Towards Secure Cloud-native Infrastructure: An Assessment of Containers, Container Orchestration Systems, Challenges and Opportunities

Benjamin Wagrez, *Member, IEEE,* and Deepak Nadig, *Member, IEEE*

***Abstract*—This paper presents a comprehensive review and security assessment of containers and container orchestration systems for securing cloud-native infrastructures. We highlight best practices for containerization within container orchestration systems such as Kubernetes. Containerization is a lightweight virtualization technology that presents a user-mode execution environment while sharing the host kernel at the host level [1]. Our assessment encompasses various security domains, including authentication and authorization strategies, namespace and resource management, Kubernetes security policies, network security, and vulnerability scanning. Our literature review provides robust analysis, surveying current research that explores security and security tools related to containerization and Kubernetes. Additionally, we provide detailed guidelines for deployment strategies, policies, networking, resource management, and the underlying technologies relevant to containerization. We present an evaluation process by assessing open-source security tools based on several criteria, including customizability, ease of use, effectiveness, granularity, scalability, community adoption, and support models. The resulting assessment offers recommendations for security tools within each domain, enabling practitioners to make informed decisions in securing their containerized environments. Furthermore, this work emphasizes the importance of adhering to security best practices when implementing containerization in Kubernetes. It provides proactive measures and guidelines to enhance security, focusing on Docker due to its popularity as a preferred containerization technology. Our findings and insights from this study contribute to securing containerized deployments in Kubernetes. Through literature review, security tools evaluations, and best practice recommendations, this work aims to assist practitioners in achieving robust security for their containerized deployments in Kubernetes environments.**

***Index Terms*—Containers, Container Orchestration Systems, Cloud-native Infrastructure, Kubernetes, Container Security.**

I. Introduction

CONTAINERIZATION is a lightweight virtualization technology that presents a user-mode execution environment while sharing the host kernel at the host level [1]. Emerging as an alternative to heavier virtual machine technologies, resource usage and start-up times are effectively reduced by lightweight containers that share the host's operating system (OS) kernel. However, the increased isolation of containers is consequentially paired with a new attack surface, establishing a need for containerized infrastructure security. This paper will study container security from a container orchestrator's perspective.

B. Wagrez and D. Nadig are with the Department of Computer and Information Technology, Purdue University, West Lafayette, IN, 47907 USA.
Corresponding author: Deepak Nadig, nadig@purdue.edu.

Traditional models for infrastructure security are no longer adequate when applied to containerized infrastructure. Thus, there is a critical need for improved support for container security to deal with a growing database of known security attacks and threats at the container runtimes and the container image level. Containers run on the same host operating system and enlarge the attack surface compared to virtual machines that run on a compact hypervisor [2]. This difference is a primary reason most major cloud providers leverage virtual machines as the fundamental abstraction for representing a node. Security solutions for containerized infrastructure exist but are often overlooked during the container development process. Further, comprehensive solutions that seamlessly integrate several security solutions in a single environment are unavailable.

Where applicable, this paper will measure existing methods of securing a containerized environment across seven dimensions: (i) customizability, (ii) ease of use, (iii) effectiveness, (iv) granularity, (v) scalability, (vi) community adoption, and (vii) support model. We will test open-source tools and platforms against these dimensions to see how they score on a scale of 1 to 3 (low, medium, high) (we place N/A's where a dimension is non-applicable). We will use our research findings to rank techniques where quantitative data is unavailable. In the recommendations section, we define these dimensions and their criteria and discuss any criteria change to suit the discussed security scenario. Further, we will also discuss our critical takeaways for securing containers in a container orchestration environment.

In contrast to existing works on container security, our paper presents a unique perspective on container security in the Kubernetes context and how to secure a cluster from the perspective of a container orchestration system. The literature review and following analysis of popular Kubernetes open-source tools present a complete picture of the security needs in Kubernetes that these tools address. It is critical to understand that containerization is a rapidly growing technology, with several efforts focusing on standardization. Thus, we can expect novel solution developments that effectively deal with the issues presented in this paper. While new tools and techniques will likely address our security concerns, to ensure a consistent evaluation process, we will exclusively consider mature solutions successfully deployed at scale in this paper. Examples include sandbox stage projects cataloged by the cloud native computing foundation (CNCF) (we may reference some sandbox projects, but we will bias against nonproduction-ready technologies). Further, given that the scope of this paper is limited to free and open-source solutions, a commercial solution/security stack with enterprise benefits may be preferable for an organization's particular use case. Thus, rather than using our work as a guide to applying security tools, we recommend using this paper as a reference to understand essential security considerations and the strategies to mitigate security risks in a container orchestration environment.

A. Contributions and Organization [Deepak: Final Review Needed]

The contributions of this work are as listed:

1) Kubernetes security best practices as validated through industry experience and professional testimony.
2) List of commonly-used open-source tools sorted into security domains to address security concerns in Kubernetes.
3) A comprehensive review of container and container orchestration tool components.
4) Compiled Literature Review of Container and Container Orchestration related papers.
5) Future research directions and open challenges derived from our research are presented as research opportunities.

The paper is structured as follows: Section II introduces important terminology and definitions that will be used in the paper; Section III presents a detailed background and discussions on containerization; Next, Section IV provides a high-level overview of container orchestration systems and establishes the scope of this work in the context of orchestration systems like Kubernetes; Section V focuses on open-source tools and discusses the effect of various deployment strategies on security; A comprehensive review of existing works is presented in Section VI; Progressing towards security tools, Section VII delves into the tools, processes, and models for securing a Kubernetes container orchestration environment; Section VIII introduces a scoring model to evaluate the security tools (presented in Section VII), and strategies and recommendations for tool configurations/preferences and security practices; Lastly, we present open research challenges and future directions in Section IX and conclude our work in Section X.

II. Background

This section provides a high-level overview of the important terminology, definitions and concepts utilized throughout this paper. The following subsections provide vital information and context on the scope of containerization, container orchestration technology, and our analysis of the security tools.

A. Linux Kernel Features

This section outlines various Linux kernel features that containers employ to achieve isolation and security. We note that several containers may share the host operating system resources through the container runtimes or the containerization platforms. The following kernel features are employed extensively by all containerization platforms.

1) Namespaces: Namespaces are a feature of the Linux kernel that partitions kernel resources to isolate processes to their managed resources [3]. The container runtime assigns a namespace to run the container processes during container initialization. The container runtime assigns a new process identifier (PID) to each container, isolating different containers under the separate processes which define their own namespace. This isolation prevents the various processes (and containers) from interfering with assigned resources and process identifier (PID) allocations.

There are many namespace types within the Linux kernel, each with unique properties: user, PID, network, mount, interprocess communication, and UNIX time-sharing (UTS) namespaces. These namespace types allow the runtime to virtualize system resources such as container file systems or container networking. For example, using a container network interface (CNI), the runtime can add a virtual network interface to bind to each container. We provide a detailed discussion on CNI and Kubernetes networking in Section IV-D. During the container run, a set of namespaces are allocated, providing a layer of isolation for each part of the container segmented into a namespace.

2) Cgroups: `cgroups` are a Linux kernel feature that performs accounting, limit enforcement, and isolation on resource usage within processes [3]. In other words, `cgroups` allow for fine-grained control and enforcement over resource limits (e.g., CPU, memory, network, and disk I/O). A container runtime uses the `cgroups` drivers to share hardware resources with the containers and provides the flexibility to enforce limits on those resource allocations. Similar to namespaces, `cgroup` types are distinguishable by their functions. For example, Docker [4] employs memory, HugeTBL, CPU, CPUSet, devices, and freezer `cgroups`. In a container orchestration system such as Kubernetes [5], `cgroups` can enforce resource requests and limits at the pod level when running a container.

`cgroups v2` upgrades the original `cgroups` architecture and is compatible with modern Kubernetes implementations. The most significant changes in `cgroups v2` are a simplified tree architecture, new features and interfaces in the `cgroup` hierarchy, and a better accommodation of non-zero UID (rootless) containers [3]. Non-root users typically create rootless containers, a practice in early adoption that offers compelling security benefits over the alternative. Accommodating rootless containers is crucial from a security perspective; containers are left rootless, so they will not gain root privilege on the host if a container is compromised. `cgroups v2` is a straightforward and secure approach to resource management at the process level.

3) Seccomp: Secure Computation Node or Seccomp is a Linux kernel feature that filters system calls to the kernel [6]. Since most threats taking advantage of containers attack the host through system calls [7], the precise ability for Seccomp to assign different profiles per filter allows a more fine-grained solution to reduce the number of system calls made by containers. The filter forces the process into a secure state where the only allowed calls are `exit()`, `sigreturn()`, `read()`, and `write()` on already opened resources. Any other command will terminate the process, isolating system resources from container processes.

4) Linux Capabilities: Traditional UNIX-based implementations identify two permission categories [8] for processes, namely, privileged and unprivileged processes. These privileged processes run as root through the kernel without system checks, whereas unprivileged processes are subject to full permission checking based on their credentials. Containers face additional challenges when a service needs to perform a privileged action without being identified as a privileged process. For example, if an FTP server needs to bind to port 21 (loosely defined: attach the port to its unique address/network socket), it would require root access to perform this action. Thus, Linux capabilities turn the root and non-root dichotomy into a fine-grained access control system [6]. If a container requires port binding capabilities, it can assign that process the `CAP_NET_BIND_SERVICE` capability and bind to that port without creating a privileged process. There are currently forty different Linux Capabilities that cover a wide variety of tasks that initially required a privileged process to complete [8].

Container runtimes use these Linux kernel features to isolate and manage container instances. Therefore, used correctly, the Linux kernel features provide a necessary security layer for containers at the kernel level.

B. Linux Security Modules

Containerization and container orchestration frameworks also leverage existing security modules in the Linux kernel to configure what a container can do on a per-container basis for tasks such as Linux system calls and file permissions. The following outlines various Linux security modules employed to secure containerization systems.

1) AppArmor: AppArmor creates security profiles that various programs associate with to protect the operating system and its applications from security threats [9]. Therefore, AppArmor provides a feasible option to implement strong security controls without the time and configuration complexity of a “trusted OS” [9]. Thus, AppArmors primarily confines programs to a specific set of files, capabilities, network access, and *rlimits* [1].

2) SELinux: SELinux is a method of mandatory access control (MAC), or role-based access control (RBAC), that provides a file labeling system that can enforce policies in a fine-grained manner [1]. In contrast to AppArmor, SELinux provides more granularity for policy enforcement. SELinux applies security labels to objects to confine programs, system services, and access to various resources. These mechanisms operate independently from the native Linux access control mechanisms. For example, SELinux does not conceptualize the role of the root as a superuser.

3) TOMOYO: TOMOYO provides functionalities similar to SELinux and AppArmor with its hands-on access control. The main difference is that TOMOYO uses automatic policy development through “learning” [10]. This security module does not import profiles but instead learns an application’s system calls and behaviors in a testing environment and manages policies based on expected behaviors [10]. Thus, TOMOYO is beneficial for creating per-application security policies automatically.

4) Integrity Measurement Architecture (IMA): The integrity measurement architecture (IMA) aims to provide system integrity by calculating the hash values of specific files and performing a checksum to verify their integrity [1]. The Trusted Platform Module (TPM) [11] chip, a microcontroller dedicated to securing hardware through integrated cryptographic keys [12], manages this process. The limited registers on the microcontroller make this solution challenging to scale against an undefined number of controllers.

However, implementations of Trusted Computing technologies utilize hardware-based solutions to validate the integrity of physical platforms. Projects like Container-IMA [13] have made strides in extending the chain of trust (CoT) to the application layer and building trust in containers [14]. As of this writing, this technology is not mature for mainstream implementation.

Software TPMs (virtual TPMs) extend the functionality of TPMs to the hypervisor [15]. They enable trusted cloud computing through implementation in the host OS kernel, allowing multiple containers to query the vTPM or inside a dedicated vTPM container [6]. These vTPMs are less secure than their hardware counterparts and suffer several security vulnerabilities [6].

C. Virtualization

Virtualization is the process of abstracting the physical hardware of a computing system into one or more virtual machines. Thus, virtualization refers to creating a virtual instance (i.e., a virtual machine or VM) of a computer system, system resource, or otherwise in a layer abstracted from the actual hardware [16]. There are two main techniques for virtualization, namely full virtualization and paravirtualization (shown in Figure 1). Full virtualization simulates the underlying hardware so the guest OS/VM can run in complete isolation. A virtualization layer created by the hypervisor disengages the guest OS from the hardware [17], [18]. Typically, an OS is unaware of the isolation, and its system calls are trapped using binary translation [19]. Full virtualization is optimal when high isolation and portability are required, allowing guest OS instances to migrate between virtualized or native hardware [18]. An example of a full virtualization product is VMWare ESXi Server [20].

Paravirtualization presents a software interface to virtual machines with similar functionality to the underlying hardware. The modified interface allows expensive operations handled by the host machines through hypercalls between the virtual layer and the hardware [17]. The communication between the guest OS and hypervisor improves performance and efficiency by replacing non-virtualization instructions with hypercalls that communicate directly with the virtualization layer hypervisor [19], thus allowing instruction handling at compile time by the host [17]. The Xen project [21] is an example of paravirtualization.

Another flavor, hardware-assisted virtualization, is a type of full virtualization where the microprocessor architecture

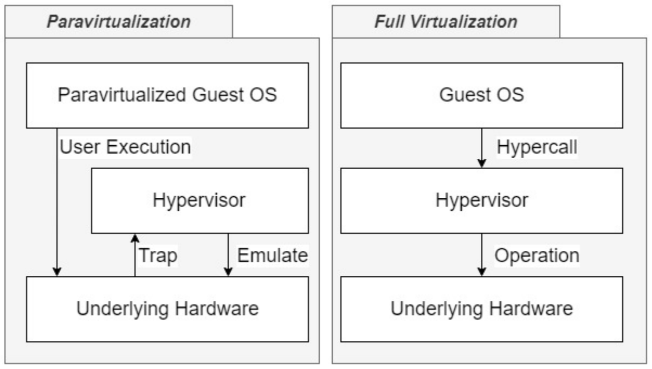

Fig. 1: Guest OS communication process with the hardware based on the virtualization technique.

has special instructions to aid the virtualization process (e.g., VT-x AMD-V) [19]. Like full virtualization, hardware assistance emulates a complete hardware environment where a guest OS can run in isolation. This approach does not require special modifications of a native full virtualization implementation as privileged and sensitive calls are set to automatically trap to the hypervisor, removing the need for either binary translation from full virtualization or hypercalls from paravirtualization [18].

Numerous works have studied virtualization systems, including virtualization techniques and solution approaches [22], [23], [24], virtualized networks and services [25], [26], and security/privacy issues [27], [28], [29], [30], [31]. While numerous solutions have been proposed, a critical challenge with virtualization and virtualized systems in cloud computing is that it is problematic to ensure that the virtualized system is secure as its security relies on the ability to ensure the hypervisor's security. Containers are a promising solution to the problem, as their design isolates applications from each other and the host system. In the Linux kernel, user namespaces implement a unique way of managing resources typically used for running container systems. This feature also finds use on virtual machines and systems without virtualization software, including those without a kernel. Inside a container, users map to process groups through those user namespaces.

Process groups are responsible for resource control and the grouping of processes in Linux environments. A process group can share a set of resources, such as files and memory, while having specific permissions for each user within that group. To achieve this, we create an initial mapping with per-user mappings between the different process groups. For example, a system administrator may create an admin process group and then add each regular (unprivileged) user to that process group. The system administrator would change the permissions on the filesystem so that all of the files belonging to processes in the admin process group are readable by all users while at the same time securing access to files belonging to an unprivileged process.

Containers differ from virtual machines (VMs) as they are more lightweight and use fewer resources. Further, containers are also more efficient than VMs since they share the same operating system kernel. The following section introduces containerization, its benefits, container runtimes, and associated platforms.

III. Containerization

In this section, we provide a comprehensive review of containerization and discuss various container technologies.

A. Introduction

Containerization is a form of virtualization that runs applications in isolated user spaces, called *containers* [32]. Containers are a unit of software that packages code and its dependencies to create a fast, platform-agnostic application [33]. They are a lightweight alternative to virtual machines. In contrast to virtual machines, containers only hold the code and dependencies necessary to run the application, thereby stripping the container image of any non-essential components [1]. Although containers existed for some time (e.g., LXC [34]), the true container era is said to have begun in 2013 with the introduction of Docker [35]. Nowadays, containers are a powerful tool in application development and have seen rapid and broad adoption.

Currently, containers have numerous use cases in various application domains. Initially, they were used to isolate application code but were not popular as they lacked many benefits available in modern containerization frameworks and runtimes. Containers have since improved to become an effective tool in application development, continuous integration and continuous development (CI/CD) pipelines, and application refactoring [36].

B. Use-cases, and Benefits

As containers are a lightweight, portable, and self-sufficient way to package an application with all its dependencies, they are a great way to deploy applications in the cloud. Unlike virtual machines, containers do not store the operating system (OS) and non-essential binaries to the application executable, allowing them to reduce the storage and compute overheads prevalent with VMs. Thus, containers find use in various application domains and use cases, such as:

- *Cloud Application Deployments*: Containers can deploy web applications on any infrastructure (public, private, hybrid and multi-cloud) that supports containerization. The deployment ecosystem includes public clouds like Amazon AWS, Google GCP and Microsoft Azure, private clouds like OpenStack, and bare metal servers.
- *Microservices and DevOps*: Containers provide a lightweight alternative to virtual machines for development environments. Thus, they are suitable for microservice architectures that rely on multiple independent, loosely coupled services for application deployments. This approach packages runtimes, code, and associated dependencies into a single container image that runs on various platforms. Further, developers can rapidly build and deploy software by seamlessly combining software development and application deployment operations.
- *Scheduling and Scalability*: Containers allow various scheduling approaches to execute workers on diverse

hosts. We can distribute tasks across hardware resources using scheduling algorithms rather than a single machine. Further, container scheduling solutions can provide a mechanism for matching tasks with available resources based on the tasks' history rather than manually scheduling every task (predictive scheduling). Also, we can scale application container instances in response to various performance metrics such as application loads, traffic, resource usage and latency.

- *Improving Application Security*: Containers isolate code and processes from each other using namespaces. Each container can be started and stopped independently, making monitoring, managing, and securing easier. Thus, they are beneficial in limiting the attack surface compared to a traditional virtual machine. Through namespaces, containers are isolated from other containers, meaning if there is a potentially malicious container or process, it will be confined within its own namespace.

Further, containers can perform data analytics by running data processing tasks isolated from the host operating system. Applications are portable as containers can migrate between hosts/ecosystems without modification, thus ensuring ease of application deployments, modernization and workload migration across data centers.

C. Runtimes and Platforms

The container runtime software runs and manages containers on a host OS. It utilizes cgroup drivers to handle the allocation of container processes and resources. There are two main container runtimes classes, namely (i) low-level runtimes and (ii) high-level runtimes. Low-level runtimes provide life-cycle management features and do not serve any other functions. Mainstream low-level container runtime examples include *runC*, *crun*, and LXC/LXD. High-level container runtimes provide additional features, including an entire suite of container management features (e.g., Containerd, CRI-O, and RKT). Although out of the scope of this paper, it is helpful to acknowledge that high-level runtimes are often built to fill a specific role/application (i.e., cloud service, operating system type, and container type). Firecracker, the Amazon Web Services container runtime, is an example of a cloud-specific runtime. For more examples of container runtimes, a comprehensive list is available from Cloud Native Computing Foundation [37].

The Container Runtime Interface (CRI) enables Kubernetes to employ a wide range of container runtimes without requiring the recompilation of cluster components [38]. A CRI plugin is embedded into the chosen runtime to make it compatible with Kubernetes primary node agent [39], i.e., Kubelet (see Section IV-C). For containerd, this plugin is known as *cri* and is builtin by default. The CRI-O container runtime is built for Kubernetes and can implement any OCI-compliant containers [40]. The implementation is similar to containerd, which supports runC or other low-level OCI-conforming runtimes to implement the OCI specifications.

The Open Container Initiative (OCI) is an open standard for Linux containers [41]. OCI results from analyzing modern container runtimes like `runc` and the need for standardization around building containers. OCI requires conformance to two specifications: the *Runtime Specification* and the *Image Specification*. The Runtime Specification outlines deploying a filesystem bundle unpacked on the disk. Thus, the standard specifies how an OCI image is unpacked and run to ensure functionality across all OCI-compliant container runtimes [41]. The Image Specification defines how to create an OCI-compliant image. Defining the process and the result of a build system to include the correct metadata about the content and its dependencies ensure that the combination (i.e., image manifest, image configuration, and filesystem serializations) is a standardized OCI image [41].

Some specialized classes of runtimes include sandboxed and virtualized container runtimes. These runtimes take a unique direction in implementations emphasizing increased isolation between containerized processes and the host [42]. Thus, a sandboxed implementation no longer shares the kernel between the host and the container process; instead, it runs on a kernel proxy layer that interacts with the host kernel. A virtualized implementation runs the containerized processes in a virtual machine rather than a host kernel, allowing the container to share the virtual machines' kernel.

Containerization platforms use container runtimes to implement their suite of tools to manage containerized applications. Different container platforms cater to particular use cases, namely container engines, container orchestrators, and managed containerization platforms. Container engines provide a runtime environment that allows the creation and management of containers on a host environment. Examples of container engines include Docker, LXC, Hyper-V containers, and Podman [43]. Container orchestrators typically provide container engine features and allow containers' management at scale. Examples of container orchestrators include Kubernetes and RedHat OpenShift [43]. Managed container platforms augment the container engine and the orchestrator and abstract the platform as a service. Managed containerization platforms include the Google Kubernetes Engine (GKE), Azure container service, Amazon elastic container service, and Rancher [43].

Due to the wide range of variability inside a Kubernetes implementation, the content of this paper will be scoped to a Kubernetes implementation that utilizes the Docker engine and Docker images. For further granularity, Docker implements Containerd finished with a suite of features for their container management software, and Containerd uses runC for the low-level management of containers.

D. Container Registry

A container registry is a repository to store container images. Depending on the registry choice, additional services may be provided with the container store. For example, a cloud-based container registry will provide the benefit of accessibility from anywhere with an internet connection. They can also provide vulnerability analysis on the images, CI/CD developer features, encryption of the container store, and better image deployment capabilities [44]. Alibaba Cloud Container Registry is an example of a cloud-based registry, Docker also

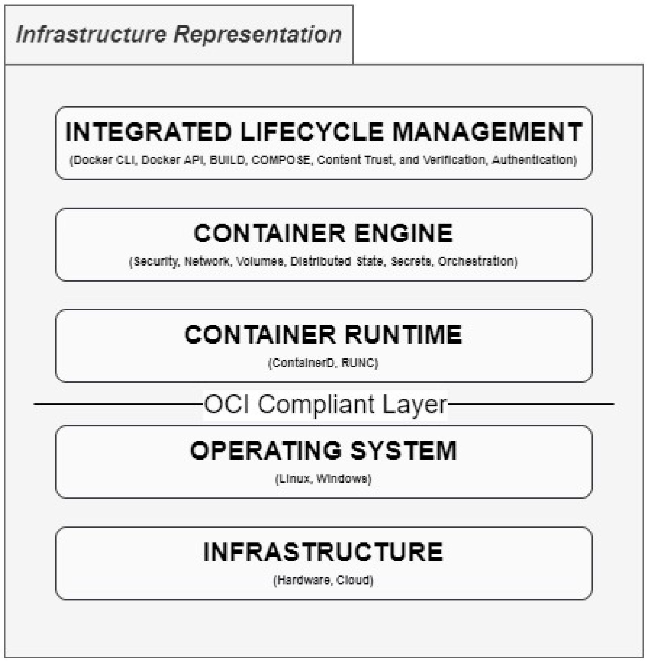
Infrastructure Representation
INTEGRATED LIFECYCLE MANAGEMENT
(Docker CLI, Docker API, BUILD, COMPOSE, Content Trust, and Verification, Authentication)
CONTAINER ENGINE
(Security, Network, Volumes, Distributed State, Secrets, Orchestration)
CONTAINER RUNTIME
(ContainerD, RUNC)
OCI Compliant Layer
OPERATING SYSTEM
(Linux, Windows)
INFRASTRUCTURE
(Hardware, Cloud)

Fig. 2: Infrastructure representation of the container runtime and its ecosystem on a host node [45].

has its own registry that can be used to publish docker images into its cloud-based Docker Hub solution. The CNCF also publishes a list of approved container registries available to developers on their cloud-native landscape [37].

As a potential attack surface, we will briefly discuss container registries in the security recommendations section. However, this paper will not comprehensively review the best container registry services, which is beyond the scope of this work.

IV. Container Orchestration

A. Introduction

The relentless pursuit of efficiency in data center technology [46] characterizes both virtualization and containerization. In contrast to virtual machines, containers are ideal for packaging microservices to reduce overhead. Further, containers reduce hardware costs through consolidated resource usage and an optimized management plane. The advent of container orchestration systems has broadened container capabilities, including scalability, portability, and resource utilization [46]. Container orchestration allows efficient and at-scale container management to enable complex application deployments and application/workload replication. Further, it automates deployment, management, scaling and networking of containers [47], thereby minimizing the tedious operational complexity of containerization. Container orchestrators provide numerous benefits for container management, the primary refinements of which are scalability and operational efficiency.

The design goals of container orchestration systems (e.g., Kubernetes) are to orchestrate containers (e.g., Docker), whereas virtualization platforms manage VM deployments. Container orchestration systems have different design goals and are optimized for a container-based architecture. In contrast, cloud virtualization platforms optimize VM deployments. In container orchestrators, containers are not limited to a single machine and are distributed across all cluster nodes. Therefore, application deployments relying on one or more containers can scale up or down to meet the applications' needs. For example, Kubernetes clusters comprise orchestrator-managed nodes with CPUs and memory, ensuring the appropriate node configurations to run workloads. Kubernetes manages cluster nodes by dynamically allocating resources across nodes based on the applications' resource requirements. Allocating resources automatically ensures that the application runs seamlessly, even as the operator adds additional cluster nodes. Further, resource scaling is dynamic and self-healing in a Kubernetes environment. Therefore, adding new cluster nodes will automatically adjust the number of supported containers per node. Thus, orchestrators ensure that all nodes in a given cluster exhibit uniform behavior.

Container orchestration systems like Kubernetes allow for the deployment of containers at scale with minimal operating system overhead. In contrast to VMs running over hypervisors, containers are lightweight and run directly on the host operating system. Container orchestration systems are ideal for developers or teams working on small to moderately-sized projects. It is also great for organizations that need to deploy multiple applications without the need for IT resources rapidly. Container orchestration systems like Kubernetes are operationally efficient, providing a platform that makes deploying and scaling applications effortless. They achieve this by abstracting away the underlying infrastructure on which the applications run; thus, organizations can focus on their applications rather than the underlying infrastructure.

B. Orchestration platforms

Major cloud platforms and cloud service providers (CSP) have rapidly adopted container orchestration platforms in open-source solutions, proprietary flavors or independent integrations. Various container orchestration platforms are available, including Kubernetes, OpenShift, Hashicorp Nomad, Apache Mesos, Rancher, and Docker Swarm. Numerous managed services are also available, including Google Cloud Run, Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Azure managed OpenShift service and RedHat OpenShift online. Kubernetes is a popular container orchestration system for both on-premise and cloud-managed deployments. This paper focuses mainly on Kubernetes solutions for container orchestration. For a brief overview of Kubernetes, see Section IV-C. Further, Kubernetes provides a cloud provider interface that allows it to integrate seamlessly with different providers [48]. With numerous features geared towards scalability, reliability and cost-efficiency, Kubernetes has emerged as the orchestration platform of choice for most deployments. In the following, we provide a brief overview of some essential Kubernetes terminology in the context of cloud-native container management and orchestration.

C. Kubernetes: High-level Overview

A Kubernetes cluster is composed of two types of nodes, namely (i) master node(s) (at least one) and (ii) worker nodes. The master node is responsible for managing all cluster objects and nodes, including keeping the cluster at the desired state, scheduling application containers on the worker nodes [49] and container lifecycle management. The core components of a master node are listed below.

1) *API Server*: The Kubernetes application program interface (API) server is the central entity that all users, cluster nodes, application deployments, services, and operational systems access in a Kubernetes cluster [50]. The API server implements representational state transfer (RESTful) APIs for Kubernetes. Further, it is responsible for processing RESTful API calls over the HTTP protocol requests from users and numerous internal/external Kubernetes components. The API server authenticates, authorizes the request, and performs the requested API operation.
2) *Controller-Manager*: In Kubernetes, the *controller-manager* is a daemon that implements the control loop in the cluster. It continuously checks the state of the cluster (through its interaction with the API server) against the desired state [5]. It will also implement changes required to reach the desired state if necessary.
3) *Scheduler*: The scheduler runs as part of the control plane and actively watches for newly created pods with no Node assigned. Its responsibility is to find the best Node for a pod to run on; this decision is based on an internal scoring evaluation after it has filtered out unfeasible nodes.
4) *etcd*: *etcd* is a strongly consistent, distributed key-value store that provides a reliable way to store data by Kubernetes. Kubernetes employs *etcd* to store cluster configurations, system states (e.g., actual and desired states), and watch functions for monitoring state changes. Further, Kubernetes uses *etcd* to effect changes necessary to reconcile the actual state with the desired state.
5) *Control plane*: The control plane is the container orchestration layer that exposes the API and interfaces to define, deploy, and manage the life-cycle of containers [5]. It includes the following services: *etcd* (key-value store for Kubernetes cluster configuration and data), API server, scheduler, controller-manager (the component that runs controller processes), and the cloud controller manager (i.e., the integration with cloud providers) [5]. All Kubernetes master nodes operate within this control plane.

The worker node functions to run and maintain containerized applications. They are also responsible for their internal networking to allow external connections to outward-facing pods/applications. Some essential components of a worker node are defined below.

1) *Pod*: A pod is a unit of work that represents a group of one or more containers [5]. All of the running processes and resources inside a pod are co-located and co-scheduled. They run in the same context with shared storage and network resources. All containers have the same IP address and share their port space inside a pod. Note that while they share these resources, from a containers perspective in its unique namespace and cgroup limits, it can not see that it is sharing these resources outside of its namespace.
2) *Kubelet*: Kubelet is a low-level component in Kubernetes that functions as the primary node agent [5]. Kubelet is responsible for starting and maintaining containers running on a singular machine. It works off a container manifest and ensures that pods are running correctly and according to their specs.
3) *Kube-proxy*: Kube-proxy is a service that provides port forwarding and IP redirection functionality to facilitate a proxy between external traffic and internal pods.
4) *Container runtime(s)*: We provide a brief discussion on container runtimes in Section III-C.

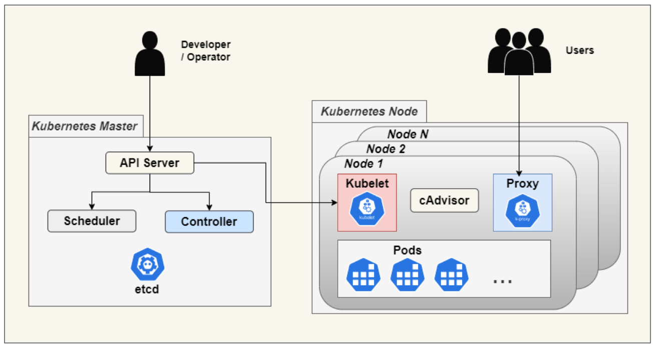

Fig. 3: Kubernetes management components for orchestrating the cluster's worker nodes [51].

D. Networking and Resource Management

Container networking is essential to connect containers within an orchestration system and the outside world. Container networking is a crucial part of container orchestration. Container orchestration systems, such as Docker Swarm and Kubernetes provide built-in support for container networking. Containers rely on numerous mechanisms to communicate, including:

- Shared Storage: Containers can share a common storage volume, such as a network file system for shared data access.
- Network Infrastructure: Containers can connect using network links that allow them to communicate using standard protocols, such as TCP/IP.
- Container Ports: Containers expose ports to the outside world to allow ingress traffic from the Internet.
- Load Balancers: Containers can be load balanced across multiple servers, allowing them to share incoming traffic load.

A container network interface (CNI) is a software interface that connects containers to the physical network. The CNI is responsible for providing the container with an IP address, routing, and other connectivity. Further, the CNI is a software-defined networking (SDN) abstraction that provides a standard API to control the underlying network. This standard API is standardized so that compliant network modules like flannel or weave can plug in to Kubernetes without additional configuration within the CNI abstraction. Within a Kubernetes implementation, there are four primary network models: container-to-container, pod-to-pod, pod-to-service, and external-to-internal networks.

- *Container-to-Container Communication* (*C2C*): C2C is the most straightforward communication model and involves two containers communicating within one pod. The containers can communicate continuously by maintaining a shared volume, allowing continuous data sharing between the two containers. Alternatively, they can utilize inter-process communications (IPC) to communicate using SystemV semaphores or POSIX shared memory [52]. Within Kubernetes C2C, they use IPC for localhost hostname communication.
- *Pod-to-Pod Communication* (*P2P*): A simple method for various Kubernetes pods to communicate is to use P2P IP communication. In this case, each pod is allocated an IP address, and at a primitive level, users can utilize the IP address to reference and communicate with a pod. The above approach is flawed as it can rely on static IP addresses, which negatively impacts the benefits of containerization. Another solution is to utilize pod-to-service communication as described next.
- *Pod-to-Service Communication* (*P2S*): P2S relies on the use of Kubernetes services as a communication medium. Pods are assigned labels that correspond to a created service. This service now serves as the pod's hostname and exposes all containers/endpoints behind that pod. Therefore, services allow a static representation of one or more dynamically created pods.
- *External-to-Internal Communication* (*E2I*): E2I represents outside traffic attempting to enter the Kubernetes cluster. This communication is only possible if components of the cluster are exposed to the network stack. Services like node port or an internal load balancer can make cluster ports and cluster IPs available for external connections [5]. A typical use case example is to expose front-end communications like an Apache web server.

Kubernetes relies on containers for application deployments; it provides several features to ensure that containers connect to each other and the outside world quickly. Service discovery capabilities in Kubernetes ensure automatic discovery and service configuration, thus making it easy for containers to find and communicate with each other. Load balancers automatically balance traffic across containers by sharing traffic loads. Ingress features support ingress controllers that allow containers to accept incoming traffic from the Internet. Lastly, container networking plugins such as Calico, Contrail, and WeaveNet support various networking technologies, such as overlay networks and software-defined networking.

CNIs improve the overall performance of the orchestration system as CNIs can provide better performance than traditional networking solutions through features such as overlay networks and SDN. Further, they offer better security than their traditional networking counterparts using network segmentation and isolation features. Through service discovery and load balancing, they provide for better network manageability. However, container networking has several challenges, including security, isolation, scalability, performance and complexity. Some of these challenges can be beat with container networking best practices, including (i) CNI Use: A CNI improves container networks' security, scalability, and performance by providing advanced networking capabilities for container communication; (ii) Network Policies: Network policies help secure containers by specifying which containers can communicate and with which external resource(s); (iii) Service Discovery and Load Balancing: Improves container connectivity through location services, automatic configuration, and traffic distribution across multiple containers.

E. Policies/Security/AAA

Mutual Transport Layer Security (mTLS) is a lightweight cryptographic protocol designed to provide transport-layer security services to data communications, particularly securing data transmitted within containers. mTLS provides cryptographic guarantees similar to TLS and secure sockets layer (SSL) but with lower overheads and better performance. Entities use the TLS key of the peer at the other end of the connection and the peer's certificate to establish a TLS connection. The TLS protocol design extends TLS to provide an efficient way to authenticate and secure connections between two endpoints that are not necessarily on the same network. In container networking, mTLS provides an encrypted connection between two endpoints regardless of how they connect to the network. Thus, mTLS employs mechanisms built into the network to establish a secure connection between the two containers. Each container registers with a certificate authority (CA) that will establish an end-to-end encrypted channel for its communications. The CA can be any server on the public Internet with a publicly-accessible CA certificate. However, it often is an organization that provides a service for validating and issuing TLS certificates to consumers. Container networking can be complex due to the lack of a unified namespace and global addressing scheme.

While mTLS relies on public key infrastructure, several practical certificate management challenges exist. First, TLS authentication relies on X.509 certificates that a CA must sign to establish trust. The verifying entities CA trust relationship extends to the signed X.509 certificate, i.e., if the CA signs the certificate, the X.509 certificate identity is trusted. However, mechanisms outside the TLS protocol are essential to establish a trusting relationship with the CA. Next, certificate management is challenging as each container relying on mTLS must ensure that the relevant CA signs its certificates. A certificate signing request (CSR) containing the identity and public key of the requestor is sent to the CA to obtain a certificate. Thus, we must ensure CA security and prevent service identity alterations and unauthorized access to a service's certificates. Lastly, certificate distribution (and certificate rotation) is a challenge, particularly for container orchestration systems like

Kubernetes, where services change rapidly due to on-the-fly creation, deletion and replication.

The challenges further compound identity generation and secure communication across distributed clusters as a comprised cluster effectively disable all connected cluster services. In this context, network service meshes may provide a convenient mechanism for usable mTLS security across clusters. However, existing service mesh architectures suffer from performance degradation at scale [53].

Container orchestration systems (e.g., Kubernetes) are highly scalable and secure and employ a combination of security features to ensure containers run in a safe environment. An essential feature of container orchestration systems is the use of security policies. Policies define the rules for how containers interact with each other and the host infrastructure. For example, a policy might specify that containers can only access specific ports on the host infrastructure, preventing containers from unauthorized access and interference. The Kubernetes framework provides a set of policies to enforce security requirements. Kubernetes enforces these policies and prevents unauthorized access to containers, ports and malicious attacks.

The Kubernetes security model relies on the principle of least privilege, i.e., containers only have sufficient privileges needed to perform their tasks. This model prevents containers from accessing data or services outside the container, preventing accidental information leaks or malicious activities. Further, the Kubernetes security-by-default design approach prevents unauthorized access and malicious activity. To enforce these policies, the Kubernetes security model utilizes two security layers: ingress and service authorization. Using ingress, Kubernetes allows access to services by defining rules controlling which hosts can invoke a particular service (configured by the kube-apiserver). Through service authorization, only known users or groups can access services. The authorization is configured in the secrets object supplied to each application.

Container orchestration systems perform AAA functions by authenticating users, authorizing resource access, and auditing user activity. Container orchestration systems such as Kubernetes perform AAA functions by authenticating users, authorizing users to access resources, and auditing user activity. Container orchestration systems typically use a single key-value store for all operations such as user authentication, authorization, and auditing. A single key-value store requires all containers to share the same credentials. For example, if a Kubernetes cluster uses the Openshift key-store, all containers in the cluster must share a valid Openshift admin role to read and write to the key-store. Using a single store has several benefits, such as improved security and simplicity. However, in some cases, using a single store for different purposes is not feasible or desirable.

Kubernetes supports several authentication mechanisms via tokens, users, and service accounts. In the token-based authentication approach, tokens disseminate as bearer tokens to prove that the person holding them is an authorized user. Using Kerberos authentication of service account tokens, service accounts authenticate via Kerberos using principals and tickets from an AD domain controller. All other authentication mechanisms are optional.

Kubernetes also provides a wide array of authorization mechanisms, including role-based access control (RBAC) and access control lists (ACL). Role-based access control (RBAC) is a type of access control that assigns users, groups, and roles to objects within an environment. These objects include files, directories, processes, and organizational units. An example of RBAC in Kubernetes is the API servers that allow users to create custom resources using the `kubectl` command. In RBAC scenarios, two roles typically represent users and administrators: the creator and the resource holder. An authorization API is a Kubernetes API server that authorizes requests based on permissions granted by the cluster administrators and users. Also, a role API is a Kubernetes API server that authorizes requests based on the role of the user making the request. The default authorization mode is Role-Based Access Control (RBAC), allowing user permission assignments based on their role.

Container orchestration systems perform auditing by logging all user activity, including changes to the state of resources such as container creation, updates, and deletions. Auditing is the only way to provide a trace of all user activity on the system. If a container is created and later deleted, the orchestration system audits resource changes that occurred before and after the change. Auditing is critical to providing transparency, but it has some drawbacks. Auditing can be resource-intensive and slow on large systems. Container orchestration systems that use external storage to store audited events are limited by the storage system's performance and cannot scale up fast enough to handle a high volume of events. Some orchestrators allow users to audit container usage by sending an event upon container creation and deletion along with information about the user that created it.

V. Deployment Strategies/Approaches

To operate a container orchestration system at scale with production workloads necessitates the consideration of numerous criteria such as (i) high availability with the replication of the underlying metadata for failure recovery, (ii) upgrade strategy and its impact on application/service downtimes, (iii) integration support for federation with other clouds/endpoints, including support for single sign-on (SSO), storage, RBACs, and networking, (iv) the availability of value-add and overlay features – to support rapid deployments for a large user base.

Typically, container orchestration systems utilize a variety of deployment models, including public cloud-managed services, deployments using management software/platforms, and on-premise deployments. The public cloud-managed services allow users to access a managed service through infrastructure as a service (IaaS) providers. A third-party vendor typically does managed deployments, the most common deployment model for organizations that do not have in-house IT personnel or resources. For example, numerous Kubernetes public cloud services are available through the Google Kubernetes Engine (GKE), Microsoft's Azure Kubernetes Service (AKS), and Amazon Elastic Container Service for Kubernetes. In this case, the cloud provider is responsible for building and managing the cluster and end users can choose from the providers' offered versions only.

Container orchestration platform deployments are also possible through management software/platforms. These tools allow the building of both on-premise or cloud (or off-premise) container orchestration solutions. The management software combines distributions, management, security, monitoring and other integration (compute, storage, networking) capabilities. Further, end users can also build and manage their own on- or off-premise clusters using open-source container orchestrator offerings. The above deployment models have their benefits and shortcomings with tradeoffs associated with management complexities, configuration control, cost, features, customizability, service integration, security and speed of deployments.

Within a Kubernetes cluster, various application deployment models are possible (in contrast to Kubernetes cluster deployments). Strategies include:

- *Rolling deployments*: This approach replaces an older (running) version of an application with a new version without cluster downtimes. This strategy may employ the maxSurge and maxUnavailable parameters to manage the deployments. The maxSurge parameter specifies the number of additional replicas of the pod to create during an update. The maxUnavailable parameter specifies the number of unavailable pods during the update process.
- *Deployments through Recreation*: In this approach, the existing application deployment is subject to a full scale down prior to scaling up the new version. Thus, this method results in application/service downtime during the update process.
- *Blue-Green Deployments* (Alternatively, Red-Black): In this strategy, blue represents the current version of the application, and green represents the new version. At any time during the deployment, only one version of the application is in production. Application traffic is always routed to the blue deployment, while the green deployment is used for development and testing. On successful development/testing of the green deployment, the blue (current) application may be rolled back, decommissioned or upgraded. Thus, the blue deployment serves as a staging environment for the development/release cycles.
- *Canaries*: The canary deployment strategy relies on a partial update (i.e., without a full rollout) to an existing system to test a new version of an application. The canary deployment strategy is similar to a controlled version of the blue-green deployment. The application deployment is gradual, resulting in a small portion of the applications' traffic routed to the new version. Further, the new version gradually rolls out to the entire infrastructure if no errors occur.

Cloud environments rely on numerous data management strategies, including distributed data processing, storage, big data services, data availability, scalability, integrity, heterogeneity, data quality management, governance, regulatory compliance, data privacy and security. However, research challenges exist with managing and governing data ecosystems in the cloud. The scalability of distributed data storage systems is critical to cloud infrastructures, and the lack of cloud-native features for RDBMS has made them less appealing to cloud applications. While NoSQL data stores have gained popularity, they can benefit from performance improvements and faster indexing capabilities. Ensuring data availability also remains a challenge as organizations evolve to support more real-time data for mobile and streaming applications. The authors in [54] present a detailed review of cloud data governance and outline various challenges with data governance, including challenges with data understanding, moderation and associated relationships in the cloud. In the following section, we present a comprehensive literature review of research focusing on the security of containers and container orchestration systems.

VI. Literature Review

In this section, we present a comprehensive literature review focusing on the security of containerization solutions, container orchestration, and security within those realms. This literature review aims to establish a foundation of knowledge in the field and reiterate the importance of cybersecurity within cloud-native infrastructures that employ containers and container orchestration systems. A common theme across many of these works is their purpose in fulfilling a real-world (and often an industry) need for security standardization in containerization. We observe this theme in the community efforts to establish best practices, introduce new security tools and technology, identify security challenges and known vulnerabilities, and more.

We note that works that do not broadly fall into one of the main cloud-native technology categories listed in Table II are listed in Section VI-G. However, as an exception, some works focusing on virtualization and containerization security have been summarized below. We do this to emphasize the importance of understanding security from a virtualization standpoint.

The work in [55] analyzes the lapse in the traditional security of containers made as a result of a trade-off in favor of efficiency. The paper presents a critical literature review of container operations and identifies trends and future research directions in container security. A similarly abstract view of container security is presented in [6]. A more narrowly scoped paper by the authors in [56] looks into security flaws within Linux containers and further collects and classifies 223 exploits effective on container platforms. The authors in [57] review the most common container solutions (e.g., LXC, LXD, Singularity, Docker, Kata-containers, and gVisor) and compare the different features offered in the context of efficiency. The work in [58] studies kernel-based solutions to propose a new taxonomy of container defense at the infrastructure level with a focus on the virtualization boundary.

With a focus on Docker within cloud computing, the work by the authors in [59] addresses concerns in image security by creating a CI/CD (continuous integration, continuous delivery/deployment) system that validates the security of Docker containers throughout the software development life cycle. The authors in [60] focus on a robust container security solution for automated cloud infrastructures that utilize CI/CD. They seek to prepare for the move to cloud computing by defining how cloud computing is transforming industrial automation for the future and creating a strategy for effective deployment and maintenance in the cloud. The work in [61] studies container technologies and identifies the established best practices and solutions that emphasize performance evaluations and run-time adaption but lack security. The complexity of containerization introduces attack surfaces, the authors in [62] identifies these challenge and divides the container system into layers, discusses security-related technology and then identifies security challenges at each layer. Through this analysis, they recommend more robust kernel isolation mechanisms and better security benchmarking tools. The following sections focus on security works that are specific to containerization and container orchestration systems.

A. Authentication & Authorization

Numerous researchers considered the implications of X.509 certificate-based authentication and recognized the move to token-based authentication and authorization as an important objective in the high energy physics R&D roadmap. This work in [63] seeks to embrace modern web technologies and enable the secure composition of computing and storage resources through token-based VO-aware (virtual organization) authentication and authorization infrastructure. The authors in [64] built their own authentication and authorization service for container-based environments from the ground up. Their role-based identity service maps roles with service-specific capabilities and validates requests against the authentication certificate placed by the client on request.

It is generally agreed that inter-service communication requires policing to prevent targeted abuse from other potentially compromised microservices. The researchers in [65] provide extensive details on this topic and propose *AutoArmor*, one of the first attempts to automate inter-service access control policies. A number of Kubernetes developers have cited nontrivial safety issues with Kubernetes operators, which require generic trusts to run actions across a cluster. The work in [66] delves into the development of *Suture*, an access-control mechanism that seeks to prevent the majority of these safety issues with operators.

B. Namespace and Resource Management

The work in [67] sought to employ the fairness metrics used by the Dominant Resource Fairness (DRF) policy in Kubernetes along with their meta-scheduler. The proposed solution, KubeSphere, incorporates a task's resource demand and average waiting time with DRF to improve scheduling efficiency. Similar to the article introducing KuberSphere, the work by the authors in [68] recognizes issues concerning Kube scheduling and fairness for multi-tenant Kubernetes implementations. They provide a framework that studies how policy-driven resource management affects fairness for tenants across a range of quantitative metrics.

The authors in [69] observed the rapid adoption of containers as a result of the significantly lower overhead of deploying containers (compared to VMs). They take this insight and model the performance and resource management in a Kubernetes system that can be used as a basis to design applications on the Kubernetes architecture.

The work by the authors in [70] provides an early perspective on Kubernetes namespace and resource provisioning architecture and implements resource monitoring and a provisioning control loop to better guide resource provisioning strategy.

C. Security Policies

Numerous works in [71] provide sample application scenarios in which we can apply RBAC, resource quotas, pod security policies, namespace segmentation, network policies and image creation/scanning policies to improve security. The authors in [72] propose KubeSec, which recognizes the need to automate critical security profiles within Kubernetes. In their assessments, they augment an automatic AppArmor profile generator by collecting behavioral data of application containers and then transforming the data into the AppArmor policy for each application container.

It is generally agreed on that building trust for containers is a prominent security issue. The work by the authors in [13] explores this problem and presents *Container-IMA* as a potential solution to cope with these integrity status challenges. Providing a broader perspective, the work in [73] analyzes representative integrity threats and describes a set of security requirements for holistic protection from malicious insiders. Similarly, when exploring the attack surfaces within Kubernetes that are under the scope of security policies, it is important to consider the attacks presented in [74].

The authors in [75] present a policy enforcement solution that addresses the increasing demand for shorter security policy enforcement response times. They leverage a machine learning-based prediction mechanism to perform computationally intensive steps in advance while keeping runtime steps lightweight.

D. Network Security

The authors in [76] attack a Kubernetes implementation and then recommend countermeasures to the performed attacks. They assert that in order for the network layer to be protected, network policies need to be implemented in addition to the use of service meshes operating at layer 7. Another work in [77] provides a valuable understanding of container networks, including their overheads, advantages, and limitations in a cloud environment.

Current industry trends favor zero-trust architectures to provide a broad security profile. Building on this idea, the authors in [78] explore previous zero-trust implementations and discuss its potential for future network security as demonstrated through a Kubernetes environment.

Current studies support the notion that the complexities behind networking and security concerns that arise within container networking present a range of challenging issues. In the interest of analyzing current network security methods, the authors in [79] evaluate the performance impact of Calico

Security Mechanism	Related Works	General Approach
Authentication & Authorization	[63], [64], [65], [66]	The consequence of a network of inter-process communication mechanisms between microservices relies on sound authentication methods within Kubernetes to be secured. Due to this approach, the business importance of security within containerization and Kubernetes is explored in these works and provides guidance to address this topic in our paper. Further, this paper relied on studying access-control policies, fine-grained access configurations and industry standards for authentication to identify and study authentication and authorization best practices.
Namespace and Resource Management	[67], [68], [69], [70]	These works collectively focus on understanding how kernel features are utilized to enable process isolation and resource management. Exploring security abstraction at the kernel layer will provide a foundation of security that lends itself to every process in the cluster. Rather than focusing on traditional isolation and resource management techniques, we explore fine-grained system call filters to understand how security profiles at the kernel layer can help filter out potentially malicious system calls.
Security Policies	[71], [72], [13], [73], [74], [75]	Industry best practices around security policies and general system hardening are explored in these works by comparing various Kubernetes systems and their use-cases. The intent of these works is to build a security definition based on real-world production Kubernetes applications. In addition to exterior hardening to prevent external hackers, these works describe the security profiles given to nodes and containers for intra-Kubernetes hardening. Extending security hardening to the Kernel features employed by the container prevents malicious nodes and processes from infecting other nodes and the cluster.
Network Security	[76], [77], [78], [79], [80], [81], [82], [83]	Microservices as part of a larger orchestration system communicate over a virtualized network non-traditionally. Without the simplicity of node-to-node communication, understanding how inter- and intra-Kubernetes components communicate over the non-traditional Kubernetes network becomes a tedious but essential task. Moreover, securing not only the communication endpoints but the communication channels becomes a priority. The works listed in this section explore penetration testing, network security case studies, and network security implications to understand the importance and impact of Kubernetes network security.
Vulnerability Scanning	[76], [84], [85], [86], [87], [88], [89]	A static Kubernetes deployment like any other system is at risk of emerging security threats and attacks. Continuous security scanning and vulnerability checking should be conducted to mitigate these risks. These works explore several open-source and proprietary solutions for security monitoring, benchmarking, and risk analysis. A strong Kubernetes implementation should utilize redundant security and vulnerability checking on a timely basis.
Deployment Security	[90], [91], [59], [60]	Kubernetes deployments are a significant risk consideration, in this section we analyze works that consider the security around the delivery of containers, processes, and application deployments into a Kubernetes cluster.
Miscellaneous	[92], [93], [94], [95], [96], [61]	Scholarly works that either belong in all of the previous categories or explore alternative security approaches to containers and container orchestration systems are listed here. Further, some of these works specifically emphasize the importance of considering security in Kubernetes deployments.

TABLE I: A summary of the works discussed in the literature review section and categorization of their associated security mechanisms explored in this work.

and Cilium and analyze the security of the network policies as implemented by these tools. Another source that seeks to provide insight into the security concerns within container networking is found in [80]. This work stresses the importance of removing the mental model of traditional network security when addressing cloud network security.

To secure multi-tenant Kubernetes systems, the authors in [81] propose a method to segment internal network communication in a Kubernetes environment where network isolation is essential.

The work by the authors in [82] seeks to build on API management by implementing a service mesh plane. They focus on securing the service mesh using mTLS with Istio on Kubernetes. They also use a smart associative model that associates new APIs to categories of service mesh.

The authors propose improvements on one-way certificate authentication of services in [83]. This work reviews a secure communication protocol, mTLS, as well as existing service mesh solutions and ultimately chooses Istio to implement in their solution design.

E. Vulnerability Scanning

The previously mentioned article by the authors in [76] also emphasizes the importance of benchmarking the Kubernetes engine. The benchmarking is performed by enlisting Kube-bench as a necessary tool to identify misconfigurations and entry points into Kubernetes. They also explore the idea of vulnerability scanning at the container level, ensuring that container executions are being scanned and configured properly as well through the use of the Clair scanner. For another perspective on container scanning, the authors in [84] propose a detection framework for Docker containers and document existing security mechanisms and the main threats Docker users face. Similarly, the work by the authors in [85] offers its own perspective on Docker container security with a priority on four aspects of Docker vulnerability: file system isolation, process and communication isolation, device management and host resource constraints, and network isolation and image transmission.

Kubernetes security auditing tools are examined within [86]. A comprehensive definition of required security auditing and shortcoming of current consumer solutions is presented in this paper. Setting the scope to Docker images, the authors in [87] provide in-depth insights into container-focused vulnerability scanners and the shortcomings of these tools. The paper by the authors in [89] extends vulnerability scanning to the application layer within containers and offers a solutions that dynamically explores and tests microservices to detect vulnerabilities.

With network benchmarking in mind, the work by the authors in [88] presents its own modular container network benchmarking platform for container network security.

F. Deployment Security

The authors in [90] demonstrate the security of Asylo by deploying artifacts secured with Asylo into a Kubernetes environment. In this context, the authors describe and deliver a complete CI/CD pipeline running on Kubernetes that addresses four gaps in existing implementations. This work provides important insights on deployment security drawn from their hands-on deployment experience.

A set of guidelines to enable security while setting up the infrastructure for deployment inside Kubernetes is described in [91]. As applications are deployed in the form of containers on a Kubernetes cluster, security must be considered down to each process. The work by the researchers in [60] seeks to prepare the move to cloud computing by defining how cloud computing is transforming industrial automation for the future and creating a strategy for effective deployment and maintenance in the cloud.

A CI/CD system is created to validate the security of Docker images as they are deployed into a Kubernetes cluster in [59]. Further, they provide a dynamic analysis on the security of Docker containers based on their behavior and show the benefits of their proposed solution over the static analyses typically used for security assessments.

G. Miscellaneous

The work by the authors in [92] provide a valuable insights for some of the standard Kubernetes recommendations outline in our paper. This work answers the research question: What Kubernetes security practices are reported by practitioners? In recognizing the current lack of best practice mechanisms and guideline, the work in [61] identifies that best practices/ state-of-the-art solutions have been established within performance evaluations and run-time adaption, but are lacking within the security context. Further, the works by the authors in [92] and [95] explores how attackers can take advantage of Kubernetes implementations that ignore security best practices.

An investigation into a microservices-based architecture running on Kubernetes was conducted by the authors in [93]. This work documents interviews with engineers responsible for the architecture to gather vulnerability and intrusion detection metrics. Following this interview, a survey is conducted to determine existing technologies that can mitigate the identified vulnerabilities. For orchestration decision-making, the article [96] seeks to provide a decision structure for identifying the best-fit container orchestration framework for an application. We note that security can lose its priority when scaled against the bigger picture of strategic decisions. For a more concrete translation the business importance of security within containerization and Kubernetes is investigated by the authors in [94].

VII. Container Orchestration Security Techniques

In this section, we present the various security components essential to evaluating the security of container orchestration systems. While these security components apply to most container orchestration systems, we limit our discussion to the Kubernetes context, which is the most widely used container orchestration platform.

A. Authentication and Authorization

Kubernetes communication and operations rely on authenticated users performing authorized actions. In this context, authentication refers to the authentication of API requests through associated plugins [5]. Authorization evaluates those authenticated requests against predefined policies [5] (e.g., allow/deny). While it is possible to employ multiple authentication methods simultaneously, Kubernetes recommends the use of at least two methods [5]. The first method will successfully authenticate requests for a short-circuit evaluation using two authentication methods. The order of operations for authenticators is not guaranteed, and thus the first authentication method returning a response dictates the authentication process.

B. Authentication Strategies

Authentication in Kubernetes employs client certificates, bearer tokens, or an authenticating proxy to authenticate API requests [5].

- *X.509 Client Certificates*: X.509 certificates are the simplest forms of authentication in Kubernetes. This method requires creating a private key and a certificate signing request for the user to authenticate. Next, the certificate authority (CA) associated with the Kubernetes cluster creates a root certificate. This process is simple, but the significant drawback is manual maintenance to update the certificates. Further, while Kubernetes relies on certificate authentication for high-security authentication, in its current state, Kubernetes has no way to query the validity of certificates. This approach still allows the use of certificates that have been lost and revoked to authenticate against the Kubernetes API server.
- *OpenID Connect Tokens (OIDC)*: OIDC tokens are a flavor of OAuth2 that relies on an OIDC provider and extends the OAuth2 protocol by adding the additional field ID token. Users must authenticate to their identity provider first and, in turn, receive an ID token. This token is used as the bearer token and sent to the Kubernetes API server for authentication. The API server then checks the token's signature and validates it against the configured certificate. Kubernetes does not provide a web interface to trigger the authentication process. As a result, to authenticate, the user must use a command-line utility (e.g.,

kubectl) or a reverse proxy that injects the token. Thus, all requests are stateless, and this solution demonstrates a very scalable authentication method.

- *Service Account Tokens*: Service account tokens make use of signed bearer tokens to validate authentication requests. These are automatically enabled and created with the service accounts created by the API server to run pods. The creation of service accounts is a manual process, and the automatically-created secret will hold the public CA of the API server and a signed JSON Web Token (JWT) for authentication.
- *Static Password Files*: Static password files, which create and identify a password file to the API server, are the least secure and the simplest authentication method. The most feasible use case is with a test Kubernetes environment that requires minimal scaling and does not suffer any risks of having a static plain-text password file on the server.

C. Additional Authentication Mechanisms

Next, we present additional authentication mechanisms for use with Kubernetes.

- *Authentication proxy* allows the Kubernetes API server to identify users from the specified HTTP request header values. This approach is designed for use with an authentication proxy that sets the HTTP header values to authenticate to the Kubernetes API server. The Proxy would acquire a token from the user's credential plugin or authentication provider, augment this into the HTTP request, and forward the request to the Kubernetes service.
- *Webhook token authentication* is a hook for verifying bearer tokens. This service allows users to authenticate through the Kubernetes API server using tokens generated by a third-party entity (e.g., Github). The Kubernetes API server interacts with a hook that POSTs a JSON-serialized `tokenreview` object to the remote service that then returns the authentication results. The primary drawback of this method is that Kubernetes has to query the authentication request to a third-party entity instead of being able to check the validity of the token in its internal data store.

If *Anonymous Requests* are enabled, Kubernetes treats any request that is not rejected but not authenticated as an anonymous request. Anonymous mode is enabled by default if an authorization mode other than *AlwaysAllow* is used; this is recommended to be disabled [97].

D. Authorization Strategies

Once a user is authenticated, the Kubernetes API server will initiate authorization checks to manage the user's access. In the following, we outline various authorization strategies available in Kubernetes.

- *RBAC Authorization*: Role-based access control (RBAC) regulates access to resources based on the roles of users within your organization. There are two types of roles: Role or ClusterRole. Roles are segmented by namespace and must specify which namespace they belong to in the cluster. ClusterRole is a non-namespaced role and applies across all namespaces; these are typically reserved for cluster-wide roles. These role types are defined through a set of permissions; these permissions are purely additive, and there are no deny rules. These roles can be bound to subjects through RoleBinding and ClusterRoleBinding. Both bindings apply the set of permissions in a role to a list of users. RoleBinding specifies a namespace to apply the permissions, and ClusterRoleBinding applies the permissions to all namespaces in the cluster. Subjects are composed of groups, users, or service accounts, and the roles bind to one of these identifiers. Conceptually, RBAC functionality is similar to that of Active Directory, with the attribution of security groups (roles) to users.
- *ABAC Authorization*: Attribution-based access control (ABAC) grants users access rights through policies combining attributes. Resources are assigned attributes that determine who can access that resource (e.g., namespace, resource, team name, IP address, time). On access, the user's attributes are checked against the policy assigned to that resource. If one attribute line matches, then the request is authorized. These policies are applied to users and resources, allowing a more fine-grained access control mechanism. However, with the improvement in granularity, there is increased complexity in managing and understanding these policies.
- *Additional Notes:* The method of Node Authorization is specially purposed towards authorizing API requests made by kubelets. Kubeletes are the primary node agent that manages (creates, updates, and destroys) pods and their containers [5]. This authorization is made with the least privilege in focus and was adopted over the RBAC solution after Kubernetes v1.8 [5].

E. Namespace and Resource Management

Process and System Resources must be managed at a granular scale to ensure isolation. Segmenting resources allows Kubernetes to reduce the inter-cluster attack surface dramatically. In this section, we provide a brief overview of namespaces and cgroups in managing Kubernetes objects and system resources.

1) Namespace Isolation: Namespace isolation involves separating containers in a cluster into their unique namespaces. As defined earlier, Namespaces are a feature of the Linux kernel that partitions kernel resources in order to isolate processes to their managed resources [3]. When Kubernetes creates a new container, it places the container under a "default" namespace if a namespace is not specified. The creation of new namespaces enables per-namespace and container resource segmentation. Thus, namespaces isolate a container and all its resources from the rest of the processes in the system. From the container's perspective, its process IDs start at 1, as if they were the system's first process. A critical challenge with namespace isolation is resources that are still namespace unaware. Consequently, Kubernetes cannot define those resources' namespace and, consequently, cannot isolate those resources to a namespace.

2) Resource Management: A container orchestration system typically employs resource requests, quotas, and limit ranges to manage resources. Below, we provide a brief overview of these mechanisms:

- *Resource Quotas* is a Kubernetes tool that limits the aggregate use of system resources like CPU runtime, system memory, I/O, network bandwidth, and object counts within a namespace. These quotas are created and managed through cgroups. Cgroups is a Linux kernel feature that performs accounting, limits enforcement, and isolation on resource usage within processes. In contrast to namespaces, while namespaces control what resources a container can see, cgroups control how much of that resource the container can use. By default, all resources in Kubernetes start with unbounded memory and CPU requests [97]. Resource quotas are created by creating a resource quota object and scoping it into a namespace.
- *Limit Ranges* provide further control over resource limits on a per-object basis. They extend the resource quotas accounting by segmenting resource limits within the namespace by an object, type, or a shared identifier attribute.
- *Resource Requests and Limits* provide a granular control on pod and container creation to request a set of resources and add an upper limit for resource usage as well. Within that namespace, the kubelet will reserve at least the requested amount of resources and limit the potential maximum resource usage for the pod. These requests and limits are specified during pod creation.

These limits influence the quality of service (QoS) class assigned to the pod. There are three QoS classes (from best to worst): guaranteed, burstable, and best-effort [48]. Depending on how many containers have their resource requests and limits specified, Kubernetes will place the pod into one of the three classes. If there is a shortage of resources, processes are killed starting at the best effort, i.e., the class with the lowest out-of-memory (OOM) score. The container runtime assigns this OOM score based on the configured QoS class.

We note that Kubernetes recommends using `systemd` as the `cgroup` driver to enable `cgroups` under their default container runtime (containerd). Therefore, analyzing the right `cgroup` driver for a given container runtime is vital.

F. Security Policies

Security policies are crucial to protect container orchestration systems from malicious actors at the host level or a container within the cluster. This section introduces various policy enforcement tools for container orchestration systems. We limit our discussion to security policies in the context of Kubernetes clusters.

1) Policy Enforcement Tools: Policies are implemented to apply a security context for pods and containers. These policies determine how workloads are managed in a Kubernetes cluster, a secure context provided by policies will reduce vulnerabilities on the cluster. This section outlines policy enforcement tools available to Kubernetes.

- *Pod Security Policy*: These policies are a cluster-level resource that provides control of the security-sensitive aspects of pod specifications. They define conditions that must be met for a pod to run on the system. These controlled conditions include host namespaces, network information, ports, security profiles, capability information, and security contexts. The fundamental issue with pod security policies was that they were confusing in implementation, and as a result, permissions applied on pods were broader than necessary. Pod security policies are depreciated as of Kubernetes v1.21 and will be removed in v1.25 [5].
- *Pod Security Admission Controller (KEP 2579)*: KEP 2579 is a new admission controller developed by Kubernetes as a replacement for the deprecated pod security policies [98]. As of this writing, the pod security admission controller is in its beta stage. This controller is a validating controller only, i.e., it does not change existing pods and only validates the security contexts of newly instantiated pods.
- *Kyverno*: Kyverno is a policy engine designed for Kubernetes that can validate, mutate, and generate configurations using admission controls and background scans [99]. Kyverno policies are based on Kubernetes resources and do not require learning a new language. However, Kyverno is still an external plugin and requires the use of an admission controller webhook. Kyverno excels in having the simplicity of Kubernetes-style composition in its policies. A drawback is an inability to compose highly complex policies due to the lack of programmability. Kyverno is an open-source project that is currently in its sandbox stages within the Cloud Native Computing Foundation (CNCF) [100]. It is a dedicated Kubernetes solution that sought to fill in the gaps presented by the pod security policy admission controller.
- *OPA Gatekeeper*: Open Policy Agent (OPA) Gatekeeper allows the enforcement of custom policies on Kubernetes objects without recompiling or reconfiguring the Kubernetes API server [101]. OPA Gatekeepers build on the original OPA implementation by introducing the following functionality, including an extensible, parameterized policy library, native Kubernetes custom resource definitions (CRD) for instantiating and extending the policy library, and audit functionality [102]. OPA executes a webhook to validate the object against the CRD-based policies implemented by OPA Gatekeeper on object creation, update, or deletion. OPA's core policy language is Rego, a datalog-like language that is purposed for OPA Gatekeeper and intended to make reading and writing policies easy [101]. This approach allows the OPA Gatekeeper to implement expressive and complex programmed policies. However, a new programming language will present a learning curve and increased time for adoption. OPA Gatekeeper is a CNCF-graduated open-source project [101]. OPA has wider adoption than the previously described approaches as it builds on the well-established Open Policy Agent, which is not exclusive to Kubernetes.

- *Admission Controllers*: controllers enforce semantic validation of objects during the create, update, and delete operations. Alternatively, new admission controller software can enforce organizational policy through admission controller webhooks. These webhooks execute when a resource is created, updated or deleted and validate those commands against the target plugin. The dynamic admission control further illustrates these functionalities in Kubernetes.
- *Additional Mechanisms*: The *Pod Security Standards* is an additional mechanism to define security policies in a Kubernetes cluster. The pod security standards define three policies to broadly cover the security spectrum. These include privileged, baseline, and restricted profiles. The privileged profile allows unrestricted access and grants the broadest possible permissions. The baseline introduces a minimally restrictive policy, preventing known privilege escalations. However, the baseline profile allows the default pod configuration. Following current pod hardening practices, the restricted profile is a heavily restricted policy. These policy definitions are decoupled from policy instantiation to allow a common language across policies.

G. Network Security

Typically, namespaces provide isolation in containers by partitioning resources among processes. Each networking component has its own namespace, i.e., the network namespace, a copy of the network stack. The network namespace includes network interfaces, routing and firewall rules assigned to each process or container [49]. The tools listed in Section VII-F1 can enforce network security policies. The container network interface (CNI) is a network plug-in that aims to connect the container engine to the network [49] and enforce policies. Since network plug-ins have different flavors, the network policies are written in the language that the plug-in reads. By default, Kubernetes uses the Canal CNI network plug-in, which is a combination of two popular plug-ins, Calico (policy capabilities) and Flannel [103] (network layer). Thus, Canal integrates the simple network overlay provided by Flannel with Calico's network policy infrastructure [103].

We note that networking is a prominent component of Kubernetes and focuses on four distinct areas, including (i) local communication that enables highly-coupled container-to-container interactions, (ii) communication between pods, (iii) pod communication with services, (iv) service communications with external systems. Kubernetes relies on dynamic port allocation to support application networking, and the Kubernetes network model implements a container runtime on each node on the cluster. These container runtimes use container network interface (CNI) plugins to manage and secure the container network. Numerous CNI plugins exist (see [104] for an exhaustive list) to support a wide range of features, network interface management, integration with other container orchestrators, and advanced IPAM features.

H. Vulnerability Scanning

Continuous vulnerability scanning and container integrity checks are essential throughout the cluster's operational lifecycle. While the above tasks are a crucial part of continuous integration and continuous deployment (i.e., CI/CD) security, they do not replace security-focused application development. This section provides an overview of some vulnerability scanning tools available.

- *Anchore Engine*: The Anchore Engine is an API-centric open-source container vulnerability scanning and software bill of materials (SBOM) generation tool [105]. The engine combines Syft and Grype into a single tool with a web-based management portal. *Syft* generates a comprehensive SBOM from container images and file systems [105]. These SBOMs are prime candidates for vulnerability detection when used with a scanner tool, for example, Grype [106]. *Grype* takes the SBOM generated by Syft, a container image, or a project directory [105] as inputs, scans these inputs and generates a list of known vulnerabilities. These vulnerabilities can be found in major operating systems, including Ubuntu, CentOS, and AWS Linux, or language-specific packages such as Ruby, Java, Python, and JavaScript [107]. Together, these two tools allow developers to perform a detailed analysis of their container images, run queries, produce reports and define policies that inform CI/CD pipelines [108]. These tools can also be augmented to support new queries, image analysis, and policies. This approach allows their utilization within a CI/CD pipeline for automatic scanning.
- *CoreOS Clair*: Clair is an open-source tool that provides the static analysis of vulnerabilities in OCI and docker containers [109]. The Clair API indexes container images and matches them against known vulnerabilities [110]. Various CVE sources refresh the vulnerability database periodically. On detecting insecure software, the tool can alert or block deployments [110]. The tool can also detect threats before container execution using static image analysis techniques, making it an essential step in a CI/CD pipeline [110].
- *Dockscan*: Dockscan is a simple script that analyzes the Docker installation and running containers for local and remote hosts [111]. Dockscan generates HTML report files on execution that can audit containers to check if they are over resource limits, or are spawning too many processes or modified files [108].
- *Kics*: Kics is a highly customizable query-based solution that compares the security of infrastructure as code (IaC) against a set of rules you define [112]. Currently, Kics scans across the following IaC solutions: Terraform, Kubernetes, Docker, AWS CloudFormation, Ansible, Helm, Google Deployment Manager, AWS SAM, Microsoft ARM, Microsoft Azure Blueprints, OpenAPI 2.0 and 3.0, Pulumi, Crossplane, Knative and Serverless Framework [112]. It can be directly integrated into the CI/CD pipeline that you are using to build on your Kubernetes deployment, some of its current integrations include

Azure Pipelines, Gitlab CLI, Terraform Cloud, AWS, CloudBuild, Visual Studio Code, and more [112].

- *Checkov*: Checkov is a static code analysis tool for scanning IaC files for misconfigurations that may lead to security or compliance problems [113]. They allow custom policies to be written in Python, YAML, or connection states but also scan for compliance with industry standards as defined by the Center for Internet Security (CIS) and Amazon Web Services (AWS) Foundations Benchmark [113]. At the moment, Checkov supports these IaC file types: Terraform (for AWS, GCP, Azure and OCI), CloudFormation (including AWS SAM), Azure Resource Manager (ARM), Serverless framework, Helm charts, Kubernetes, Docker [113].

VIII. Recommendations

In this section, we provide recommendations and best practices to secure a Kubernetes implementation for the various security techniques discussed in Section VII. First, we introduce a qualitative scoring model to rank each recommendation. Next, we evaluate existing strategies and provide some recommendations for each container orchestration security technique presented in Section VII.

A. Scoring Model and Evaluation Approach

First, we outline our evaluation model and its parameters. Our evaluation of the tools reflect a qualitative analysis based on our literature review. Our evaluations will span seven (07) dimensions for each security strategy, including (i) customizability, (ii) Ease of use, (iii) Effectiveness, (iv) Granularity, (v) Scalability, (vi) Community Adoption, and (vii) Support Model. We note that all of the seven dimensions may not be applicable to each security strategy. In the case a dimension is not applicable, you will see a N/A value filled in for that particular dimension. Otherwise, scoring will range from 1 to 3 (low, middle, high) and will be used to evaluate the tool within that dimension.

The evaluated score for a tool is founded upon our collective research, literature review, and evaluation of the tool. Some factors that influenced the scores are the level of support these tools have in the community (e.g., a CNCF-backed project), the tools' reception in the community (widespread adoption of the tool speaks to its effectiveness and ease of use), as well as the cadence of development and support. Further, we leverage our experience with Kubernetes and container security to make these evaluations to extrapolate the security recommendations below.

We further break down the above dimensions as detailed next. *Customizability* indicates the level of configurability the tool has, which includes its flexibility to be used within a unique environment or to fulfill a specific use case. *Ease of use* is a measure of the technical proficiency required to operate the tool, as an example, a tool with its own proprietary programming language will be scored higher than a tool that utilizes a common, standard language. *Effectiveness* represents the tool's ability to solve the security concerns in its respective section. In the case of a scanner, this can be a measure of how many container image flaws the tool is able to identify against the median performance of scanners (measured against the true count of flaws). For a security policy, this would represent the tool's ability to stop malicious activity in the orchestrator's environment down to the container. *Granularity* reflects the tool's ability to exact very specific instructions, allowing security to be deployed at a very low level. A simple example of granularity would be different security groups applied to a schema allowing different users to have a different representation of data in front of them. In this case, we look for more complex granularity, for example, policies that limit a container's resources by affecting the `cgroups` that manage them. *Scalability* shows how the tool scales from a small-scale environment to a larger enterprise-level environment. *Community Adoption* is determined by the following the project has in the community as well as current active adopters across top companies that lead their industry. *Support Model* is a value representation of the support that is offered by the tool owners, this can be seen through an online forum or some tiered model that provides enterprise-level support.

B. Authentication Strategies

1) Evaluation: This section will utilize six of the seven measured dimensions in the context of authentication strategy: *Customizability*, *Ease of use*, *Effectiveness*, *Granularity*, *Scalability*, and *Community Adoption*. The support model is not included in this evaluation as the recommended authentication strategies do not rely on a proprietary tool but instead on industry standards of authentication. As a result, the support responsibility falls on the implementing party.

The results of the evaluation are shown in Table II.

Evaluation of Authentication Methods							
Methods	**Customizability**	**Ease of use**	**Effectiveness**	**Granularity**	**Scalability**	**Community adoption**	**Support model**
X.509 Certificates	2	2	2	3	1	2	N/A
OpenID Connect	3	2	3	2	3	3	N/A
Service Account Tokens	1	3	2	3	3	3	N/A
Static Password file	2	3	1	2	1	1	N/A

TABLE II: Evaluation results for Authentication Strategies: X.509 certificates, OpenID connect, service account tokens, and static password file. Note that the support model is unrated as these strategies typically rely on an organization to implement and maintain.

2) Recommendations: Service account tokens are automatically enabled and should be maintained as they provide an easy, automated authentication method for service accounts.

Along with service tokens, the OpenID Connect token authentication method is recommended as the best-practice option by a majority of practitioners [97].

The default authentication modes as well as anonymous requests need to be disabled. The default authorization method is set to `AlwaysAllow` and needs to be set to use all the required forms of authentication. Note that with multiple authentication methods listed, an actor only needs to pass one method to gain access. Without clear limits and policies, the user impersonation feature should also be disabled. Finally, regarding admission controllers, different types of code that intercept requests to the Kubernetes API server prior to the persistence of the object but after authentication and authorization, need to be enabled. Many Kubernetes services discussed in this paper utilize admission controllers that must to be deliberately turned on to function.

C. Authorization Strategies

1) Evaluation: This section will define six of the seven measured dimensions in the context of Authorization Methods: *Customizability*, *Ease of use*, *Effectiveness*, *Granularity*, *Scalability*, and *Community Adoption*. The support model is once again not scored in this section as the methods recommended are common industry standards and are not explicitly offered as a tool/proprietary product.

The results of the evaluation are shown in Table III.

Evaluation of Authorization Methods							
Methods	**Customizability**	**Ease of use**	**Effectiveness**	**Granularity**	**Scalability**	**Community adoption**	**Support model**
RBAC	2	3	3	2	3	3	N/A
ABAC	3	1	3	3	3	2	N/A

TABLE III: Evaluation results for Authorization Strategies: RBAC and ABAC. Note that support model is unrated, these strategies are the responsibility of the organization implementing them and other than documentation, typically, there is a lack infrastructure for a support model.

2) Recommendations: Both RBAC and ABAC methods are effective solutions as a Kubernetes authorization methodology. RBAC is an easier solution to manage and understand as it functions similarly to access control within popular directory services like Windows Active Directory [114]. It can also be coupled with software like RBAC Lookup and RBAC management by Fairwinds which can make management and visibility into your RBAC policies easier [115]. A potential drawback for ABAC is that it requires SSH and root access to make authorization changes, potentially increasing your risk by giving a service root access. However, ABAC provides a more granular approach to security. A strong implementation of ABAC provides a higher level of control over an organization's assets than RBAC.

[Deepak: Incomplete sentence] In the event a tool that manages authorization is required, an example of a comprehensive tool for Authorization management would be Cerbos. Cerbos abstracts the authorization layer between applications and cluster resources, allowing us to implement YAML-based access control policies. Further, Cerbos implements a dynamic blend of RBAC and ABAC models by allowing us to attach attribute roles to policies [116].

D. Namespace and Resource Management

An evaluation was deemed unnecessary in this section. All the techniques mentioned below are built-in and should be utilized to minimize the attack surface within an environment.

1) Recommendations: Segmenting containers further than the default namespace is required for a more secure system. Practitioners recommend that namespaces be separated for each team or development focus [97]. If a container becomes compromised and gains root-level access, its impact will be limited inside to its defined namespace. Further, authorization methods can use namespaces as an attribute on which to enforce access control, providing more granular security on resources.

Resource quotas are important to preventing a DoS (Denial of Service) attack from using all the resources on the system [6] or to prevent a container mid-meltdown from causing more damage. Resource quotas should be used to manage the number of instances for a container, the CPU utilization, and memory consumption within a pod or namespace [97]. Resource requests and limits should be used for important pods and their containers to increase their priority inside of the QoS hierarchy.

Limit ranges should be considered as a useful policy tool to enact per-object resource limits across a namespace that may lack individual resource requests/limits or as a secondary form of redundancy to ensure no over-allocation of resources to a container/object type.

E. Kubernetes Security Policies

This section compares different policy enforcement tools to implement on a Kubernetes cluster and then outlines important security practices that should be considered in policy-making.

1) Evaluation of Policy Enforcement Tools: This section will define the seven measured dimensions in the context of Policy Enforcement Tools: *Customizability*, *Ease of use*, *Effectiveness*, *Granularity*, *Scalability*, *Community Adoption*, and *Support Model*.

The results of the evaluation are shown in Table IV.

2) Policy Enforcement Recommendations: There are a few policy enforcement tools not mentioned in this study: K-rail [117] and MagTape [118]. These smaller solutions lack the widespread adoption deemed necessary to consider at the time of this writing. Given that the Kubernetes Pod Security Policy is being depreciated and the KEP 2579 is still in beta,

Evaluation of Policy Enforcement Methods							
Methods	**Customizability**	**Ease of use**	**Effectiveness**	**Granularity**	**Scalability**	**Community adoption**	**Support model**
Pod Security Policy	2	2	2	2	3	2	2
Pod Security Admission	2	3	2	3	3	1	1
Kyverno	2	3	3	2	3	2	2
OPA Gatekeeper	3	2	3	3	3	3	3

TABLE IV: Evaluation results for Policy Enforcement Tools: Pod Security Policy, Pod Security Admission, Kyverno, and OPA Gatekeeper.

the preferred policy enforcement tools are Kyverno and OPA Gatekeeper.

If the use case is to minimize the attack surface on a Kubernetes cluster through the use of complex policies, OPA Gatekeeper is the most powerful policy enforcement tool [119]. Kyverno covers most pod security features and offers a model of simplicity through its policy configuration readability and use of Kubernetes resources in its policies. Nonetheless, it struggles compared to the Gatekeepers' large library of templates and security feature maturity. However, Kyverno remains a strong consideration for easier and similarly secure implementation of policy enforcement.

3) Security Policy Recommendations: This section seeks to outline important considerations and practices when enacting policy enforcement. All of the practices listed below are implementable by both Kyverno and OPA Gatekeeper policy enforcement tools. However, we will discuss these practices in the context of an OPA Gatekeeper implementation.

- *Non-root & Read-only Containers*: Most containers will run as root by default. This can be convenient as it allows unrestricted container management, which may be useful for development. In production, however, this gives malicious actors on the container unrestricted container management, allowing them to run unsafe processes, inject code, open ports, and many more actions that make your application vulnerable [120]. Using Linux capabilities, as explained in the background section (See Section II), enables us to run containers under a non-root identity, using capabilities to perform the necessary augmentations on the container while maintaining a non-root identity [6]. Further, it also ensures that within a policy enforcement, we can force non-root containers on any production or exposed clusters.
  Containers should also be required to be a read-only root file system [121]. Unless a container is stateful, there is no need to write any data on the file system. If stateful, it should have precise permissions to write exclusively on the persisted data. Allowing unnecessary expanded permissions across the filesystem undermines the container's immutable infrastructure and increases the attack surface within the container [122]. We refer the reader to the capabilities section under LSM [Deepak: Section **??**] and kernel features below for further guidance on non-root and read-only capabilities.
- *Require Labels*: Labels are an important tool to organize and create subsets of objects [5]. They represent a key/value pair that can be applied as an identifying attribute of an object. Policy enforcement tools can be used to force created objects to have labels through constraint templates. It should also be noted that Kubernetes services can also utilize label selectors to help manage different cluster components. An important use case consideration is forcing the use of labels during namespace creation. By default, namespaces are not required to be labeled. Policy enforcement can require specific attributes to be specified to allow more organization and easier querying of Kubernetes objects.
- *Require Specific Container Registries*: Kubernetes will, by default, pull images from the Docker Hub if the request is not a fully qualified path. If a use case requires a private registry, constraint templates can be used again to require specific repository addresses. This can remove errors in programming from pulling the wrong image or the use of a non-approved image from another repository. An analysis on the top 1000 docker images in 2017 showed that 70% of the images suffered from high severity vulnerabilities, and 54% have critical severity issues [123]. Best practices to secure containers include validating the security of the container image and installing the minimal version of operating systems to reduce the attack surface [123][97]. It is important to run containers through vulnerability scanning tools to identify a significant portion of the flaws in the image. As an example, see Docker Content Trust (DCT), for a method of verifying image digital signatures sent from remote docker registries [124]. Another useful tool is the Notary project, which consists of a server/client infrastructure for running and interacting with trusted collections [125].
- *Namespace and Resource Management*: Namespace and resource management have been explained in detail previously (See Section VII-E). We can employ policy enforcement to force the use of resource management during object creation.
- *LSMs and Kernel Features* Various important security profiles pertaining to Linux Security Modules (LSM) and Kernel Features will be considered in this section. Because all of these tools have been previously identified and defined, this section will contextualize them inside their Kubernetes environment.

1) *Capabilities* are a kernel feature that can escalate certain functions of a superuser to a specified process. This allows a container to run privileged actions without running as root. Capabilities are crucial for maintaining non-root containers which

reduce the attack surface within a container. As described earlier, capabilities can be configured through the security context configuration for pods. Using security contexts and a security policy that enforces the use of security contexts, ensure that *allowPrivilegeEscalation* is set to false. This ties in well with non-root containers and read only containers — Which can be achieved through a *runasuser "user-id"* and a *readOnlyRootFilesystem* security context.

2) *SELinux* is an LSM that provides an alternate security infrastructure within the Linux Kernel. With precise specifications, it can finely control what actions a system allows users, processes, and daemons. Thus limiting compromised users or daemons from inflicting harm on a system. In the context of a container, SELinux is used to assign security labels, a default profile will allow seLinuxOptions to be specified [5]. These seLinuxOptions can be used to apply SELinux labels to a container.
3) *AppArmor* is another LSM that supplements the standard Linux user and group-based permissions to confine programs to a limited set of resources [5]. An AppArmor profile can be configured for any application to reduce its potential attack surface and provide greater in-depth defense [5]. These profiles are specified per container. Enforcement works by auditing compliance of prerequisites, and then forwarding the profile to the container runtime for enforcement. Prerequisites need to be met for the container to run [5]. The main use of AppArmor is to restrict a container's individual processes' access to resources.
4) *Seccomp* is another kernel feature that can restrict a container's syscalls [5]. It allows unprivileged processes to constrain themselves and drop the ability to make certain system calls. It benefits from being early in the system call handling sequence by reducing the amount of code malicious actors go through till they are blocked by seccomp [126]. The default profile blocks 44 system calls out of 300+ and still provides wide application compatibility [127].

- *Generic Security Policies* General-purpose security policies must be implemented to protect Kubernetes clusters from external attacks. Open network ports for Kubernetes tools like Kubelet, API server, etcd datastore, or other network plugins should require authentication for access [97]. SSH access from public networks should be restricted [97]. Users must adhere to least privilege and an audit policy should be created to log user actions [97].

F. Network Security Recommendations

Data running across a Kubernetes cluster should be limited to a zero-trust principle. Traffic from outside the cluster is a clear threat, but an obscure attack surface comes from other pods within the cluster. If compromised, a rogue pod can send malicious traffic to other pods or the public network. Network policies should be used to filter both ingress and egress traffic across the cluster. These network policies can be applied to specific pods or groups of pods, if a pod is not affected by a network policy, then all connections are allowed to and from the pod [128]. As a base network policy, it is recommended to apply a default-deny-all network policy to all pods to prevent unapproved communication.

Monitor Network Traffic to catch suspicious traffic or to tighten controls on existing traffic. Monitoring tools are listed in section VIII-H under logging; it is the responsibility of an organization to choose the monitoring tool that is most beneficial based on the organizational infrastructure setup.

We recommend prioritizing and enabling secure sockets layer (SSL) or transport layer security (TLS) protocols to secure and encrypt communication between Kubernetes components. Apply TLS and SSL certificates for Kubernetes components and enable TLS communication between the Kubernetes API server, etcd, kubelet, and kubectl. For dual component authentication, use a networking module that allows the use of mTLS (mutual TLS).

For further segmentation, different network segments can be created in Kubernetes, allowing different environments like production and development to have their own segments similar to a traditional network. Within these environments, pods act as endpoints and can be assigned their own IP addresses.

G. Vulnerability Scanning

1) Evaluation: This section will define the seven measured dimensions in the context of Vulnerability Scanning: *Customizability*, *Ease of use*, *Effectiveness*, *Granularity*, *Scalability*, *Community Adoption*, and *Support Model*.

The results of the evaluation are shown in Table V.

Evaluation of Vulnerability Scanning Methods							
Methods	**Customizability**	**Ease of use**	**Effectiveness**	**Granularity**	**Scalability**	**Community adoption**	**Support model**
Anchore Engine	1	2	3	3	3	2	3
CoreOS Clair	2	2	3	3	3	3	3
Dockscan	1	3	2	1	3	3	2
Kics	3	1	3	3	2	1	1
Checkov	3	2	2	2	3	3	2

TABLE V: Evaluation results for Vulnerability Scanning tools: Anchore Engine, CoreOS Clair, and Dockscan.

2) Recommendations: Practitioners recommend using a mix of Dockscan and CoreOS Clair for vulnerability scanning [97] your container executions. Clair will provide the static scanning before container execution and Dockerscan can generate reports on existing containers. However, policies are written in OPA and the creation of query rules can be a complex task. Fortunately, there are pre-configured queries that can help an organization establish a base of best practices for a given environment. Anchore Engine functionality extends itself partially to both Dockscan and CoreOS, but versions other than the enterprise version are missing important policing and reporting capabilities.

Regarding infrastructure as code (IaC) scanning, Kics offers a competitive solution for a CI/CD pipeline IaC test that can outperform many tools due to its granularity and flexibility [112]. A competitor is Checkov by Bridgecrew which provides static code analysis of IaC files for misconfigurations [113]. This Python-based scanning tool also provides a second paid model for enterprise support. Other enterprise open-source projects that are worth considering if you are willing to pay for scanning software are Terrascan [129] and Indeni Cloudrail [130]. A study measuring these tools against common use cases in Terraform concluded that the open-source scanning tool with the best overall catch rate was Kics, which dominated AWS-based Terraform code detection [131]. Given this unique use case, we cannot definitively recommend one tool over the other.

Code should be implemented through CI/CD pipelines that include security checks like the tools above. Further checks on the code can come from the Kubernetes security policies described previously that audit object creation, mutation, and deletion. These CI builds should automatically fail and generate an alert when they fail security checks. For infrastructure benchmarking, refer to *Benchmarking* in Section VIII-H.

H. Other Security Recommendations

This section describes additional Kubernetes security best practices that are important but do not warrant a separate section.

- *Logging*: Logging should be enabled for applications, containers within each pod, and Kubernetes clusters for system health checks [97]. These logs should then be monitored at regular intervals, alerts should be set up to catch anomalous changes in log metrics [97]. The practice of continuously checking and reporting on logs through monitoring software will allow for rapid response if the cluster approaches or finds itself in an unhealthy state. We recommend organizations employ an observability tool for this purpose. Examples include: Kubernetes dashboard [132], Grafana [133], Elastic Stack (ELK) [134], Prometheus [135], Falco [136], and FluentD [137].
  Cloud-native observability is a growing interest in an effort to achieve the expected benefits of cloud. It represents the monitoring goal of securing and maintaining system performance and availability. The CNCF and CNCF Observability technical advisory group (TAG) reported on how organizations are using observability tools at the end of 2021: the report found that of the CNCF backed projects, Prometheus had an 86% adoption rate and is currently leading in adoption as a Cloud-native observability tool [138].
- *Node Security*: In order to secure Kubernetes nodes, ensure that access requires the use of certificates rather than a username-password combination. This switch provides a number of benefits: the authentication token is no longer under the control of the user who may create an easy-to-remember non-secure password. Certificates can be configured to expire forcing a password rotation after a period of time. Finally, the asymmetric pattern of certificates prevents the party from knowing the user's true private key/identity, preventing them or a middle-man from impersonating the user's identity.
- *Restrict Access to etcd*: etcd is a key-value datastore that is used by Kubernetes for data access. Secrets are by default stored inside this datastore, base64 encoded in plain-text, and while Kubernetes can encrypt etcd, the encryption key is stored as plain-text in the config file for the master node. This is typically found on the server if the API server is running as a service or it can be found mounted on the API server pod if it's being run on a pod. It is recommended to use an alternate secret management tool like Vault [139] or CyberArk [140] for further encryption and restricted access [97].
  Another method of restricting access to etcd is by using a different certificate authority for the communication between the API server and the etcd server. Segmenting their authentication plane adds a layer of complexity to the configuration of the system. However, this segmentation conforms to the CIS benchmark for security in Kubernetes [141].
- *Benchmarking*: Benchmarking is an important auditing step to check the state of a Kubernetes system against a set of recommendations for configuring Kubernetes.
  Kube-bench is an open-source tool that implements tests documented in the CIS Kubernetes Benchmark [142]. This tool will identify policy misconfigurations and entry points that can be advantageous to attackers.
  Kube-hunter is an open-source tool that supports the Kubernetes Mitre ATT&CK matrix, which standardizes categories and types of attacks as they pertain to Kubernetes [143]. They have a mapping as to what attack techniques Kube-hunter covers in testing. This is a good way of hunting down the most common security weaknesses in a cluster.
  Polaris from Fairwinds is another benchmarking tool that validates best practices in a Kubernetes cluster and allows further customization to deter configuration drift inside a deployment [144]. It can also be used during the CI/CD process as an admission controller to reject or modify workloads that don't adhere to an organization's policies.
  Open-SCAP is a benchmarking tool that is based on an open-source security guide known as SCAP. The security guide is a collaborative effort between many security-first organizations to create a comprehensive, public security guide to promote best practices. Open-SCAP can be used to measure a Kubernetes instance against this SCAP guide [145].
  Center for Internet Security (CIS) is responsible for maintaining a published list of the latest benchmarks

for Kubernetes [141]. These benchmarks are segmented by operating system and include checks for mounts, partitions, updated packages, process hardening, access control, boot setting, various LSM and kernel features, and plenty more [141]. It represents a comprehensive analysis of your system state to confirm a strong security posture.

- *Continuous Update*: Security patches for new vulnerabilities are only applied up to three prior Kubernetes versions [146]. Ensure that steps are taken to upgrade Kubernetes to the latest version when possible.
- *Controlling access to the Kubelet*: By default Kubelets allow unauthenticated access their HTTPS endpoints API [5]. Enable Kubelet authentication and authorization to secure this endpoint.
- *Rotate Credentials*: Important credentials should be complex and have a shorter lifetime to ensure security. Rotate these passwords, tokens, or certificates often to prevent the abuse of a compromised credential.

IX. Open Research Challenges and Future Directions

A. Current Containerization Challenges

In this section, we outline various challenges associated with containers and containerization platforms.

1) Monitoring Limitations: Containerization adds a dynamic infrastructure layer that must be tracked and discovered before it can be monitored. As a result, additional tools must be utilized to automatically discover containers or provision monitoring infrastructure alongside containers. This is necessary because the default monitoring capabilities of a Docker container consist of runtime metrics only. Without a granular tracking method, it is difficult to evaluate performance, transactions, and other important metrics per container/app basis.

2) Platform dependence (WSL): Containers are designed to be platform-independent and provide containerization capabilities that can run on various operating systems. However, some platform-dependent considerations exist, especially when working with Windows Subsystem for Linux (WSL). WSL is a compatibility layer in Windows that enables running a Linux distribution alongside Windows [147]. Although WSL supports running containers and container orchestration systems on Windows, there are limitations and issues compared to running them natively on Linux. While leveraging WSL, issues may arise in performance, file systems, volume mounting, networking, and resource allocation limits. It is worth noting that Microsoft has been actively working on improving the integration of Docker and Kubernetes with WSL, and newer versions of WSL, such as WSL 2, have significantly improved performance and compatibility. For the best containers and container orchestration system experience, it is recommended to run them on a native Linux environment.

3) (Persistent) Storage Complexities: By default, container storage is stateless; storage is provisioned when the container is spun up, and removed when the container shuts down. We can provision stateful, persistent storage for containers in K8s through persistent volumes. These persistent volumes can point to an external or physical storage provider on the host. As persistent storage scales, so does the overall storage complexity. Reservations are required across various storage providers to fulfill business needs, further, if additional data processing or storage in a particular database is required then additional connections will need to be dynamically allocated for those resources. Depending on storage type, allocation size, allocation method, and security, storage allocation may require a range of storage classes for container orchestration systems to function effectively.

4) Flawed Container Images: A security risk within containerization stems from existing security defects in container templates and images. Relying on vulnerability scanners in a container repository is not a fail-proof strategy. Current best-performing vulnerability detecting tools are missing 34% of vulnerabilities [87]. Therefore, deploying a container inside a Kubernetes cluster with a flaw leading to a security breach may result in malicious nodes. We can secure a cluster, but the cloud-native infrastructure will remain vulnerable if the pods are ill-equipped with container images that have not been security-hardened. These faulty containers create another attack surface within the cluster. Without proper limits in place to reduce its impact (limit memory, CPU, and data storage to prevent the malicious node from overloading/DoS-ing the system) as well as limiting its network access, these malicious nodes have a chance at infecting other nodes in the cluster, or the cluster itself. Research must focus on creating a golden image methodology, i.e., a consistent and secure way to harden container images.

B. Open Research Questions

In this section, we provide guidance for future research on the following directions:

1) Evaluating the Performance of Security Tools: The impact of recommended security tools on the overall performance of cloud-native infrastructures should be a future research priority. While a few papers compare security tools in a Kubernetes environment concerning resource utilization, efficiency, and impact on applications, the impact of security tools on the overall infrastructure is not well-studied. A detailed literature review of this class of works into a comprehensive performance evaluation would be a useful resource and help cloud operators, and decision-makers consider what tools an organization should commit to for their infrastructure.

2) Security of Virtualized Networks and Network Service Meshes (NSM): Research focusing on virtualized cloud-native networks and service mesh infrastructures is essential to provide insights into the security of service-based communication. Abstracting service communication into its own plane with a service mesh network (e.g., ISTIO, LinkerD, Cilium) is beneficial to understanding the security requirements of microservices communication and east-west traffic within a cluster. Therefore, research should focus on evaluating and understanding the security of service mesh networking solutions in the context of container orchestration systems.

3) Edge-Cloud Dichotomy and its Effect on Integrated Security: Future research should investigate the integration of containerization and container orchestrators (e.g., Kubernetes-compliant variants for edge computing and IoT environments) for addressing security challenges related to latency, resource constraints, intermittent connectivity, and managing distributed deployments in integrated edge-cloud environments.

4) Cloud-native Observability/Monitoring focusing on Security: Better tooling and techniques for monitoring and observability in containerized environments are under development through technologies like eBPF. Novel observability and monitoring approaches must include exploring methods for real-time performance monitoring, log aggregation, distributed tracing, and efficient troubleshooting of containerized applications without burdening the host infrastructure. Before use in production systems, extensive research is necessary into the security consideration behind such monitoring tool deployments and their effect on cloud-native observability.

C. Notable Future Projects

Next, we provide a brief overview of a few notable future projects on rising new technologies or business cases. Cilium and its Kubernetes sub-project Tetragon focus on real-time security through the extended Berkeley Packet Filters (eBPF) filtering [148]. eBPF kernel module specializes in adding code to the kernel safely, efficiently allowing the kernel to be extended, abstracted, or altered in a manner that will not corrupt the system. It is arguable that secure, flexible, and easy confinement of processes could be implemented through the use of eBPF in place of older kernel abstraction modules and features (e.g., namespaces, cgroups) [149]. eBPF monitoring is a technique used to observe and analyze the behavior of a system by leveraging eBPF. eBPF monitoring involves writing and attaching eBPF programs to specific kernel events or hooks, known as tracepoints, to capture and process data in real time. These programs can collect various metrics and information about the system, such as network traffic, system calls, file-system activity, process behavior, and more. eBPF programs can also be used to filter, aggregate, or transform the collected data before it is made available to user space. KubeArmor is a Kubernetes/cloud Intrusion Detection System (IDS) that combines policy enforcement and observability into one platform [150]. This is a CNCF sandbox project that focuses on runtime protection through the use of eBPF and Linux Security Models (LSM) [150].

X. Conclusions [Deepak: Final Review Needed]

This paper researched existing open-source tools to secure a Kubernetes environment in the context of Authentication & Authorization VII-A, Namespace and Resource Management VII-E, Kubernetes Security Policies VII-F, Network Security VII-G, and Vulnerability Analysis VII-H. The goal of this paper is to provide a comprehensive resource for tools that can be referenced for Kubernetes hardening. Furthermore, this paper can be utilized for a conceptual understanding of Kubernetes and its underlying components in the context of security.

Although Kubernetes remains the most popular container orchestration software, different aspects of Kubernetes security remain volatile. In response, tools are being created and depreciated every month, while there is a lack of documentation that can help choose between these tools for specific use-case. A comprehensive, updating guide to Kubernetes hardening is a needed, but missing piece of documentation.

Future research will include a further analysis of securing a container network and inter-process communication (IPC) security. An analysis of commercially-available tools should be analyzed within its commercial application. Ideally, collecting quantitative data of tools implemented in production-level environments should be used to support the use of one tool over another. These research topics would lead to further valuable insight into Kubernetes hardening.

Acknowledgments [Deepak: Final Review Needed]

The authors would like to thank the following individuals and organizations for their support and contribution to their research.

We would like to thank the Cloud Native Computing Foundation (CNCF), whose cataloging and support of cloud-native projects has enabled us to bring forward effective projects that may have otherwise been overlooked.

We would also like to express my gratitude to the department of Computer and Information Technology at Purdue whose resources and support made this research possible.

Finally, we thank our family and friends, whose unwavering support and encouragement were instrumental to our success.

References

[1] M. Bélair, S. Laniepce, and J.-M. Menaud, "Leveraging kernel security mechanisms to improve container security: A survey," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3339252.3340502

[2] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Applied Sciences*, vol. 9, no. 5, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/931

[3] S. van Kalken. What are namespaces and cgroups, and how do they work? [Online]. Available: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

[4] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[5] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[6] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[7] S. Ghavamnia, T. Palit, and A. Benameur, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," p. 16.

[8] M. Kerrisk. capabilities(7) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html

[9] M. Bauer, "Paranoid penguin: An introduction to novell apparmor," *Linux J.*, vol. 2006, no. 148, p. 13, aug 2006.

[10] Tomoyo. Tomoyo linux. [Online]. Available: https://tomoyo.osdn.jp/about.html.en

[11] N. Sumrall and M. Novoa, "Trusted computing group (TCG) and the TPM 1.2 specification," in *Intel Developer Forum*, vol. 32, 2003.

[12] D. et al. Trusted platform module technology overview. [Online]. Available: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview

[13] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "{Container-IMA}: A privacy-preserving Integrity Measurement Architecture for Containers," 2019, pp. 487–500. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/luo

[14] W. Luo, Q. Shen†, Y. Xia, and Z. Wu†, "Container-ima: A privacy-preserving integrity measurement architecture for containers," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 487–500. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/luo

[15] R. Perez, R. Sailer, L. van Doorn *et al.*, "vTPM: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[16] opensource. What is virtualization. [Online]. Available: https://opensource.com/resources/virtualization

[17] geeksforgeeks. Difference between full virtualization and paravirtualization. [Online]. Available: https://www.geeksforgeeks.org/difference-between-full-virtualization-and-paravirtualization/

[18] D. Barrett and G. Kipper, "1 - how virtualization happens," pp. 3–24, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597495578000011

[19] vmware. Understanding full virtualization, paravirutalization, and hardware assist. [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

[20] VMWare. Vmware esxi server. [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html

[21] Xen. Xen project. [Online]. Available: https://xenproject.org/

[22] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data Center Network Virtualization: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[23] M. Alouane and H. El Bakkali, "Virtualization in Cloud Computing: Existing solutions and new approach," in *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, 2016, pp. 116–123.

[24] M. U. Bokhari, Q. Makki, and Y. K. Tamandani, "A Survey on Cloud Computing," in *Big Data Analytics*, V. B. Aggarwal, V. Bhatnagar, and D. K. Mishra, Eds. Singapore: Springer Singapore, 2018, pp. 149–164.

[25] Q. Duan, Y. Yan, and A. V. Vasilakos, "A Survey on Service-Oriented Network Virtualization Toward Convergence of Networking and Cloud Computing," *IEEE Transactions on Network and Service Management*, vol. 9, no. 4, pp. 373–392, 2012.

[26] H. Alshaer, "An overview of network virtualization and cloud network as a service," *International Journal of Network Management*, vol. 25, no. 1, pp. 1–30, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1882

[27] F. Sierra-Arriaga, R. Branco, and B. Lee, "Security Issues and Challenges for Virtualization Technologies," *ACM Comput. Surv.*, vol. 53, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3382190

[28] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, Security Threats, and Solutions," *ACM Comput. Surv.*, vol. 45, no. 2, mar 2013. [Online]. Available: https://doi.org/10.1145/2431211.2431216

[29] G. Pék, L. Buttyán, and B. Bencsáth, "A Survey of Security Issues in Hardware Virtualization," *ACM Comput. Surv.*, vol. 45, no. 3, jul 2013. [Online]. Available: https://doi.org/10.1145/2480741.2480757

[30] W. Yang and C. Fung, "A survey on security in network functions virtualization," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 15–19.

[31] L. Alhenaki, A. Alwatban, B. Alamri, and N. Alarifi, "A Survey on the Security of Cloud Computing," in *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, 2019, pp. 1–7.

[32] Citrix. What is containerization? [Online]. Available: https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html

[33] docker. Use containers to build, share and run your applications. [Online]. Available: https://www.docker.com/resources/what-container

[34] C. Ltd. What is lxc? [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[35] I. C. Team. Containers vs. virtual machines (vms): What's the difference? [Online]. Available: https://www.ibm.com/cloud/blog/containers-vs-vms

[36] NetApp. What are containers? [Online]. Available: https://www.netapp.com/devops-solutions/what-are-containers/

[37] CNCF. Cloud native landscape. [Online]. Available: https://landscape.cncf.io/

[38] Kubernetes. Container runtime interface (cri). [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cri/

[39] Random-Liu. containerd/cri. [Online]. Available: https://github.com/containerd/cri

[40] CRI-O. Cri-o. [Online]. Available: https://cri-o.io/

[41] Opencontainer. Open container initiative. [Online]. Available: https://opencontainers.org/

[42] Aquasec. 3 types of container runtime and the kubernetes connection. [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-security/container-runtime/

[43] Aquasec. What are container platforms? [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-platforms/container-platforms-6-best-practices-and-15-top-solutions/

[44] RedHat. What is a container registry? [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry

[45] M. Crosby. What is containerd ? [Online]. Available: https://www.docker.com/blog/what-is-containerd-runtime/

[46] H. Haider. Virtualization vs containers: Scalability, portability and resource utilization. [Online]. Available: https://www.replex.io/blog/virtualization-vs-containers-scalability-portability-and-resource-utilization

[47] Redhat. What is container orchestration? [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration

[48] C. Sayfan, *Mastering Kubernetes*, ser. International series of monographs on physics. Packt Publishing Ltd., 2017.

[49] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 05, pp. 46–56, sep 2021.

[50] C. T. Brendan Burns, *Managing Kubernetes*. O'Reilly Media, Inc., 2018.

[51] D. N. An introduction to kubernetes: Learn the basics. [Online]. Available: https://www.whizlabs.com/blog/introduction-to-kubernetes/

[52] M. Zand. Review of container-to-container communications in kubernetes. [Online]. Available: https://thenewstack.io/review-of-container-to-container-communications-in-kubernetes

[53] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225.

[54] M. Al-Ruithe, E. Benkhelifa, and K. Hameed, "A systematic literature review of data governance and cloud data governance," *Personal Ubiquitous Comput.*, vol. 23, no. 5–6, p. 839–859, jan 2018. [Online]. Available: https://doi.org/10.1007/s00779-017-1104-3

[55] X. Zhao, H. Yan, and J. Zhang, "A critical review of container security operations," *Maritime Policy & Management*, vol. 44, no. 2, pp. 170–186, Feb. 2017, publisher: Routledge _eprint: https://doi.org/10.1080/03088839.2016.1253883.

[56] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 418–429. [Online]. Available: https://doi.org/10.1145/3274694.3274720

[57] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia Computer Science*, vol. 175, pp. 157–164, Jan. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705092031704X

[58] M. Bélair, S. Laniepce, and J.-M. Menaud, "Leveraging Kernel Security Mechanisms to Improve Container Security: a Survey," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3339252.3340502

[59] K. Brady, S. Moon, T. Nguyen, and J. Coffman, "Docker Container Security in Cloud Computing," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2020, pp. 0975–0980.

[60] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," in *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, Mar. 2019, pp. 467–470.

[61] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, p. e5668, 2020, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5668.

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

[63] A. Ceccanti, E. Vianello, M. Caberletti, and F. Giacomini, "Beyond X.509: token-based authentication and authorization for HEP," *EPJ Web of Conferences*, vol. 214, p. 09002, 2019, publisher: EDP Sciences. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/abs/2019/19/epjconf_chep2018_09002/epjconf_chep2018_09002.html

[64] B. Ramakrishnan and A. Mahendrakar, "A Scalable Client Authentication & Authorization Service for {Container-Based} Environments," 2015. [Online]. Available: https://www.usenix.org/conference/ucms15/summit-program/presentation/ramakrishnan

[65] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic Policy Generation for {Inter-Service} Access Control of Microservices," 2021, pp. 3971–3988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing

[66] A. Mahajan and T. A. Benson, "Suture: Stitching Safety onto Kubernetes Operators," in *Proceedings of the Student Workshop*, ser. CoNEXT'20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3426746.3434055

[67] A. Beltre, P. Saha, and M. Govindaraju, "KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters," in *2019 IEEE Cloud Summit*, Aug. 2019, pp. 14–20.

[68] A. Beltre, P. Saha, and M. Govindaraju, "Framework for Analysing a Policy-driven Multi-Tenant Kubernetes Environment," in *2021 IEEE Cloud Summit (Cloud Summit)*, Oct. 2021, pp. 49–56.

[69] V. Medel, O. Rana, J. a. Bañares, and U. Arronategui, "Modelling performance &amp; resource management in kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, ser. UCC '16. New York, NY, USA: Association for Computing Machinery, Dec. 2016, pp. 257–262. [Online]. Available: https://doi.org/10.1145/2996890.3007869

[70] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec. 2017, pp. 1–6.

[71] C. Felix, H. Garg, and S. Dikaleh, "Kubernetes security and access management: A workshop exploring security & access features in kubernetes," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 395–396.

[72] H. Zhu and C. Gehrmann, "Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine," in *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, Jan. 2022, pp. 129–137, iSSN: 2155-2509.

[73] M. Ahmadvand, A. Pretschner, K. Ball, and D. Eyring, "Integrity Protection Against Insiders in Microservice-Based Infrastructures: From Threats to a Security Framework," in *Software Technologies: Applications and Foundations*, ser. Lecture Notes in Computer Science, M. Mazzara, I. Ober, and G. Salaün, Eds. Cham: Springer International Publishing, 2018, pp. 573–588.

[74] L. Csikor, C. Rothenberg, D. P. Pezaros, S. Schmid, L. Toka, and G. Rétvári, "Policy Injection: A Cloud Dataplane DoS Attack," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 147–149. [Online]. Available: https://doi.org/10.1145/3234200.3234250

[75] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang, "ProSPEC: Proactive Security Policy Enforcement for Containers," in *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 155–166. [Online]. Available: https://doi.org/10.1145/3508398.3511515

[76] P. Mytilinakis, "Attack methods and defenses on Kubernetes," Master's thesis, University of Piraeus, Jun. 2020. [Online]. Available: http://dx.doi.org/10.26267/unipi_dione/311

[77] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An Analysis and Empirical Study of Container Networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, Apr. 2018, pp. 189–197.

[78] D. D'Silva and D. D. Ambawade, "Building A Zero Trust Architecture Using Kubernetes," in *2021 6th International Conference for Convergence in Technology (I2CT)*, Apr. 2021, pp. 1–8.

[79] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network Policies in Kubernetes: Performance Evaluation and Security Analysis," in *2021 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, Jun. 2021, pp. 407–412, iSSN: 2575-4912.

[80] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the Security Implications of Kubernetes Networking," *IEEE Security Privacy*, vol. 19, no. 5, pp. 46–56, Sep. 2021, conference Name: IEEE Security Privacy.

[81] X. Nguyen, "Network isolation for Kubernetes hard multi-tenancy," Aug. 2020, accepted: 2020-08-23T17:12:47Z. [Online]. Available: https://aaltodoc.aalto.fi:443/handle/123456789/46078

[82] F. Hussain, W. Li, B. Noye, S. Sharieh, and A. Ferworn, "Intelligent Service Mesh Framework for API Security and Management," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Oct. 2019, pp. 0735–0742, iSSN: 2644-3163.

[83] A. Kurbatov, "Design and implementation of secure communication between microservices."

[84] D. Huang, H. Cui, S. Wen, and C. Huang, "Security Analysis and Threats Detection Techniques on Docker Container," in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, Dec. 2019, pp. 1214–1220.

[85] J. Wenhao and L. Zheng, "Vulnerability Analysis and Security Research of Docker Container," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, Sep. 2020, pp. 354–357.

[86] M. Rangta, "Tools for Security Auditing and Hardening in Microservices Architecture," Jan. 2022, accepted: 2022-02-06T18:01:04Z. [Online]. Available: https://aaltodoc.aalto.fi:443/handle/123456789/112846

[87] O. Javed and S. Toor, "Understanding the Quality of Container Security Vulnerability Detection Tools," *arXiv:2101.03844 [cs]*, Jan. 2021, arXiv: 2101.03844. [Online]. Available: http://arxiv.org/abs/2101.03844

[88] A. Osman, S. Hanisch, and T. Strufe, "SeCoNetBench: A modular framework for Secure Container Networking Benchmarks," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, Jun. 2019, pp. 21–28.

[89] K. A. Torkura, M. I. H. Sukmana, F. Cheng, and C. Meinel, "CAVAS: Neutralizing Application and Container Security Vulnerabilities in the Cloud Native Era," in *Security and Privacy in Communication Networks*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, pp. 471–490.

[90] J. Mahboob and J. Coffman, "A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2021, pp. 0529–0535.

[91] M. Agrawal and K. Abhijeet, "Security Audit of Kubernetes based Container Deployments: A Comprehensive Review," vol. 07, no. 06, p. 6, 2020.

[92] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," *arXiv:2006.15275 [cs]*, Jun. 2020, arXiv: 2006.15275. [Online]. Available: http://arxiv.org/abs/2006.15275

[93] D. Muresu, *Investigating the security of a microservices architecture : A case study on microservice and Kubernetes Security*, 2021. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-302579

[94] M. Colman, "Containers and Kubernetes: Security is not an Afterthought," *ITNOW*, vol. 64, no. 1, pp. 44–45, Mar. 2022. [Online]. Available: https://doi.org/10.1093/itnow/bwac023

[95] S. I. Shamim, "Mitigating security attacks in kubernetes manifests for security best practices violation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1689–1690. [Online]. Available: https://doi.org/10.1145/3468264.3473495

[96] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," *Applied Sciences*, vol. 9, no. 5, p. 931, Jan. 2019, number: 5 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/931

[97] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," 09 2020, pp. 58–64.

[98] Kubernetes. Kep 2579. [Online]. Available: https://github.com/kubernetes/enhancements/issues/2579
[99] Kyverno. Kyverno. [Online]. Available: https://github.com/kyverno/kyverno/
[100] Kyverno. Kubernetes native policy management. [Online]. Available: https://kyverno.io/
[101] O. P. A. Gatekeeper. Policy-based control for cloud native environments. [Online]. Available: https://www.openpolicyagent.org/docs/
[102] O. P. A. Github. Opa gatekeeper introduction. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/
[103] Tigera. Install calico for policy and flannel (aka canal) for networking. [Online]. Available: https://projectcalico.docs.tigera.io/getting-started/kubernetes/flannel/flannel
[104] Kubernetes. Networking and network policy. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/addons/#networking-and-network-policy
[105] Anchore. Open source container security. [Online]. Available: https://anchore.com/opensource/#tools
[106] Anchore. Syft. [Online]. Available: https://github.com/anchore/syft
[107] Anchore. Grype. [Online]. Available: https://github.com/anchore/grype
[108] M. Burillo. 29 docker security tools compared. [Online]. Available: https://sysdig.com/blog/20-docker-security-tools/
[109] R. Hat. What is clair? [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-clair
[110] R. Hat. What is clair. [Online]. Available: https://quay.github.io/clair/
[111] dockscan. dockscan. [Online]. Available: https://github.com/kost/dockscan
[112] Checkmarx. Kics. [Online]. Available: https://docs.kics.io/latest/integrations/
[113] Bridgecrew. What is checkov? [Online]. Available: https://www.checkov.io/1.Welcome/What%20is%20Checkov.html
[114] Microsoft. Identity and access documentation. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/identity/identity-and-access
[115] FairWindsOps. Rbac lookup. [Online]. Available: https://github.com/FairwindsOps/rbac-lookup
[116] Cerbos. Cerbos. [Online]. Available: https://github.com/cerbos/cerbos
[117] K-rail. K-rail. [Online]. Available: https://github.com/cruise-automation/k-rail
[118] MagTape. Magtape. [Online]. Available: https://github.com/tmobile/magtape
[119] F. W. Maarten van der Slik, "Validating the replacement filtering features of popular alternative admission controllers for pod security policies," *https://rp.os3.nl/2020-2021/p76/report.pdf*, vol. 2, p. 14, aug 2021.
[120] R. C. Godoy. Why non-root containers are important for security. [Online]. Available: https://engineering.bitnami.com/articles/why-non-root-containers-are-important-for-security.html
[121] K. Bruner. Better kubernetes security with open policy agent (opa) - part 1. [Online]. Available: https://cloud.redhat.com/blog/better-kubernetes-security-with-open-policy-agent-opa-part-1
[122] datadoghq. Container's root filesystem is mounted as read only. [Online]. Available: https://docs.datadoghq.com/security_platform/default_rules/cis-docker-1.2.0-5.12/
[123] V. Jain, B. Singh, M. Khenwar, and M. Sharma, "Static vulnerability analysis of docker images," *IOP Conference Series: Materials Science and Engineering*, vol. 1131, no. 1, p. 012018, apr 2021. [Online]. Available: https://doi.org/10.1088/1757-899x/1131/1/012018
[124] Docker. Content trust in docker. [Online]. Available: https://docs.docker.com/engine/security/trust/
[125] Notary. Notary. [Online]. Available: https://github.com/notaryproject/notary
[126] S. Ruffell. Linux security modules (lsms) vs secure computing mode (seccomp). [Online]. Available: https://www.starlab.io/blog/linux-security-modules-lsms-vs-secure-computing-mode-seccomp
[127] Docker. Seccomp security profiles for docker. [Online]. Available: https://github.com/notaryproject/notary
[128] V. Venugopal. Guide to kubernetes ingress network policies. [Online]. Available: https://cloud.redhat.com/blog/guide-to-kubernetes-ingress-network-policies
[129] tenable. Terrascan by tenable. [Online]. Available: https://runterrascan.io/
[130] Indendi. Indeni cloud rail. [Online]. Available: https://cloudrail.app/
[131] iacsecurity. tool-compare. [Online]. Available: https://github.com/iacsecurity/tool-compare
[132] K. Dashboard. Kubernetes dashboard. [Online]. Available: https://github.com/kubernetes/dashboard
[133] G. Labs. Dashboards. [Online]. Available: https://grafana.com/grafana/dashboards/?search%3Dkubernetes
[134] Elastic. The elastic stack. [Online]. Available: https://www.elastic.co/elastic-stack/
[135] Prometheus. Prometheus. [Online]. Available: https://prometheus.io/
[136] Sysdig. Falco. [Online]. Available: https://github.com/falcosecurity/falco
[137] FluentD. Build your unified logging layer. [Online]. Available: https://www.fluentd.org/
[138] CNCF. Cloud native observability microsurvey: Prometheus leads the way, but hurdles remain to understanding the health of systems. [Online]. Available: https://www.cncf.io/wp-content/uploads/2022/03/CNCF_Observability_MicroSurvey_030222.pdf
[139] HashiCorp. Kubernetes. [Online]. Available: https://www.vaultproject.io/docs/platform/k8s
[140] CyberArk. Cyberark secrets provider for kubernetes. [Online]. Available: https://docs.cyberark.com/Product-Doc/OnlineHelp/AAM-DAP/11.2/en/Content/Integrations/Kubernetes_deployApplicationsConjur-k8s-Secrets.htm
[141] C. for Internet Security (CIS). Making the connected world a safer place. [Online]. Available: https://www.cisecurity.org/
[142] Kube-Bench. Kube-bench. [Online]. Available: https://github.com/aquasecurity/kube-bench
[143] Aquasec. Aqua kube-hunter. [Online]. Available: https://github.com/aquasecurity/kube-hunter
[144] Fairwinds. Polaris. [Online]. Available: https://github.com/FairwindsOps/polaris
[145] OpenScap. Openscap audit, fix, and be happy. [Online]. Available: https://www.open-scap.org/
[146] StackRox. Kubernetes security 101: Risks and 29 best practices. [Online]. Available: https://www.stackrox.io/blog/kubernetes-security-101/
[147] Microsoft. Windows subsystem for linux documentation. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/
[148] Cilium. Tetragon. [Online]. Available: https://github.com/cilium
[149] W. Findlay, A. Somayaji, and D. Barrera, "bpfbox: Simple Precise Process Confinement with eBPF," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*. Virtual Event USA: ACM, Nov. 2020, pp. 91–103. [Online]. Available: https://dl.acm.org/doi/10.1145/3411495.3421358
[150] AccuKnox. Runtime protection for kubernetes & other cloud workloads. [Online]. Available: https://kubearmor.io/

Benjamin Wagrez is an Enterprise Technology Experienced Consultant at West Monroe where he specializes in infrastructure automation, cloud infrastructure, and Kubernetes. Previously he has worked as a DevOps engineer at 3M specializing in AWS. Ben has a B.S in Network Engineering Technology from Purdue University where he graduated with honors and distinction. He is a Certified Kubernetes Administrator (CKA) as well as a certified CompTIA Secure Infrastructure Specialist. He also holds a Terraform associate certification and has various certifications within the Microsoft Azure and AWS clouds.

Deepak Nadig is currently an Assistant Professor in the Department of Computer and Information Technology at Purdue University. From 2009 to 2015, Dr. Nadig was the Director of Technology and Research at SOLUTT Corporation, India. He was instrumental in leading the networking and wireless operations in 4G/LTE and multi-gigabit wireless technologies. His research focuses on the interplay between Software Defined Networks and Network Functions Virtualization for developing secure and intelligent next-generation networks. His research interests are in Computer Networks, Software Defined Networks, Network Functions Virtualization, Cloud-native Infrastructure, Network Security and AI/ML applications to networking. Dr. Nadig has a Ph.D. in Computer Engineering from the University of Nebraska-Lincoln (UNL).