a cluster with 1 master node and 5 worker nodes connected with 25 Gbps network. Each supports AMD SEV-SNP [4] powered by an AMD EPYC 7713 processor of 3.67 GHz with 64 cores and 256 GB RAM. We establish external trusted and untrusted etcd clusters within VM TEEs and regular VMs, respectively. We use external etcd clusters by default. For the trusted counter service, we add a 20 ms delay during the data commit to mimic the cost to access it.

Workloads. K-Bench [111] is a specialized benchmarking tool for k8s. We use K-bench to generate Pod operations such as `create`, `delete`, and `list` to test the performance of the control plane. We vary the number of tenants in the K-Bench generator to stress the cluster. For data plane performance, we test several real-world applications including PHP-Apache (web servers), Redis (a distributed in-memory database), and Apache Kafka (a distributed event streaming platform) by deploying them as Pods and using their respective benchmark tools to obtain the maximum throughput.

Baselines. In addition to the k8s as the insecure baselines, our evaluation also includes the minimal protection scheme without security patches (i.e., CoCo [12]), and the monolithic protection scheme (i.e., Constellation [85]), as the secure baselines. Only Constellation approaches the security of Pyramid, although it is still a weaker scheme than Pyramid for lack of protection against malicious node disconnection (Section 5.2). For Pyramid, we evaluate both Pyramid on Vcluster (by default, denoted as Pyramid-V) and Pyramid directly on k8s (denoted as Pyramid-K). Besides, since the design of trusted etcd is an independent issue, we show the control-plane performance both when using the trivial method (just put etcd into trusted VMs, marked as "w/o secure etcd") and when using our enhanced design introduced in Section 5.3.

6.2 Control Plane Performance

For operations like `delete` and `list`, all systems (w/o secure etcd) perform similarly, including Pyramid, since Pyramid does not involve additional network round trips for such simple operations. We mainly focus on `create`, which is more representative. Figure 8a shows the latency increases

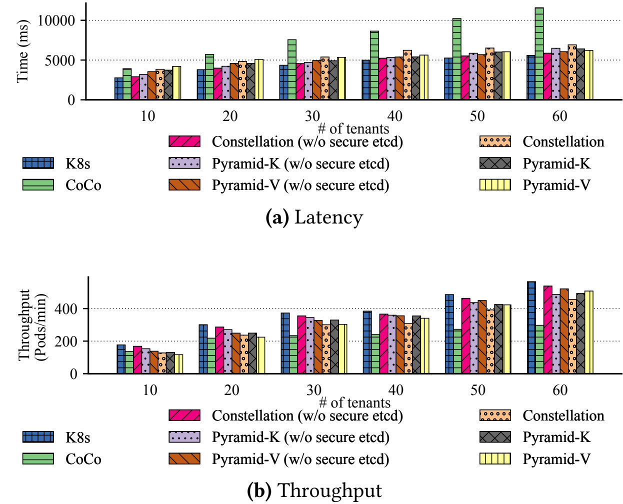


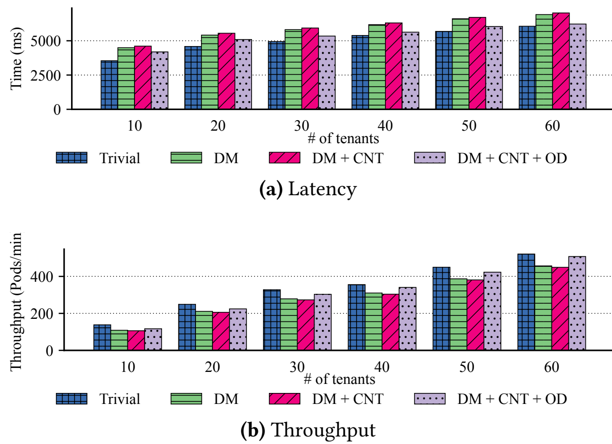
Figure 8. Overall performance for control planes.

as the workloads become heavier in all systems due to the increasing loads on the etcd.

If we do not use secure etcd, Constellation shows 5% additional cost over k8s on average, demonstrating the overheads of VM TEEs, while Pyramid-K and -V show 11% and 14% overheads, respectively. Compared to CoCo, Constellation, Pyramid-K and -V show speedups of 1.35× to 1.98×, 1.23× to 1.79×, and 1.10× to 1.91×, respectively. The overheads of CoCo mainly result from the VM TEE creation for each Pod. As more workloads stress the hypervisor, the speedup over CoCo becomes larger. CoCo uses the untrusted control plane directly and does not use secure etcd, which has security issues. If we further consider and evaluate CoCo with patches in Section 3.3, it would involve frequent interaction with remote clients during Pod creation, causing more cost than what we show here. Besides, the slowdown of Pyramid-K and -V over Constellation decreases from 1.10× and 1.22× to 1.02× and 1.03× as we add more tenants, because the additional network traffic overheads between untrusted and trusted layers for Pyramid-V are dwarfed by the overheads of etcd accesses, which both systems suffer from.

On the other hand, using secure etcd incurs additional slowdown of 1.20×, 1.05×, and 1.08× for Constellation, Pyramid-K, and -V, respectively. The smaller overheads for Pyramid-K and -V result from the object divergence optimization. Pyramid-K and -V achieve 1.08× and 1.02× speedups over Constellation. The difference lies in that an additional layer introduced by Vcluster incurs additional network round trips. Besides, Pyramid-K and -V cause 1.17× and 1.23× overheads on average compared to the vanilla k8s, while achieving 1.46× and 1.35× speedups than CoCo.

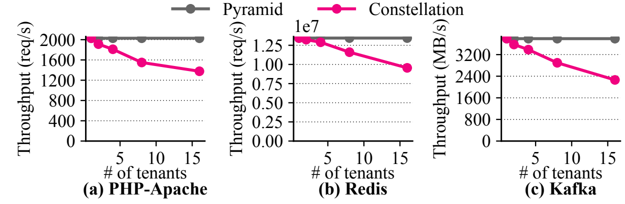
Looking at the latency breakdown, scheduling and Pod launching dominate the end-to-end control plane time for



(a) Latency

(b) Throughput

Figure 9. Comparison between different etcd techniques.



(a) PHP-Apache **(b) Redis** **(c) Kafka**

Figure 10. Data plane throughput for real-world workloads.

all systems, while their respective portions vary. For k8s, scheduling accounts for about 50%, while for CoCo, Constellation, Pyramid-K, and -V it is 33%, 51%, 58%, and 62%, respectively. Scheduling is dwarfed by Pod launching time in CoCo. For Pyramid-K and -V, scheduling involves communication among more planes, thus taking longer.

The throughput exhibits a similar trend as shown in Figure 8b. On average, the throughput of Pyramid-V is 1.03× faster and 1.09× slower than Constellation, with and without secure etcd, respectively. The throughput of CoCo is higher than Pyramid-V at first. However, as the workloads increase, the throughput of Pyramid-V overtakes that of CoCo, since Pod launching in CoCo is more expensive. Compared to insecure k8s, the overheads of Pyramid-V with secure etcd vary from 1.5× to 1.1× with the increasing workloads.

Effect of trusted etcd. We evaluate Pyramid with trivial etcd, etcd enabling `dm-crypt` and `dm-integrity` (`DM`), etcd with freshness guarantee using counter services (`CNT`), and optimization of object divergence (`OD`), as shown in Figure 9. `DM` brings the dominant overheads to secure etcd, ranging from 1.27× to 1.14× as the number of tenants increases. It demonstrates the extreme case of trading performance for transparent encryption on persistent data. Besides, the addition of counter service (`CNT`) only occurs 2% overheads on average. Object divergence (`OD`) brings 1.1× speedup since it redirects the non-sensitive accesses targeting a trusted etcd to an untrusted etcd, reducing the cost.

6.3 Data Plane Performance

The moderate control plane overheads presented in previous subsections mainly trade for (1) non-intrusive modification and pluggable implementation, and (2) better resource utilization for higher data plane efficiency. Here we show the second benefit. We mainly compare Pyramid-V with Constellation because Pyramid-K, k8s, and CoCo naturally unify the resource management and should be the same as Pyramid-V in terms of resource fragmentation. In a single-tenant setting, the tenant occupies the whole cluster, and Pyramid and Constellation should have similar resource fragmentation. When there are n tenants, one tenant occupies a portion of the cluster. Cloud providers usually sell virtual server instances with different capacities that are formed by splitting the physical nodes. We assume that each of the n tenants requests 5 virtual nodes, each with $1/n$ capacity of a physical node. For Constellation, Pods are scheduled separately in each tenant cluster. We deploy real-world applications individually using their benchmark tools to saturate the cluster. Then the throughput on all tenant clusters is summed and compared with the throughput in Pyramid.

We study PHP-Apache, Redis, and Kafka, which mainly consume CPU resources, memory resources, and both, respectively. As shown in Figure 10, with the tenants increasing from 2 to 16, the throughput improvement of Pyramid over Constellation varies from 1.06×, 1.02×, and 1.07×, to 1.47×, 1.41×, and 1.67×, for PHP-Apache, Redis, and Kafka, respectively. The less resource fragmentation, the more Pods can be deployed, and the higher throughput for Pods' clients.

6.4 Resource Utilization at Large Scales

To model the real-world scenario where there is a large-scale and heterogeneous k8s cluster holding mixed applications, we use a real-world dataset and the state-of-the-art open-source scheduler simulator [108] to evaluate the resource utilization for Pyramid and Constellation. Specifically, the dataset contains thousands of Pods with different resource requirements and around 1000 nodes with different resource capacities. The Pods are categorized into long-running (e.g., services) and short-lived Pods (e.g., computing tasks). Because the nodes have different capacities, the evaluation methodology is as follows. For each evaluation point, each cluster is initialized with no nodes, and the same set of Pods are evenly distributed to all tenants. As a Pod is requested, we allocate a best-fit node for it. If the Pod can fit in an existing node, no additional node is allocated. For long-running Pods, we quantify the scheduling capability by considering the ratio of the total amount of resources occupied by the Pods to the total capacity of allocated nodes. The larger the ratio, the less waste there is for nodes. On the other hand, for short-lived Pods, the ratio fluctuates over time and becomes

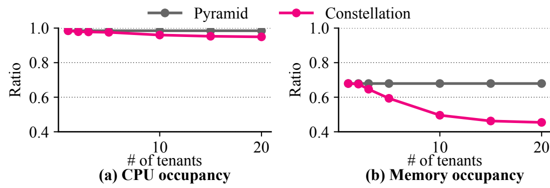


Figure 11. Resource utilization for long-running Pods.

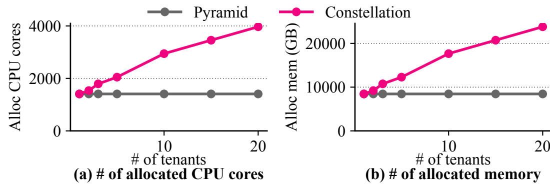


Figure 12. Allocated resources for short-lived Pods.

less meaningful. Therefore, we mainly count the amount of allocated resources to quantify resource utilization. The fewer allocated resources, the better.

The resource utilization for long-running Pods is shown in Figure 11. As the number of tenants increases, the saving of CPU cores is amplified to 3.5%, and that of memory reaches 22.5%. For short-lived Pods, as shown in Figure 12, if there are 20 tenants, the number of allocated CPU cores is reduced from 3968 to 1408, and the allocated memory is diminished from 23 808 GB to 8448 GB, by a significant factor of 2.8×.

6.5 Security Analysis

We place everything excluding the scheduler into TEEs and restrict the interface from the untrusted CRI shim to the delegated kubelet (Section 5.1). All the metadata in the trusted layer are stored in the trusted etcd, which is protected by TEEs and trusted counter services. As the metadata leave the TEE memory, they are transparently encrypted and authenticated. The metadata (e.g., trusted Pod objects) only flow inside the trusted layer. Although the scheduler is removed to ensure all tenants rely on the centralized host scheduler for scheduling, it does not lead to the metadata flowing to the untrusted layer. Instead, shadow Pods are generated inside the trusted layer and flow to the untrusted layer, with sensitive fields removed. Therefore, attackers cannot compromise the integrity of trusted Pods by distorting the remaining fields in shadow Pods, given that the trusted layer spawns Pods according to the specification stored in the trusted etcd, not relying on the shadow Pod's specification.

To automatically protect tenant and user data inside Pods, as data leave TEEs through I/O, they are transparently encrypted and authenticated for both storage and network communication. Besides, object divergence only exposes resource information and events. Events are sanitized, and the k8s functionalities in the trusted layer do not rely on the integrity of events.

Remaining attacks only include maliciously running or stopping a Pod through the untrusted CRI shim, as well as network manipulation. We rely on the technique of exclusionary guards (Section 5.2) to ensure that the attackers can only maliciously delete Pods or drop (encrypted) network packets, which is equivalent to accidental Pod crashes that must already be considered by distributed applications and can be addressed by consensus protocols (e.g., Paxos [52] and Raft [72]). Thus Pyramid provides the same security as normal distributed systems. Besides, for vulnerabilities explicitly related to k8s API manipulation [75, 76], Pyramid solves them since the control plane is fully protected.

7 Related Work

There has been some previous work using TEEs to protect k8s. As illustrated in Section 3, Constellation [85] is a monolithic protection scheme that puts all k8s components into TEEs, while CoCo [12] only protects Pods using TEEs. Both Constellation and CoCo are based on VM TEEs. However, some designs also combine process-based TEEs like Intel SGX (Section 2.1) with k8s. MarbleRun [34] can be installed on k8s clusters supporting SGX. It is used to orchestrate enclaves and build service-to-service communication among enclaves. Enclave-cc [28] acts as a container runtime (similar to CoCo), but the containers are protected by SGX. SCONE [82] and Vaucher et al. [103] also use SGX to protect containers.

As for k8s multi-tenancy, there are mainly three types of isolation: multi-cluster, virtual control plane, and namespace. Multi-cluster is a straightforward way to isolate tenants. As for virtual-control-plane-based methods, apart from Vcluster [60] and virtual cluster [114], Kamaji [16] also provides dedicated control planes to tenants and runs control plane components inside Pods on the supercluster. Official hierarchical namespace (HNC) [98] extends the original k8s namespace to support further dividing a namespace into sub-namespaces. Capsule [87] can union multiple k8s namespaces as a group to a tenant. Kiosk [59] also utilizes the k8s namespace and is pluggable to any existing k8s cluster.

8 Conclusions

We propose Pyramid, a novel container orchestration system that integrates VM-based TEEs into the k8s framework, in order to provide both security for private data processing, and flexible and efficient resource management across multiple tenants. The key design principle of Pyramid is to minimize intrusive changes to the highly complicated k8s code base and workflow. We therefore introduce a separate trusted k8s on top of the original untrusted k8s cluster and make them cooperate on resource allocation. We demonstrate two implementations of Pyramid: Pyramid-K directly running on k8s, and Pyramid-V integrated with Vcluster. We compare them against both the original insecure k8s system and the

two straightforward design principles of monolithic and minimal. Our evaluation shows that PYRAMID can achieve 1.4× speedup on the data plane, with comparable control-plane performance to the two secure baselines.

Acknowledgments

The authors would like to thank our shepherd, Landon Cox, and the anonymous reviewers, for their valuable suggestions, as well as the Tsinghua IDEAL group members for constructive discussion. Weijie Liu is supported by the National Natural Science Foundation of China under Grant No.62502237.

References

[1] Shai Almog. 2023. Debugging Kubernetes. In *Practical Debugging at Scale: Cloud Native Debugging in Kubernetes and Production*. 119–134.

[2] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security. *White paper* (2004).

[3] Amazon. 2024. *Hard Multi-Tenancy*. Retrieved September 29, 2024 from https://aws.github.io/aws-eks-best-practices/security/docs/multitenancy/#hard-multi-tenancy

[4] AMD. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Retrieved July 2, 2022 from https://www.amd.com/system/files/TechDocs/SEV-SNPstrengthening-vm-isolation-with-integrity-protection-and-more.pdf

[5] AMD. 2023. *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization*. Retrieved December 9, 2023 from https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[7] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation*. 193–208.

[8] Fatima M. Anwar and Mani Srivastava. 2019. Applications and Challenges in Securing Time. In *12th USENIX Workshop on Cyber Security Experimentation and Test*.

[9] ARM. 2021. *Arm Confidential Compute Architecture*. Retrieved July 15, 2023 from https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*. 689–703.

[11] The CoCo Authors. 2023. *Securing the Kata Control Plane*. Retrieved September 24, 2023 from https://github.com/confidential-containers/confidential-containers/issues/53

[12] The Confidential Container Authors. 2023. *Github Homepage: Confidential Containers*. Retrieved September 24, 2023 from https://github.com/confidential-containers

[13] The Kubernetes Authors. 2023. *Kubernetes Documents: Custom Resources*. Retrieved September 24, 2023 from https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources

[14] The Kubernetes Authors. 2023. *Kubernetes Documents: Pod Lifecycle*. Retrieved September 24, 2023 from https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle

[15] The Kubernetes authors. 2023. *Kubernetes Homepage*. Retrieved September 22, 2023 from http://kubernetes.io

[16] Authors of Kamaji. 2023. *Kamaji*. Retrieved October 15, 2023 from https://github.com/clastix/kamaji

[17] Azure. 2024. *Multi-Tenancy Types*. Retrieved September 29, 2024 from https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/aks#multitenancy-types

[18] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, Pramod Bhatotia, et al. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference*. 65–79.

[19] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and A. Shulman-Peleg. 2016. Secure Yet Usable: Protecting Servers and Linux Containers. *IBM Journal of Research and Development* 60, 4 (2016), 12:1–12:10.

[20] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems* 33, 3 (2015), 1–26.

[21] Steve Buchanan, Janaka Rangama, Ned Bellavance, Steve Buchanan, Janaka Rangama, and Ned Bellavance. 2020. Kubectl Overview. *Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration* (2020), 51–62.

[22] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. 2017. Fault Attacks on Encrypted General Purpose Compute Platforms. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 197–204.

[23] Carmen Carrión. 2022. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. *ACM Computing Surveys* 55, 7 (2022).

[24] Emiliano Casalicchio. 2019. Container Orchestration: A Survey. *Systems Modeling: Methodologies and Tools* (2019), 221–235.

[25] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API Is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 253–264.

[26] Sunyanan Choochotkaew, Tatsuhiro Chiba, Scott Trent, Takeshi Yoshimura, and Marcelo Amaral. 2022. AutoDECK: Automated Declarative Performance Evaluation and Tuning Framework on Kubernetes. In *2022 IEEE 15th International Conference on Cloud Computing*. 309–314.

[27] Cilium Authors. 2024. *IPsec Transparent Encryption*. Retrieved October 1, 2024 from https://docs.cilium.io/en/stable/security/network/encryption-ipsec/

[28] Confidential Containers Community. 2023. *Enclave-cc*. Retrieved October 15, 2023 from https://github.com/confidential-containers/enclave-cc

[29] Ramzi Debab and Walid Khaled Hidouci. 2021. Containers Runtimes War: A Comparative Study. In *Proceedings of the Future Technologies Conference 2020, Volume 2*. 135–161.

[30] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. 2022. StrongBox: A GPU TEE on Arm Endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 769–783.

[31] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Michael O'Connor, Rob Nertney, and Phil Rogers. 2023. Creating the First Confidential GPUs: The Team at NVIDIA Brings Confidentiality and Integrity to User Code and Data for Accelerated Computing. *Queue* 21, 4 (2023), 68–93.

[32] Ruxiao Duan, Fan Zhang, and Samee U Khan. 2021. A Case Study on Five Maturity Levels of A Kubernetes Operator. In *IEEE Cloud Summit*. 1–6.

[33] Paul DuBois and Michael Widenius. 2000. *MySQL*. New Riders Indianapolis, IN.

[34] Edgeless Systems. 2023. *MarbleRun: The Easiest Way to Orchestrate Enclaves*. Retrieved October 15, 2023 from https://www.edgeless.

systems/products/marblerun/
[35] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the Cryptographic Hardware and Embedded Systems*. 251–261.
[36] Google. 2024. *Multi-Tenancy Overview*. Retrieved September 29, 2024 from https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview
[37] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.
[38] Hima Govind and Horacio González-Vélez. 2021. Benchmarking Serverless Workloads on Kubernetes. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. 704–712.
[39] Gilang Mentari Hamidy, Pieter Philippaerts, and Wouter Joosen. 2023. T3E: A Practical Solution to Trusted Time in Secure Enclaves. In *International Conference on Network and System Security*. 305–326.
[40] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 129–142.
[41] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.
[42] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. 2005. Efficient Constructions for One-Way Hash Chains. In *Applied Cryptography and Network Security*. Springer, 423–441.
[43] Intel. 2018. *Intel Software Guard Extensions (Intel SGX) Developer Guide*. Retrieved June 30, 2022 from https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html
[44] Intel. 2021. *Intel Trust Domain Extensions*. Retrieved July 2, 2022 from https://cdrdv2.intel.com/v1/dl/getContent/690419
[45] Intel. 2023. *Intel TDX Connect TEE-IO Device Guide*. Retrieved December 9, 2023 from https://cdrdv2-public.intel.com/772642/whitepaper-tee-io-device-guide-v0-6-5.pdf
[46] Hung-Chin Jang and Shih-Yu Luo. 2023. Enhancing Node Fault Tolerance through High-Availability Clusters in Kubernetes. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data*. 30–35.
[47] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. 7–12.
[48] Matthew A. Johnson, Stavros Volos, Ken Gordon, Sean T. Allen, Christoph M. Wintersteiger, Sylvan Clebsch, John Starks, and Manuel Costa. 2023. Parma: Confidential Containers via Attested Execution Policies. arXiv:2302.03976 [cs.CR]
[49] David Kaplan. 2017. Protecting VM Register State with SEV-ES. *White paper, Feb* (2017).
[50] Ali Akbar Khatami, Yudha Purwanto, and Muhammad Faris Ruriawan. 2020. High Availability Storage Server with Kubernetes. In *2020 International Conference on Information Technology Systems and Innovation*. IEEE, 74–78.
[51] Marc Lacoste and Vincent Lefebvre. 2023. Trusted Execution Environments for Telecoms: Strengths, Weaknesses, Opportunities, and Threats. *IEEE Security & Privacy* 21, 3 (2023), 37–46.
[52] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.
[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium*. 557–574.
[54] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *2023 USENIX Annual Technical Conference*. 1–15.
[55] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2937–2950.
[56] Xiang Li, Fabing Li, and Mingyu Gao. 2023. Flare: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1439–1452.
[57] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 418–429.
[58] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. 2014. Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2014), 1350–1363.
[59] Loft Labs. 2023. *Kiosk*. Retrieved October 15, 2023 from https://github.com/loft-sh/kiosk
[60] Loft Labs. 2023. *Vcluster Homepage*. Retrieved September 24, 2023 from https://www.vcluster.com
[61] Sergio López-Huguet, J Damià Segrelles, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. 2020. Seamlessly Managing HPC Workloads through Kubernetes. In *International Conference on High Performance Computing*. 310–320.
[62] Haohui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. 2023. Honeycomb: Secure and Efficient GPU Executions via Static Validation. In *17th USENIX Symposium on Operating Systems Design and Implementation*. 155–172.
[63] Philippe Martin and Philippe Martin. 2021. Observability. 175–183.
[64] Philippe Martin and Philippe Martin. 2021. Scheduling Pods. *Kubernetes: Preparing for the CKA and CKAD Certifications* (2021), 89–100.
[65] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-Core Platforms. In *Proceedings of the 24th USENIX Security Symposium*. 865–880.
[66] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Security Symposium*. 1289–1306.
[67] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.
[68] Fernando Harald Barreiro Megino, Jeffrey Ryan Albert, Frank Berghaus, Kaushik De, FaHui Lin, Danika MacDonell, Tadashi Maeno, Ricardo Brito Da Rocha, Rolf Seuster, Ryan Paul Taylor, et al. 2020. Using Kubernetes as an ATLAS Computing Site. In *EPJ Web of Conferences*, Vol. 245. 07025.
[69] Germán Moltó, Miguel Caballer, Eloy Romero, and Carlos De Alfonso. 2013. Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements. *Procedia Computer Science* 18 (2013), 159–168.
[70] Multicluster Special Interest Group. 2023. *Github Pages about Kubernetes Multi-Cluster*. Retrieved September 24, 2023 from https://github.com/kubernetes/community/tree/master/sig-multicluster
[71] Nguyen Thanh Nguyen and Younghan Kim. 2022. A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes

Cluster. In *2022 27th Asia Pacific Conference on Communications*. 651–654.

[72] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*. 305–319.

[73] Nigel Poulton. 2023. *The Kubernetes Book*. NIGEL POULTON LTD.

[74] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. (2019). arXiv:1908.11143 [cs.CR]

[75] Recorded Future, Inc. 2024. *CVE-2024-21376*. Retrieved September 29, 2024 from https://www.recordedfuture.com/vulnerability-database/CVE-2024-21376CVE-2024-21376

[76] Recorded Future, Inc. 2024. *CVE-2024-21400*. Retrieved September 29, 2024 from https://www.recordedfuture.com/vulnerability-database/CVE-2024-21400CVE-2024-21400

[77] Eric Rescorla. 2000. *HTTP over TLS*. Technical Report.

[78] Leonard Richardson and Sam Ruby. 2008. *RESTful Web Services*. O'Reilly Media, Inc.

[79] Ravi Sahita, Dror Caspi, Barry Huntley, Vincent Scarlata, Baruch Chaikin, Siddhartha Chhabra, Arie Aharon, and Ido Ouziel. 2021. Security Analysis of Confidential-Compute Instruction Set Architecture for Virtualized Workloads. In *2021 International Symposium on Secure and Private Execution Environment Design*. 121–131.

[80] Jerome H Saltzer and Michael D Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[81] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 3–24.

[82] Scontain. 2024. *Scone and Kubernetes*. Retrieved February 7, 2024 from https://sconedocs.github.io/k8s_concepts

[83] Chenggang Shan, Yuanqing Xia, Yufeng Zhan, and Jinhui Zhang. 2023. KubeAdaptor: A Docking Framework for Workflow Containerization on Kubernetes. *Future Generation Computer Systems* 148 (2023), 584–599.

[84] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[85] Edgeless Systems. 2023. *Constellation: The World's Most Secure Kubernetes*. Retrieved September 24, 2023 from https://docs.edgeless.systems/constellation

[86] Edgeless Systems. 2024. *Scale Your Cluster*. Retrieved September 29, 2024 from https://docs.edgeless.systems/constellation/workflows/scale

[87] The Capsule Community. 2023. *Capsule*. Retrieved October 15, 2023 from https://github.com/projectcapsule/capsule

[88] The etcd Authors. 2023. *Etcd: A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System*. Retrieved September 24, 2023 from https://etcd.io

[89] The Kubernetes Authors. 2021. *Three Tenancy Models For Kubernetes*. Retrieved Match 3, 2025 from https://kubernetes.io/blog/2021/04/15/three-tenancy-models-for-kubernetes/

[90] The Kubernetes Authors. 2023. *Kubernetes Documents: Container Runtime Interface*. Retrieved September 24, 2023 from https://kubernetes.io/docs/concepts/architecture/cri/

[91] The Kubernetes Authors. 2023. *Kubernetes Documents: kube-apiserver*. Retrieved October 8, 2023 from https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[92] The Kubernetes Authors. 2023. *Kubernetes Documents: Kubernetes API*. Retrieved October 8, 2023 from https://kubernetes.io/docs/reference/kubernetes-api/

[93] The Kubernetes Authors. 2023. *Kubernetes Documents: Leases*. Retrieved October 5, 2023 from https://kubernetes.io/docs/concepts/architecture/leases/

[94] The Kubernetes Authors. 2023. *Kubernetes Documents: Multi-Tenancy*. Retrieved September 24, 2023 from https://kubernetes.io/docs/concepts/security/multi-tenancy

[95] The Kubernetes Authors. 2023. *Kubernetes Documents: Pod API*. Retrieved October 5, 2023 from https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/

[96] The Kubernetes Authors. 2023. *Kubernetes Documents: Volumes*. Retrieved October 8, 2023 from https://kubernetes.io/docs/concepts/storage/volumes

[97] The Kubernetes Authors. 2024. *Network Policies*. Retrieved October 1, 2024 from https://kubernetes.io/docs/concepts/services-networking/network-policies/

[98] The Kubernetes Community. 2023. *The Hierarchical Namespace Controller (HNC)*. Retrieved October 15, 2023 from https://github.com/kubernetes-sigs/hierarchical-namespaces

[99] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: A Trusted Lease Primitive for Distributed Systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 387–400.

[100] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 645–658.

[101] Ray Valdez, Md Salman Ahmed, Zhongshu Gu, Christophe De Dinechin, Pau-Chen Cheng, and Hani Jamjoom. 2024. Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers. In *ACM Annual Conference on Computer and Communications Security*.

[102] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium*. 1041–1056.

[103] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. 2018. SGX-Aware Container Orchestration for Heterogeneous Clusters. In *2018 IEEE 38th International Conference on Distributed Computing Systems*. 730–741.

[104] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2019. Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes. In *IEEE 19th International Conference on Software Quality, Reliability and Security*. IEEE, 176–185.

[105] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. 2022. Engraft: Enclave-Guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2841–2855.

[106] Xuhao Wang, Qingni Shen, Wu Luo, and Pengfei Wu. 2020. RSDS: Getting System Call Whitelist for Container Through Dynamic and Static Analysis. In *2020 IEEE 13th International Conference on Cloud Computing*. 600–608.

[107] Stack Watch. 2024. *Recent Microsoft Azure Kubernetes Service Security Advisories*. Retrieved September 29, 2024 from https://stack.watch/product/microsoft/azure-kubernetes-service/

[108] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference*. 995–1008.

[109] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. Sevurity: No Security without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy*. 1483–1496.

[110] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 640–656.

[111] Yong Li, Helen Liu. 2023. *K-Bench*. Retrieved October 15, 2023 from https://github.com/vmware-tanzu/k-bench

[112] Mark Zhao and G. Edward Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. 229–244.

[113] Weiming Zhao, Zhenlin Wang, and Yingwei Luo. 2009. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 21–30.

[114] Chao Zheng, Qinghui Zhuang, and Fei Guo. 2021. A Multi-Tenant Framework for Cloud Container Services. In *2021 IEEE 41st International Conference on Distributed Computing Systems*. 359–369.

A High-Level API Resources in Kubernetes

We can type `kubectl apply -f nginx.yaml`, with the file shown in Listing 3, to create a Deployment object. The field `.spec.replicas` indicates 5 nginx Pods to be created, and k8s automatically detects the liveness of these Pods. If any of them crashes, a new one is created by k8s to maintain the desired amount of running Pods. To update all the 5 nginx Pods to use a new image, e.g., version 1.25.2, we can type `kubectl set image deployment.v1.apps/nginxd nginx=nginx:1.25.2`. Likewise, a single command `kubectl scale deployment/nginxd -replicas=6` scales the Deployment to 6 Pods.

Listing 3. An example of a Deployment object: nginx.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginxd # object name
spec:
  replicas: 5 # number of Pods
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      containers: # this Pod runs one container
      - name: nginx # container name
        image: nginx:1.24.2 # Docker Hub image
        ports:
        - containerPort: 80 # exposed port
```

While Deployments are suitable for stateless workloads, StatefulSets are for stateful workloads like MySQL [32, 33], which require persistent storage [104]. The MySQL service has a leader-follower topology, where one leader handles both read and write requests, and the followers only handle reads. Deploying it using a StatefulSet naturally follows the topology because the StatefulSet maintains a sticky identity for each of its Pods, from 0 to the number of replicas, and each has its own storage volume. Different roles (leader or follower) can be assigned to the Pods according to their identities (Section 3.3).

Apart from the up-and-running service Pods deployed through Deployments and StatefulSets, task Pods that run to the end can also be deployed in k8s through a high-level workload abstraction named a Job. For example, AI/HPC workloads can run on k8s using Jobs [26, 61, 68]. The job can ensure that all Pods successfully complete.

B Pod-Level Attacks on CoCo

Lack of protection on Pod objects. Obviously, the Pod object, either in the format of `config.json` processed by the low-level container runtime, or in the format of the internal object processed by k8s core components like the API server, is controlled by the untrusted host. Therefore, the attacker can tamper with and snoop the Pod metadata. For example in Listing 1, the attacker can trivially get the value of the environment variable named `secret`. Moreover, it can replace the command to be executed as the container starts with an arbitrary malicious command. Given that downward API is provided to expose Pod information to its containers to use, for example, applications in container `hello` can use the Pod name `hello-1` through the `MY_POD_NAME` environment variable, all fields specified in the YAML file should keep their integrity while some may also need encryption.

Lack of protection on referenced resource objects. The Pod metadata can not only be directly specified in the Pod object but also determined by resolving the referenced object. For example, supposing that a configMap named `myconfigmap` contains some key-value pairs, the container `hello` can use those pairs by mounting the configMap at the path `/config` inside the container. Therefore, the referenced object should also be protected.

Lack of protection on Pod lifecycle. In the minimal protection scheme, we notice that the untrusted host manages the Pod lifecycle by sending the command into the VM, including the start and stop of its containers. Considering a Pod only owns one container and the container may write to persistent storage, an attacker can maliciously spawn two identical containers in the Pod, thus sharing the storage and causing incorrect results due to unexpected concurrency issues. Besides, kubectl [21] allows a tenant to execute any command in its running Pods. From the view of Pod VMs, the command comes from the untrusted host and may be falsified. It is undesired to simply disable the functionality since it is commonly used in operation and maintenance [1, 63]. One may consider a whitelist allowing certain types of commands to execute, which needs a careful tradeoff between security and functionality [19, 106]. One may also consider using a secure channel for the tenant Pod to deliver commands. However, k8s components like kubelet also need to send commands to containers (e.g., container probes [14]). The case is intractable since k8s components are not in protection.