Configuration Parameters that Facilitate Security Attacks for Kubernetes Pods

Abstract—**While Kubernetes allows organizations to rapidly deploy software, Kubernetes-related security attacks can cause serious consequences for organizations. A systematic investigation of configuration parameters that facilitate security attacks can aid practitioners in identifying configuration parameters that need to be avoided in order to secure a Kubernetes-based deployment infrastructure. To that end, we investigate configuration parameters that facilitate attacks for pods, i.e., fundamental deployment units in Kubernetes. Our approach is informed by gaining an understanding of the states associated with the pod lifecyle. Using our approach, we identify 6 attacks unique to Kubernetes that can be facilitated using combinations of 21 configuration parameters. By conducting quantitative analysis with 2,039 open source software (OSS) Kubernetes manifests, we find one manifest, on average, to contain 15.9% of the configuration parameters needed to conduct security attacks. Based on our findings, we provide recommendations for toolsmiths on how to enhance Kubernetes-related static analysis tools by providing context on how combinations of configuration parameters can be used to facilitate pod-related attacks.**

Index Terms—**configuration, container orchestration, devops, empirical study, kubernetes, security**

I. Introduction

In order to rapidly deploy software applications to end-users, organizations use container orchestration, the practice of automatically managing containers at scale [18]. Use of Kubernetes, a popular tool to implement the practice of container orchestration, has yielded benefits for organizations, such as Capital One and OpenAI. For example, in the case of Capital One, with Kubernetes the software deployment rate "*increased by several orders of magnitude*" [20]. OpenAI uses Kubernetes to manage containers that run on more than 2,500 servers, yielding lower costs for experimentation and maintenance [21]. Documented evidence of such benefits has helped Kubernetes to become the most popular tool to implement the practice of container orchestration, with an expected market size of 7.8 billion USD by 2030 [30].

Despite the above-mentioned benefits, Kubernetes-based software deployment infrastructure can be susceptible to security attacks. For example, a Kubernetes-related configuration parameter facilitated a security attack called cryptojacking. Cryptojacking is the attack of using a computing resource to stealthily mine cryptocurrency without the user's awareness [37]. As part of this attack, malicious users gained access to the containers managed with Kubernetes, and used those containers to mine cryptocurruencies. Such examples of cryptojacking, are consequential as each of these attacks can incur thousands of dollars in unwanted expenditures, as it happened for the Telnet cryptojacking attack [1].

The above-mentioned example related to cryptojacking showcases the need for adopting a secure provisioning process for container orchestration. In the case of Kubernetes, a pod is the most fundamental unit for performing container orchestration [23]. Pods groupify multiple containers together, and manage the entire provisioning process, for example, specifying the CPU and memory that will be allocated for the containers, the image source that the containers will be using, and the system-level privileges that the containers may leverage [19], [23]. In order to facilitate automated management of containers, pods provide a wide range of configuration parameters using which Kubernetes users provision and manage the behavior of containers [23]. As such, in order to secure the container orchestration process with Kubernetes, practitioners must identify pod-related configuration parameters that can facilitate security attacks.

Identifying pod-related configuration parameters that facilitate security attacks pose the following challenges:

- **Stateful nature of configuration parameters**: Configuration parameters of pods are stateful, i.e., certain configuration parameters are only activated at certain states of the pod lifecyle. An automated approach aimed at finding configuration parameters that facilitate security attacks must account for the lifecycle states and their corresponding configuration parameters. (*Addressed in Section III-A2*)
- **Security requirements for pods**: Kubernetes pods have unique properties that necessitates accounting for security requirements unique to pods. In order to find configuration parameters that facilitate security attacks, security requirements unique to pods need to be identified. (*Addressed in Section III-A3*)
- **Exploration of configuration parameters**: Kubernetes allows multiple configurations to provision pods, each of which have multiple parameters. Manual exploration of all of these combinations of configuration parameters is practically impossible, necessitating an automated approach. (*Addressed in Section III-A4*)

The above-mentioned discussion highlights the need for a systematic investigation to determine which configuration pa-

[1]https://thenewstack.io/cryptojacking-free-money-for-attackers-huge-cloud-bill-for-you/#

rameters facilitate security attacks for Kubernetes pods. Such investigation could yield an approach that address the above-mentioned challenges in order to derive relevant configuration parameters. While empirical research related to Kubernetes have addressed topics related to quality assurance [6], [28], [33], there is a lack of investigation on what configuration parameters facilitate security attacks. Such an investigation can be helpful for (i) toolsmiths to enhance detection of pod-related security weaknesses in Kubernetes manifests; and (ii) researchers to understand what configuration parameters facilitate security attacks.

Accordingly, we answer the following research questions:

- ***RQ1*: What configuration parameters facilitate security attacks for Kubernetes pods?**
- ***RQ2*: How frequently do identified configuration parameters appear in Kubernetes manifests?**
- ***RQ3*: What states in the pod lifecycle map with security attacks for Kubernetes pods?**

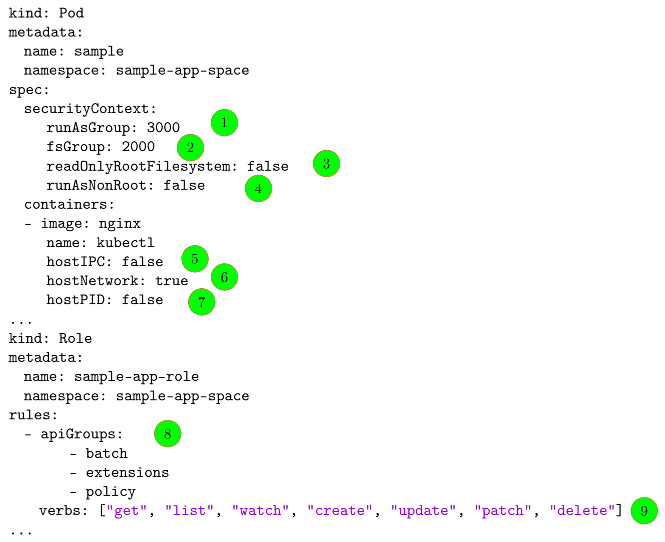
We answer our research questions by gaining an understanding of the lifecycle states of a Kubernetes pod, and using that understanding to represent the lifecycle states using a finite state machine (FSM). Next, we use the constructed FSM to identify a set of configuration parameters that can be used to conduct security attacks. We also quantify the frequency of the identified configuration parameters using an empirical analysis with 2,189 open source software (OSS) Kubernetes manifests. Dataset and source code used in our paper is available online [3].

Contributions: We list our contributions as follows:

- A list of 21 configuration parameters combinations of which facilitate 6 security attacks for Kubernetes pods (Section III-B);
- A mapping between of pod-related configuration parameters and Kubernetes-related security attacks (Section III-B); and
- A mapping between states and pod-related security attacks (Section IV-B2).

II. Motivating Example

We motivate our paper using Figure 1, where we present an example Kubernetes manifest. The manifest is used to specify configuration parameters for a pod called ‘sample’ with ‘nginx’ container images. We also observe the manifest to include configuration parameters for role-based access control (RBAC) using the `kind: Role` object. In the case of configuration parameters, such as `name` and `namespace`, a practitioner can assign any strings so that the pod ‘sample’ is deployed with adequate RBAC configuration parameters. However, prior to execution, in the case of nine configurations, as indicated with the green circles, the practitioner must determine if one or a combination of these configuration parameters can yield security attacks. Of these 9 configurations, (i) 5 are Boolean,



```
kind: Pod
metadata:
  name: sample
  namespace: sample-app-space
spec:
  securityContext:
     runAsGroup: 3000         1
     fsGroup: 2000          2
     readOnlyRootFilesystem: false    3
     runAsNonRoot: false        4
  containers:
  - image: nginx
     name: kubectl
     hostIPC: false        5
     hostNetwork: true       6
     hostPID: false         7
...
kind: Role
metadata:
  name: sample-app-role
  namespace: sample-app-space
rules:
  - apiGroups:         8
        - batch
        - extensions
        - policy
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]  9
...
```

Fig. 1: An example to motivate our empirical study.

(ii) 2 are of type Integer, each yielding 2^{32} possible values, (iii) one configuration with 3 values, and (iv) 1 configuration with 7 possible strings. To determine if these 9 configurations cause attacks, a user has to manually explore all 1.2×10^{22} possible combinations for the 9 configurations.

The example presented in Figure 1 showcases the challenges associated with identifying configuration parameters that can facilitate security attacks for Kubernetes pods:

- According to the example in Figure 1, there are 1.2×10^{22} combinations of configuration parameters to explore, which is practically impossible with manual analysis;
- For each of these 1.2×10^{22} combinations practitioners must have an understanding of the stateful nature of configurations where certain configuration parameters are applicable to certain states of the pod lifecycle. For example, `hostIPC`, `hostNetwork`, and `hostPID` in Figure 1 are applicable for a state called ‘image pulled from registry’ [19]. For an average Kubernetes user, basic operation of Kubernetes poses challenges [22], which makes the understanding of the stateful nature of configuration parameters even more challenging; and
- There is a lack of understanding on what security-related requirements pods need to abide, which is pivotal to derive configuration parameters facilitate security attacks.

III. RQ1: Configuration Parameters that Facilitate Security Attacks for Pods

Methodology and results for RQ1 are discussed in this section.

A. Methodology

We provide the methodology to answer RQ1 by *first* providing necessary background on pods. *Second*, we describe our

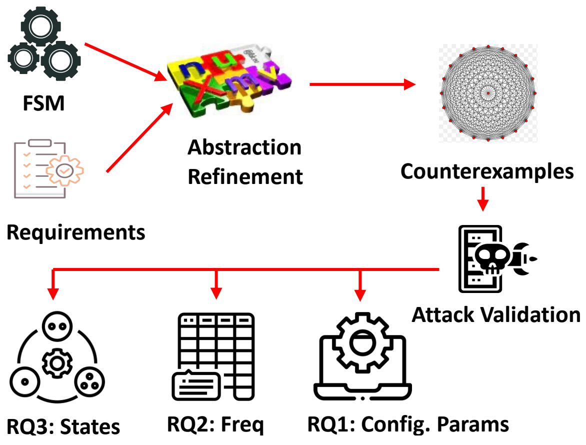


Fig. 2: An overview of our methodology.

procedure of constructing and using an FSM to identify configuration parameters that facilitate security attacks for pods. An overview of our methodology is presented in Figure 2.

1) Background on Kubernetes Pods: Kubernetes enables practitioners to manage container-based applications at scale [19] [7]. Practitioners can install Kubernetes on-premise, on cloud platforms, or a combination of both. A Kubernetes installation is colloquially referred to as a Kubernetes cluster [19]. Kubernetes allows namespaces, which provide a mechanism to isolate groups of resources within a Kubernetes cluster. A pod is the most fundamental unit of a Kubernetes cluster [19]. A pod groups one or more containers with shared network and storage resources.

2) Representation of the Pod Lifecycle: A pod undergoes five phases. Each phase includes multiple states. These phases are:

- *Pending*: In this phase, a pod is created using the 'kubectl' utility. Manifests developed by practitioners are executed by the 'kubectl' utility to initiate the pod creation process.
- *Starting*: In this phase, configuration parameters obtained from the manifests are inspected automatically using a set of syntax-related validation checks. If the validation checks pass, then pod configuration parameters are stored in 'etcd', which is a database that uses a key-value mechanism to store all pod-related configuration parameters [19]. The container image needed for the pods is also pulled by using the container runtime engine.
- *Running*: In this phase, the pod specified with the manifest can communicate with other pods in the same node or in a different node.
- *Failed*: In this phase, if one or more containers are terminated, then the pod to which the container(s) belongs to fail.
- *Terminated*: In this phase, the lifecycle of a pod ends, and all the resources allocated to the pod is released.

The first step towards answering RQ1 is to represent the above-mentioned phases in a manner so that the states that belong to each of the five phases, and their corresponding configuration parameters are accounted. We use a FSM to represent the lifecycle of a pod. We select a FSM-based representation because (i) pod-related configurations are stateful, i.e., certain configurations are exhibited in certain phases; and (ii) the lifecycle of a pod can be represented as an FSM as it has starting and ending states, where transitions between states occur due to certain conditions. The FSM can be represented using the following tuple: $<\Gamma, \theta, \psi, \beta, \tau>$, where Γ represents a finite set of input variables, θ represents a set of output variables, ψ represents a set of state variables, β represents a set of initial states, and τ represents a set of variables assignments in ψ defining the transition relationships. Our FSM-based approach alleviates the challenge of accounting for the stateful nature of configuration parameters through the usage of state variables, ψ, and transition conditions, τ, for the derived FSM.

We use Figure 3 to further illustrate our FSM construction process. For the sake of simplicity, we provide a subset of the state transitions that are possible for the 'Starting' phase. In this particular FSM, the set ψ includes five states, where β is `request-initiated`. τ represents the four transitioning conditions in forms of variable assignments, namely, 'authorized bootstrapping', 'valid API request', 'valid admission controller', and 'unsuccessful authorization and authentication'. The input variables are the configuration parameters along with other variables that are applicable for state. For example, in the case of 'request-initiated', example configuration parameters that we use as input variables are 'default namespace', 'hostIPC', and 'hostPID'.

3) Derivation of Pod-related Security Requirements: We address the challenge of identifying pod-related security requirements applicable by applying grey literature review [12], where we analyze Internet artifacts. *First*, we use the Google search engine in incognito mode to collect the top 300 search results for three search strings: 'kubernetes pod security requirements', 'kubernetes security guidelines', and 'kubernetes pod security rules'. Initially, we start with 'kubernetes pod security requirements', but later add 'kubernetes security guidelines' and 'Kubernetes pod security rules' as from our exploration we observe 'rules' and 'guidelines' to be synonymously used with 'requirements'. *Second*, we apply the following filtering criteria (i) exclude artifacts not written in English, (ii) exclude artifacts published before 2016, as the initial version of Kubernetes was released in 2016 [23], (iii) exclude artifacts that are duplicates, and (iv) include artifacts that describe at least one pod-related security guideline. *Third*, from our search results, we obtain 21 Internet artifacts using which we apply open coding [31] to derive necessary requirements.

We identify 9 requirements: 'any container running in a pod must specify resource limits', 'containers with unnecessary privilege cannot be executed inside a pod', 'images with incorrect configurations can not be pulled from an unauthorized registry', 'unnecessary permission to host file system need to be revoked', 'restrict malicious users in obtaining

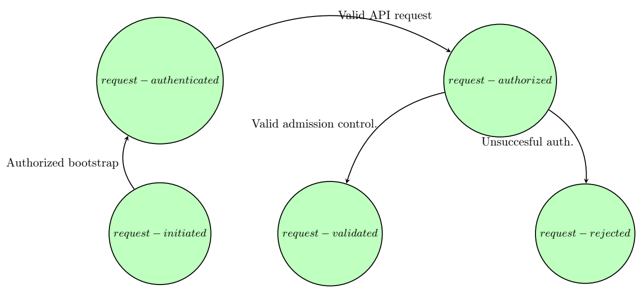


Fig. 3: A finite state machine representing the a subset of the states related to the 'Starting' phase.

secrets from the container inside a pod', 'admission controller must be enabled for pods', 'network policies must be enabled for pods', 'TLS encryption must be enabled for pod-related communication', and 'restrict permission to read/watch secret'.

4) Abstraction Refinement: We take inspiration from the counterexample guided abstraction refinement (CEGAR) approach [10] to refine our FSM in order to derive necessary configuration parameters. In the case of a CEGAR-based approach, the proposed approach verifies if a concrete model Ξ satisfies requirement Υ by creating an abstract model Ξ_a from ψ, and testing if Ξ_a complies with Υ. If the compliance is concluded, then Ξ is deemed to satisfy Υ. Otherwise, a counterexample κ is generated. If κ is realizable in ψ. a failure in verification is found. If κ is not realized, then Ξ_a is regenerated to exclude κ, and the process is repeated until either ψ satisfies Υ or a realizable counterexample is found.

We use the requirement 'any container running in a pod must specify resource limits' as an example to illustrate the CEGAR-based approach. Our approach will first verify if our FSM is compliant with the requirement. If not, then a counterexample will be generated. Next, our approach will inspect if the generated counterexample is realizable, i.e., if the counterexample contains state variables that are only applicable for pods. If yes, then the counterexample is valid and used for further analysis. Otherwise, the counterexample is discarded. This process is repeated until the requirement is verified by the FSM or a realizable counterexample is generated.

For implementing CEGAR-based abstraction refinement, we use NuXMV [9]. NuXMV is a tool used to perform software verification by leveraging SAT solvers [9]. As input NuXMV expects a program developed in the SMV language with the following items: (i) requirement that need to be verified using linear temporal logic formulas [10]; (ii) the FSM implemented as state variables; and (iii) applicable transition conditions for the FSM [9]. Upon execution, NuXMV will use SAT solvers to identify a counterexample that violates the provided requirement. The use of SAT solvers accounts for the challenge of exploring multiple combinations of configuration parameters automatically. In our case, we develop a NuXMV program with the state variables and relevant transition conditions for our FSM. We execute the developed NuXMV program 9 times for each of the 9 logical formulas. These logical formulas are encoded representations of the 9 requirements obtained in Section III-A3.

5) Counterexample Generation: Using NuXMV, we obtain counterexamples for each violated requirement listed in Section III-A3. Execution for each of the 9 requirements resulted in generation of counterexamples. Each generated counterexample includes the following items: (i) states of the pod lifecycle that changed prior to generating the counterexample, (ii) the state and transition conditions that led to the generation of the counterexample, and (iii) encoded configuration parameters represented as state variables. Inclusion of a configuration parameter within a counterexample is indicative of facilitating an attack, however, whether or not the configuration parameter facilitates an attack is subject to further validation.

6) Attack Validation: We determine configuration parameters that facilitate an attack by examining if the configuration parameters included in a counterexample, are responsible to trigger a security attack. In order to perform the examination, we execute six attacks unique to pods that are described in this section. Prior to describing the attacks, we describe the threat model, followed by a description of the execution environment where we conducted the attacks.

Threat Model: We assume that Kubernetes manifests are developed without malicious intent, but still can include configuration parameters, which can cause security attacks. A malicious user—a Kubernetes user who owns a pod and wants to compromise other pods—attempts to launch attacks against Kubernetes-based installations by leveraging one or multiple configuration parameters. The goal of the malicious user is to (a) gain unauthorized access, and/or (b) disrupt availability for any Kubernetes-based container infrastructure. If successful, the malicious user may also perform other pernicious attacks including crypto-mining attacks and stealing intellectual property. Disruption in availability can cause large-scale outages for end-users.

Execution Environment: We conduct all attacks on the MacBook with no connection to external networks. We use the `kubeadm` utility that provides a multi-node Kubernetes cluster, which includes a microservice application called 'microservice-demo' [13]. We use Vagrant to set up three nodes, which are virtual machines running on a MacBook with Intel Core i7, 8 core CPU, and 16GB of memory. For each attack, access to a valid 'kubeconfig' file and the ability to use the 'kubectl CLI' utility are assumed to be available.

Attack#1-*Access cluster secrets*: (i) *create malicious pod*: A user creates a malicious pod with an unscanned container image and missing admission controller [16] [26]. The pod's configuration parameters are: active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, auto mounted token, privileged `securityContext`, inactive read-only for root filesystem, capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser`. This allows to communicate with a remote machine administered by the user. (ii) *Listening*: The user uses ncat [25] in order to listen the IP address and port of the malicious pod to connect the pod to get a remote shell from a remote machine [32]. (iii) *Access service account token*: The user accesses the underlying host to get the service account token of the default `serviceAccount` [32]. (iv) *Install kubectl*: The user accesses the service account token of default `serviceAccount` as privileged service account with a privileged role. With this access, the user installs `kubectl` in order to send a request to the Kubernetes API server with the service account token. (v) *Access secrets*: To read cluster secrets, the user leverages missing admission controller and creates a new privileged pod with the token of a privileged service account.

Attack#2-*Dashboard maneuver*: (i) *Get access to privileged pod*: A user creates a pod with the following: escalated child process, inactive read-only for root filesystem, capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser`. The user then gets access to a privileged pod with missing admission controller [16] [26]. (ii) *Access underlying host*: The user accesses the underlying host with active `hostNetwork`, active `hostIPC`, active `hostPID`, privileged `securityContext`, auto mounted token, and active `hostPath` to get the service account token of default `serviceAccount` [32]. (iii) *Dashboard access*: The user accesses the service account token of default `serviceAccount`. With this token and a privileged default `serviceAccount`, the user creates a role with privileged role binding, which is later used to access the Kubernetes dashboard.

Attack#3-*Database tampering*: (i) *create malicious pod*: a user creates a malicious pod with an unscanned container image and missing admission controller [16] [26]. The user creates the pod with the following configuration parameters: active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, active `hostPath`, auto mounted token, privileged `securityContext`, inactive read-only for root filesystem, default namespace, capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser`. (ii) *listening*: The user uses ncat [25] in order to listen the IP address and port of the malicious pod to connect the pod to get a remote shell from a remote machine [32]. (iii) *access Redis database*: The user uses active `hostNetwork`, missing network policy, and default namespace to establish a connection with a Redis data storage in port 6379. (iv) *tamper database*: This enables the user to update the data in the Redis database as Redis does not require any authentication by default [29] [36]. The user can modify or delete sensitive information from the Redis database. As a result, the pods may get tampered data.

Attack#4-*Denial of service (DOS)*: (i) *create malicious pod*: a user creates a malicious pod with missing admission controller and missing resource limit [16] [26]. The pod is created with: active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, active `hostPath`, auto mounted token, privileged `securityContext`, inactive read-only for root filesystem, and capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser` [32]. (ii) *listening*: The user uses ncat [25] to listen the IP address and port of the malicious pod to connect the pod to get a remote shell from a remote machine [32]. (iii) *DOS execution*: The user runs a malicious program inside the running container of the pod with inactive read-only for root filesystem and missing resource limit. This allows the pod to exhaust allocated CPU and memory, and disrupt availability by causing a DOS attack.

Attack#5-*Etcd takeover*: (i) *get access to privileged pod*: a user gets access to a privileged pod that starts with missing admission controller [16] [26], which is created in the control plane. The user creates the pod with the following configuration parameters: active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, privileged `securityContext`, auto mounted token, inactive read-only for root filesystem, active `hostPath`, capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser`. (ii) *listening*: The user uses ncat [25] to listen the IP address and port of the malicious pod to connect the pod to get a remote shell from a remote machine [32]. (iii) *access underlying host*: The user can access the underlying host with active `hostNetwork`, active `hostIPC`, active `hostPID`, privileged `securityContext`, auto mounted token, and active `hostPath` to get the service account token of the default `serviceAccount` [32]. (iv) *access service account token*: The user installs `kubectl` tool from the container using inactive read-only for root filesystem. The user can send a request to the Kubernetes API server with the service account token of privileged service account. (v) *install etcd client*: The user installs etcd client and can access the key and certificate for etcd using active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, inactive `runAsNonRoot`, privileged `securityContext`, inactive read-only for root filesystem, and capability abuse. (vi) *access etcd*: This enables the user to use the key and certificate to access etcd.

Attack#6-*Pod disruption*: (i) *access privileged pod*: A user gets access to a privileged pod that starts with missing admission controller [16] [26]. The user creates the pod with the following configuration parameters: active `hostNetwork`, active `hostIPC`, active `hostPID`, escalated child process, privileged `securityContext`, auto mounted token, inactive read-only for root filesystem, active `hostPath`, capability abuse, inactive `runAsNonRoot`, and inactive `runAsUser`. (ii) *access service account token*: the user accesses the underlying host to get the service account token of default `serviceAccount` [32]. The user installs `kubectl` tool from the container using inactive read-only for root filesystem. The user sends a request to the Kubernetes API server with the obtained service account token. (iii) *privileged role binding*: with the obtained service account token, the user creates a privileged service account using a role with privileged role binding. (iv) *pod disruption*: the user creates a new malicious privileged pod with the service account token, and run the pod as a cryptominer.

B. Answer to RQ1

In this section, we answer "**RQ1: What configuration parameters facilitate security attacks for Kubernetes pods?**" We identify 21 configuration parameters, combinations of which are required to conduct 6 pod-related attacks. Table I provides a mapping between pod-related security attacks and the derived configuration parameters. For example, executing the access cluster secrets attack requires 14 configuration parameters. The 11 configuration parameters that are common across all attacks are: escalated child process, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPID`, missing admission controller, privileged `securityContext`, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, and auto mounted token. Examples of configuration parameters are presented in Figure 4. Definition of each configuration parameter is provided below.

Implications of each attack is provided in Table II. The 'Validated' column shows that all attacks have been validated. Whether or not confidentiality, integrity, or availability, is violated is presented in the 'Violated' column. Finally, the 'Consequence' column represents the consequence of each attack with respect to securing Kubernetes-based clusters.

I. Active `hostIPC` - This provides a container within a pod with the privilege to intercept all inter-process communications (IPCs) of the host machine.

II. Active `hostNetwork` - This activates `hostNetwork`. This configuration parameter allows applications to ping and intercept all network interfaces of the host machine.

III. Active `hostPath` - This allows mounting of a file or a directory from the host node filesystem into a pod with `hostPathEnabled`.

IV. Active `hostPID` - This activates `hostPID` that allows a pod to access the namespace and find processes running on the host.

V. Auto mounted token - This allows a pod to automatically activate tokens used for service accounts inside the pod. These tokens are used to authenticate requests from processes within the cluster to the Kubernetes API server.

VI. Capability abuse - This activates Linux capabilities. This category includes configuration parameters that allow a pod to gain root-level access. This category includes two sub-categories: `CAP_SYS_ADMIN` and `CAP_SYS_MODULE`.

VII. Default namespace - This activates usage of the default namespace. The configuration allows a pod to be deployed in a shared virtual cluster.

VIII. Default `serviceAccount` - This allows a pod to use the default service account.

IX. Escalated child process - This allows a child process in a container to gain more privilege than its parent process.

X. Inactive read-only for root filesystem - This allows the mounting of the container's root file system to be writable with `readOnlyRootFileSystem: False`.

XI. Inactive `runAsNonRoot` - This allows unauthorized write permissions for the filesystem inside a container of a pod using `runAsNonRoot: false`.

XII. Inactive `runAsUser` - This allows a container to run as a root user inside a pod using `runAsUser: false`.

XIII. Missing admission controller - The configuration parameter of not using an admission controller with `AdmissionConfiguration`. Missing admission controller allows a request to bypass necessary security compliance for pod creation.

XIV. Missing network policy - The configuration parameter of not using the `NetworkPolicy` object to specify network policies for pods. Without network policy traffic flows in an unrestricted manner between pods.

XV. Missing resource limit - This allows a container within a pod to run without CPU and memory limit, in turn consuming all available resources. `limit` and `resource` are used to specify resource limits for containers.

XVI. Missing SSL/TLS for HTTP - The configuration parameter of using HTTP without SSL or TLS support. This configuration allows the transmission of HTTP-based pod traffic between pods inside the Kubernetes cluster without SSL/TLS encryption.

XVII. Privileged default `serviceAccount` - This allows a privileged role to be used by a default ServiceAccount.

XVIII. Privileged role - This allows excessive permission to a role. A role is a code construct that allows permissions for a particular namespace [19]. Using the `verbs: ["*"]` configuration for role-related rules, a role becomes privileged. Verbs are used to specify all possible permissions for a pod.

XIX. Privileged `RoleBinding` - This allows a `RoleBinding` object for a Kubernetes-based cluster to list, create, modify, and delete any resources in the entire cluster in an unauthorized fashion. `RoleBinding` is a Kubernetes object that grants the permissions defined in a role to a user or set of users. The `cluster-admin` configuration used in the context of `roleRef` allows a `RoleBinding` object to become privileged.

XX. Privileged `serviceAccount` - This creates a privileged `ServiceAccount`. A service account is a non-human account that provides an identity for processes that run in a pod. In Figure 4, 'sample-sa' is a privileged `ServiceAccount` as it uses a `Role` and `RoleBinding` with a default namespace.

TABLE I: Mapping of Configuration Parameters to Pod-related Attacks

Attack Name	Config. Params.	Count
Access cluster secrets	escalated child process, auto mounted token, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPID`, missing admission controller, privileged `securityContext`, privileged role, privileged service account, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, default `serviceAccount`	14
Dashboard maneuver	escalated child process, auto mounted token, capability abuse, privileged role binding, active `hostIPC`, active `hostNetwork`, active `hostPID`, missing admission controller, privileged `securityContext`, privileged default `serviceAccount`, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, default `serviceAccount`	14
Database tampering	escalated child process, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPath`, active `hostPID`, auto mounted token, default namespace, missing admission controller, missing network policy, privileged `securityContext`, privileged service account, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`	15
DOS	escalated child process, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPath`, active `hostPID`, auto mounted token, missing admission controller, missing resource limit, privileged `securityContext`, inactive `runAsNonRoot`, inactive `runAsUser`, inactive read-only for root filesystem	13
Etcd takeover	escalated child process, auto mounted token, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPID`, missing admission controller, privileged `securityContext`, privileged service account, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, default `serviceAccount`	13
Pod disruption	escalated child process, auto mounted token, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPath`, active `hostPID`, missing admission controller, privileged `securityContext`, privileged role, privileged service account, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, default `serviceAccount`	15

TABLE II: Security Attacks Related to Kubernetes Pods and Their Implications

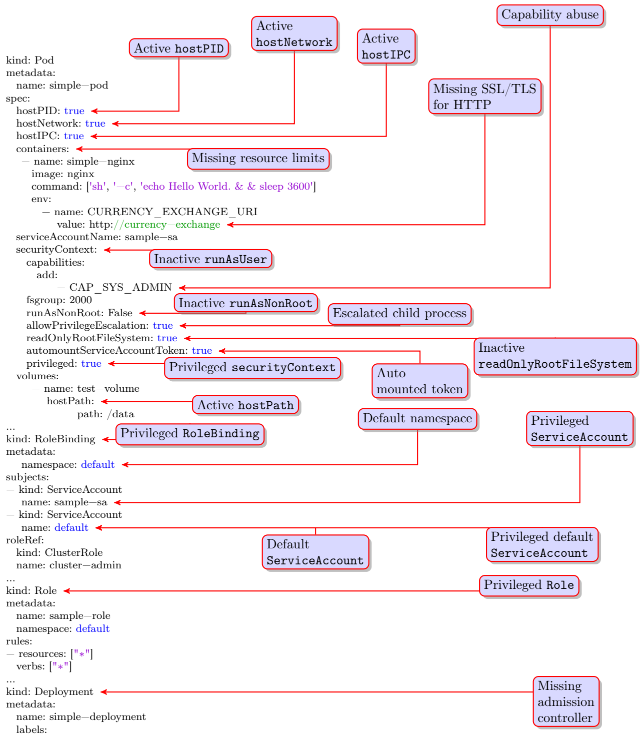
Attack Name	Implication	Objective	Validated
Access cluster secrets	Gain unauthorized access to secrets used to manage the Kubernetes-based cluster	Confidentiality	Yes
Dashboard maneuver	Gain unauthorized access to dashboard and control resources similar to the Tesla attack [11]	Integrity	Yes
Database tampering	Delete and/or modify any cluster-related information stored in the Redis database	Integrity	Yes
DOS	Denial of service to disrupt availability	Availability	Yes
Etcd takeover	Gain unauthorized access to a key-value storage system used as backing store for all cluster data	Confidentiality	Yes
Pod disruption	Disrupt other pods with the help of cryptominer	Availability	Yes

XXI. Privileged `securityContext` - This allows a privileged `securityContext` by disabling all security features of the pod or container.

IV. Empirical Study

In this section, we answer the following research questions:

***RQ2*: How frequently do configuration parameters that facilitate security attacks for Kubernetes pods appear in manifests?**



```
kind: Pod
metadata:
  name: simple−pod
spec:
  hostPID: true
  hostNetwork: true
  hostIPC: true
  containers:
   − name: simple−nginx
     image: nginx
     command: ['sh', '−c', 'echo Hello World. & & sleep 3600']
     env:
       − name: CURRENCY_EXCHANGE_URI
         value: http://currency−exchange
  serviceAccountName: sample−sa
  securityContext:
    capabilities:
      add:
          − CAP_SYS_ADMIN
    fsgroup: 2000
    runAsNonRoot: False
    allowPrivilegeEscalation: true
    readOnlyRootFileSystem: true
    automountServiceAccountToken: true
    privileged: true
  volumes:
     − name: test−volume
       hostPath:
             path: /data
...
kind: RoleBinding
metadata:
   namespace: default
subjects:
− kind: ServiceAccount
   name: sample−sa
− kind: ServiceAccount
   name: default
roleRef:
  kind: ClusterRole
  name: cluster−admin
...
kind: Role
metadata:
  name: sample−role
  namespace: default
rules:
− resources: ["∗"]
  verbs: ["∗"]
...
kind: Deployment
metadata:
  name: simple−deployment
  labels:
```

Fig. 4: Examples of configuration parameters that facilitate security attacks.

***RQ3*: What states in the pod lifecycle map with security attacks for Kubernetes pods?**

A. Methodology for Our Empirical Study

Our methodology is described as follows.

1) RQ2: Frequency of Configuration Parameters: We use the following steps:

TABLE III: Dataset Summary

Attribute	GitHub	GitLab
Repositories	71	21
Contributors	1,187	977
Commits	37,184	15,870
Kubernetes Manifests	1,590	449
Manifest Size (LOC)	148,558	51,512

Dataset Collection: We use OSS repositories that contain Kubernetes manifests and are available on two social coding websites namely, GitHub and GitLab. We apply the following filtering criteria for both data sources: (i) at least 10% of the files in the repository must be Kubernetes manifests; (ii) the repository must be available for download; (iii) the repository is not a clone to avoid duplicates; (iv) the repository must have ≥ 2 commits per month. Munaiah et al. [24] previously used the threshold of ≥ 2 commits per month to determine which repositories have enough software development activity; (v) the repository has ≥ 5 contributors; (vi) the repository is not used for a 'toy' project. We consider a project as 'toy' project if description and content of the README file for each projects indicates that the project is used to demonstrate examples, conduct course work, and used as book chapters. Attributes of our datasets is available in Table III.

SLI-KUBE Extension: We apply and extend SLI-KUBE [28], a state-of-the-art static analysis tool to identify configuration parameters in Kubernetes manifests. Source code of SLI-KUBE is open source and extensible, which allow us to add rules for the 13 configuration parameters that are not covered by SLI-KUBE. For determining the accuracy of the tool, we use a sample with 95% confidence interval, 5% margin of error, and 50% population proportion from the set of 2,039 manifests. We end up with a sample of 324 manifests for which we observe a precision and recall of 0.9 for 6 categories. Obtained precision and recall provides us the confidence that the extended version of SLI-KUBE may generate false positives but not miss any of the configuration parameters.

Analysis: We answer RQ2 by using two metrics. Both of these metrics are related to quantifying how many of the configuration parameters that are needed to conduct a security attack, reside in a single manifest or in all manifests within a repository. The two metrics are: (i) manifest-based attack coverage (MAC) that measures the proportion of configuration parameters needed to execute attack a that reside in a manifest; and (ii) repository-based attack coverage (RAC) that measures the proportion of configuration parameters needed to execute attack a that reside in manifests within the repository. We use Equations 1 and 2 respectively, to calculate MAC and RAC. Let us consider two manifests $m1$ and $m2$ to reside in repository r. For attack a, if three configuration parameters $c1$, $c2$, and $c3$ are required, and $m1$ and $m2$ respectively includes $c1$ and $c2$, then the manifest coverage for $m1$ and $m2$ will be 33.3%, whereas the repository coverage will be 66.6%.

$$\text{Manifest Coverage } [a](\%) = \frac{\text{\# of config. params used for attack } a \text{ present in the manifest}}{\text{total \# of configurations needed for attack } a} * 100\% \quad (1)$$

$$\text{Repository Coverage } [a](\%) = \frac{\text{\# of config. params used for attack } a \text{ present in manifests of the repository}}{\text{total \# of config. params needed for attack } a} *100\% \quad (2)$$

2) RQ3: Pod States that Map with Attacks: The focus of RQ3 is to identify a mapping between pod states and identified attacks. As states are integral to the pod lifecycle, by deriving a mapping between states and attacks we can generate further insights. These insights can be helpful for (i) researchers to gain an understanding of how states are related with attacks, and (ii) toolsmiths on how to detect and mitigate configuration parameters that facilitate attack. We answer RQ3 by using the mapping of configuration parameters and attacks from Table I. For each configuration parameter, we identify the corresponding pod state. We determine a mapping to exist between a state s and an attack a if one or multiple configuration parameters that are used in an attack a, belongs to state s.

B. Results

We provide the answers to RQ2 and RQ3 in this section.

1) Answer to RQ2: In this section, we answer "**RQ2: How frequently do configuration parameters for Kubernetes pods appear in manifests?**" by reporting the manifest-based attack coverage (MAC) and repository-based attack coverage (RAC) metrics. The distribution of MAC and RAC values are presented in Table V. The 'Combined' row presents the distribution of MAC and RAC values considering all 6 attacks. Based on MAC values, on average, a manifest includes 13.7 and 18.1 of the configuration parameters needed to conduct an attack respectively, for the GitHub and GitLab dataset. Based on RAC values, on average, a repository includes 18.3 and 22.3 of the configuration parameters needed to conduct an attack respectively, for the GitHub and GitLab dataset. Considering both datasets, on average a single manifest and a single repository respectively, includes 15.9% and 20.3% of the configuration parameters needed to conduct an attack. For the GitHub dataset, the maximum MAC and RAC values are observed for database tampering, where a manifest includes 58.8% of the required configuration parameters. In the case of GitLab dataset, maximum MAC and RAC values are observed for four attacks: access cluster secrets, dashboard maneuver, etcd takeover, and pod disruption.

The individual frequency of configuration parameters is available in Table IV. We observe missing network policy is the most frequently occurring category.

2) Answer to RQ3: In this section, we answer "**RQ3: What states in the pod lifecycle map with security attacks for Kubernetes pods?**". We identify the following states that relate with security attacks. A mapping between identified pod

TABLE IV: Frequency of Configuration Parameters

Configuration	GitHub	GitLab
Active `hostIPC`	1	0
Active `hostNetwork`	11	3
Active `hostPath`	86	12
Active `hostPID`	5	0
Auto mounted token	881	181
Capability abuse	0	20
Default namespace	95	2
Default `serviceAccount`	822	152
Escalated child process	3	0
Inactive read-only for root filesystem	653	102
Inactive `runAsNonRoot`	586	93
Inactive `runAsUser`	582	93
Missing admission controller	1,509	480
Missing SSL/TLS for HTTP	395	217
Missing network policy	1,571	481
Missing resource limit	70	10
Privileged default `serviceAccount`	0	1
Privileged role	49	40
Privileged role binding	11	2
Privileged service account	75	66
Privileged `securityContext`	1	9
Total	7,406	1,964

states and 6 attacks is available in Table VI. Table VI is sorted alphabetically based on 'State Name'.

1. **Desired pod state stored in etcd:** The state where the Kubernetes API stores the pod object specification in etcd.

2. **Image provided to container runtime interface:** The state where the container image is provided to the container runtime interface.

3. **Image pull from registry:** The state where the kubelet agent in the worker node pulls image from the container image registry.

4. **Kubelet receives pod specification:** The state where the kubelet agent in the worker node receives the pod specification from the Kubernetes API server.

5. **Pod creation request initiated:** The state where the Kubernetes API server creates a pod object.

6. **Pod evicted:** The state where a pod evicts other pods due to unnecessary resource consumption.

7. **Pod starting:** The state where the pod has its own storage, and at least one container running inside itself.

8. **Pod running:** The state where the containers in a pod is running.

9. **Remote service connected:** The state where the pod has exposed shell to an unauthorized remote machine.

10. **Request authenticated:** The state where the Kubernetes API server authenticates pod creation request.

11. **Request authorized:** The state where the Kubernetes API server authorizes a pod creation request.

12. **Volume mounted:** The state where the pod mounts volume for the container.

V. Discussion

We discuss the implications and threats to validity of our paper as follows:

A. Implications

Implications for Enhancing Security Static Analysis Tools: In a recent empirical study [28], researchers used their subjective judgement to derive 11 configuration parameters with security implications. Their [28] conclusion was that mitigation of each of the configuration parameters is important because they can individually cause attacks. While we acknowledge the importance of their empirical study, our analysis provides a different view. On the contrary to their study [28] we find that a single configuration parameter cannot be used to conduct security attacks. Instead, a combination of configuration parameters—at the very least a combination of 13—is needed to conduct a security attack. This finding has implications for toolsmiths who are involved in developing security static analysis tools, such as SLI-KUBE, Trivy [2], and KubeLinter [17]. These tools should be enhanced by incorporating attack-related information by highlighting the fact that a single configuration parameter does not cause an attack. For example, currently both SLI-KUBE and KubeLinter report security misconfigurations without providing the full context that the detected configuration parameter is consequential only when used with other configuration parameters. Without this context, a practitioner may deem the detected misconfiguration useless and not take any action to fix. Reporting of irrelevant static analysis alerts can leave security weaknesses unmitigated [15], [27], which in turn can hinder the secure development of software artifacts. Hence, incorporation of necessary attack-related information presented in Section III-B could aid in secure development of Kubernetes manifests.

Implications for Enhancing Kubelet: Our findings can be helpful for enhancing Kubelet, an automated agent within Kubernetes that executes pod requirements and manages pod-related resources [19]. From Table VI, we observe nine states to map to all attacks. Amongst these states, 'Kubelet receives pod specification' is the state when 'kubelet' receives the pod specification from the Kubernetes API server. Prior to transitioning to other pod-related states, kubelet can be enhanced to identify if the pod specification includes identified configuration parameters. If yes, then kubelet can stop executing the next steps, and report back to the end-user that there is a configuration parameter present in the manifest that needs to be mitigated.

Implications for Researchers: Our empirical study lays the groundwork for conducting future research in the following directions: (i) derivation and application of FSM-based approaches to investigate reliability concerns for Kubernetes along with security attacks; and (ii) replication of our FSM-based approach for other Kubernetes entities, such as operators, control planes, and network planes.

TABLE V: Answer to RQ2: Manifest-based Attack Coverage (MAC) and Repository-based Attack Coverage (RAC)

Attack	**MAC**		**RAC**	
	GitHub (Min, Median, Avg., Max)	**GitLab** (Min, Median, Avg., Max)	**GitHub** (Min, Median, Avg., Max)	**GitLab** (Min, Median, Avg., Max)
Access cluster secrets	(0.0, 16.6, 17.4, 55.5)	(0.0, 5.5, 12.9, 55.5)	(0.0, 16.6, 17.7, 38.9)	(0.0, 16.6, 21.5, 55.5)
Dashboard maneuver	(0.0, 16.6, 17.4, 55.5)	(0.0, 5.5, 12.9, 55.5)	(0.0, 16.6, 17.7, 38.9)	(0.0, 16.6, 21.5, 55.5)
Database tampering	(0.0, 17.6, 21.6, 58.8)	(0.0, 17.6, 17.6, 47.0)	(0.0, 17.6, 21.9, 41.1)	(0.0, 23.5, 25.3, 47.0)
Denial of service	(0.0, 13.3, 17.5, 53.3)	(0.0, 6.6, 13.2, 46.6)	(0.0, 13.3, 17.5, 33.3)	(0.0, 20.0, 22.5, 46.6)
Etcd takeover	(0.0, 16.6, 17.4, 55.5)	(0.0, 5.5, 12.9, 55.5)	(0.0, 16.6, 17.7, 38.9)	(0.0, 16.6, 21.5, 55.5)
Pod disruption	(0.0, 16.6, 17.4, 55.5)	(0.0, 5.5, 12.9, 55.5)	(0.0, 16.6, 17.7, 38.9)	(0.0, 16.6, 21.5, 55.5)
Combined	(0.0, 16.6, 18.1, 58.8)	(0.0, 5.5, 13.7, 55.5)	(0.0, 16.6, 18.3, 41.1)	(0.0, 16.6, 22.3, 55.5)

TABLE VI: Mapping between Pod State and Attacks

State Name	**Attack**
Desired pod state stored in etcd	All
Image provided to container runtime interface	All
Image pull from registry	All
Kubelet receives pod specification	All
Pod creation request initiated	All
Pod evicted	DOS
Pod running	Dashboard maneuver, Pod disruption
Pod starting	All
Remote service connected	Access cluster secrets, Database tampering, Etcd takeover
Request authenticated	All
Request authorized	All
Volume mounted	All

B. Threats to Validity

The limitations of our paper are:

Conclusion Validity: Our constructed FSM is susceptible to generating counterexamples, which include configuration parameters that are false positives. We mitigate this limitation by determining which combinations of configuration parameters actually lead to security attacks. Also, our derived set of 9 requirements used for RQ1 may not be comprehensive.

Construct Validity: Our constructed FSM applies for a generic pod, and not for specific workloads, such as replica sets. Also, we have used FSM to represent the lifecycle of pods. This could be limited as our construction of FSM is dependent on adequate representation of states that belong to each of the phases in the pod lifecycle. We mitigate this limitation by following the Kubernetes documentation [19] that discusses the states of the pod lifecycle.

External Validity: Our results are limited to a set of OSS repositories that we curated. Therefore, our findings may also not generalize to proprietary datasets and other OSS datasets not used in the paper. Our assumption of having access to a valid a kubeconfig file and the 'kubectl' utility may not be applicable for real-world Kubernetes-based deployments.

VI. Related Work

Our paper is related to prior research in the area of Kubernetes security. Anomaly detection is one topic researchers have focused on. For example, Hariri et al. [14] proposed a tool for detecting anomalies for Kubernetes-based astronomy data analysis. Tien et al. [35] implemented a tool for anomaly detection in the Kubernetes cluster, using neural network approaches. Cao et al. [8] proposed a state-based model that detects attacks in Kubernetes cluster. Other perspectives on Kubernetes security, such as proposal of an automated threat mitigation framework [1], zero trust architecture [34], and serverless architectures [4] have also been investigated. Security-focused empirical studies have also garnered interest. Researchers have quantified vulnerability-related commits for Kubernetes manifests [6], and derived best practices [33]. Blaise et al. [5] evaluated security model of a Kubernetes package manager by identifying the riskiest attack paths. Rahman et al. [28]'s recent paper is the closest in spirit to our work. They [28] applied qualitative analysis on Kubernetes manifests to derive 11 misconfigurations. Unlike their paper [28], we have: (i) identified combinations of configuration parameters that cause security attacks by accounting for the pod lifecycle; and (ii) provided a mapping between attacks and states. Furthermore, we identify 13 configuration parameters not reported in their paper [28]: active `hostPath`, auto mounted token, default namespace, default `serviceAccount`, inactive read-only root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, missing admission controller, missing network policy, privileged default `serviceAccount`, privileged role, privileged role binding, and privileged `serviceAccount`.

VII. Conclusion

While pods in Kubernetes play a pivotal role to rapidly deploy software applications, they can be susceptible to security attacks. In our empirical study, we have used a FSM-based approach to determine a set of configuration parameters that facilitate security attacks for pods. We identify 6 attacks unique to Kubernetes that can be facilitated using combinations of 21 configuration parameters. Based on our findings, we recommend (i) toolsmiths to enhance existing security static analysis tools to incorporate attack information, and (ii) researchers to assess the security of non-pod Kubernetes entities, such as operators.

References

[1] 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: STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*. Springer, 2018, pp. 573–588.

[2] Aqua Trivy, "Trivy - The all-in-one open source security scanner," https://trivy.dev/, 2024, [Online; accessed 21-January-2024].

[3] A. Authors, "Verifiability package for paper," https://figshare.com/s/cecfd62d8a2ae23e4c80, 2024, [Online; accessed 20-March-2024].

[4] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the serverless architecture for securing linux containers," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 401–404.

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

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

[7] Canonical, "Kubernetes and cloud native operations report 2021," 2021. [Online]. Available: https://juju.is/cloud-native-kubernetes-usage-report-2021

[8] C. Cao, A. Blaise, S. Verwer, and F. Rebecchi, "Learning state machines to monitor and detect anomalies on a kubernetes cluster," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–9.

[9] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 334–342.

[10] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem *et al.*, *Handbook of model checking*. Springer, 2018, vol. 10.

[11] Tesla cloud resources are hacked to run cryptocurrency-mining malware, February 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/

[12] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[13] Google, "microservice-demo," https://github.com/GoogleCloudPlatform/microservices-demo, 2023, [Online; accessed 10-Apr-2023].

[14] S. Hariri and M. C. Kind, "Batch and online anomaly detection for scientific applications in a kubernetes environment," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, ser. ScienceCloud'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3217880.3217883

[15] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 161–170.

[16] https://www.armosec.io, "Definitive Guide to Kubernetes Admission Controller," https://www.armosec.io/blog/kubernetes-admission-controller/, 2023, [Online; accessed 28-April-2023].

[17] KubeLinter, "Introduction - KubeLinter," https://docs.kubelinter.io/, 2024, [Online; accessed 19-January-2024].

[18] Kubernetes User Case Studies, May 2020. [Online]. Available: https://kubernetes.io/case-studies/

[19] Kubernetes, "Production-grade container orchestration," 2021. [Online]. Available: https://kubernetes.io/

[20] ——, "Case Study: Capital One," https://kubernetes.io/case-studies/capital-one/, 2023, [Online; accessed 10-June-2023].

[21] ——, "Case Study: OpenAI," https://kubernetes.io/case-studies/openai/, 2023, [Online; accessed 11-June-2023].

[22] Michelle Yakura, "5 challenges that can put your Kubernetes deployments at risk," https://www.mirantis.com/blog/5-challenges-that-can-put-your-kubernetes-deployments-at-risk/, 2023, [Online; accessed 21-Feb-2024].

[23] S. Miles, *Kubernetes: A Step-By-Step Guide For Beginners To Build, Manage, Develop, and Intelligently Deploy Applications By Using Kubernetes (2020 Edition)*. Independently Published, 2020. [Online]. Available: https://books.google.com/books?id=M4VmzQEACAAJ

[24] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, pp. 1–35, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9512-6

[25] Nmap.org, "netcat tool," https://nmap.org/ncat/, 2023, [Online; accessed 28-April-2023].

[26] NSA, "Kubernetes Hardening Guidance," https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/1/CTR_KUBERNETESHARDENINGGUIDANCE.PDF, 2021, [Online; accessed 10-Jan-2022].

[27] A. Rahman and C. Parnin, "Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management," *IEEE Transactions on Software Engineering*, vol. 49, no. 06, pp. 3536–3553, June 2023.

[28] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3579639

[29] redis.io, "Redis Security," https://redis.io/docs/management/security/, 2023, [Online; accessed 28-April-2023].

[30] M. N. Research, "Case Study: OpenAI," https://www.globenewswire.com/news-release/2023/03/06/2621358/0/en/Latest-Global-Kubernetes-Market-Size-Share-Worth-USD-7-8-Billion-by-2030-at-an-23-40-CAGR-Markets-N-Research-Share-Trends-Cap-Adoption-Forecast-Segmentation-Growth-Value.html, 2023, [Online; accessed 12-June-2023].

[31] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[32] Seth Art, Principal Security Consultant, "Bad Pods: Kubernetes Pod Privilege Escalation," https://bishopfox.com/blog/kubernetes-pod-privilege-escalation, 2023, [Online; accessed 28-April-2023].

[33] M. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *2020 IEEE Secure Development (SecDev)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2020, pp. 58–64.

[34] N. Surantha and F. Ivan, "Secure kubernetes networking design based on zero trust model: A case study of financial service enterprise in indonesia," in *Innovative Mobile and Internet Services in Ubiquitous Computing: Proceedings of the 13th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2019)*. Springer, 2020, pp. 348–361.

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

[36] tutorialspoint.com, "Redis - Security," https://www.tutorialspoint.com/redis/redis_security.htm, 2023, [Online; accessed 28-April-2023].

[37] A. Zimba, Z. Wang, M. Mulenga, and N. H. Odongo, "Crypto mining attacks in information systems: An emerging threat to cyber security," *Journal of Computer Information Systems*, 2018.