Mitigation of Security Misconfigurations in Kubernetes-based Container Orchestration: A Techno-Educational Approach

by

Md Shazibul Islam Shamim

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 3, 2024

Keywords: security misconfiguration, security practices, kubernetes, model checking, cybersecurity education, authentic learning

Approved by

Dr. Akond Rahman, Assistant Professor of Computer Science and Software Engineering
Dr. Drew Springall, Assistant Professor of Computer Science and Software Engineering
Dr. Jakita O. Thomas, Philpott-WestPoint Stevens Associate Professor of Computer Science and Software Engineering
Dr. Samuel Mulder, Associate Research Professor of Computer Science and Software Engineering
Dr. Mehdi Sadi, Assistant Professor of Electrical and Computer Engineering

Abstract

Kubernetes has emerged as the preferred tool for implementing automated container orchestration, offering significant advantages for IT organizations. However, the presence of security misconfigurations can render Kubernetes-based software deployments vulnerable to security attacks. **The goal of this doctoral dissertation is to help practitioners secure their Kubernetes-based container-orchestration process by adopting a techno-educational approach.** This PhD dissertation advances the science of Kubernetes misconfigurations by conducting three empirical studies. *First*, in order to assist practitioners in enhancing the security of their Kubernetes clusters, a qualitative analysis is conducted on 104 Internet artifacts, including blog posts, resulting in the identification of 11 Kubernetes security best practices. *Second*, to help practitioners secure the container orchestration with Kubernetes, we conduct a systematic investigation of configuration parameters that can aid practitioners in identifying configuration parameters that need to be avoided in order to secure a Kubernetes-based deployment infrastructure. Our approach is informed by gaining an understanding of the states associated with the pod lifecycle. Using our approach, we identify 6 attacks unique to Kubernetes that can be facilitated using combinations of 21 configuration parameters. *Finally*, we adopt authentic learning-based exercise to provide students with practical, hands-on experiences in addressing real-world challenges in Kubernetes security. We deploy our authentic learning-based exercise in 4 semesters among 246 students. Furthermore, we observe that 90.6% and 93.3% students report that they learned about Kubernetes security misconfigurations and the automated configuration management tools, respectively. Furthermore, students report that the instructor's academic, industry, and research backgrounds are useful for authentic learning exercises.

Acknowledgments

First and foremost, I express my deepest gratitude to Almighty Allah for His blessings throughout my PhD journey. I extend my heartfelt thanks to my parents, especially my mother, Mst Selina Akhter Jahan, whose unwavering support has been the cornerstone of my academic career. I am also profoundly grateful to my father, Md Shafiqul Islam, for his steadfast support and faith in me. I am thankful to my brother, Sakibul Islam Shimul, and my cousin, Saimum Rahman, Firoz Hasan, Shariful Alam for their support during the challenging pandemic era, when I had just started my PhD. I thank my wife, Marufa Islam, for her constant support and inspiration during the challenging phase of my PhD research. Without the love and sacrifices of my family, this achievement would not have been possible. I am also grateful to all my extended family members for their inspiration, prayers, and support.

I would like to thank my PhD supervisor, Dr. Akond Rahman, for his invaluable guidance, continuous support, and patience throughout my PhD journey. Special thanks to my dissertation committee members, Dr. Drew Springall, Dr. Jakita Owenby Thomas, and Dr. Samuel Mulder, for their thorough review of my PhD proposal documents and dissertation, and for their valuable suggestions that significantly enhanced the quality of my work. My sincere thanks also go to Dr. Mehdi Sadi for serving as the University reader for my PhD dissertation and providing invaluable feedback.

I would like to thank my friends from Bangladesh University of Engineering Technology, Tennessee Tech University, and Auburn University. I am also grateful to my colleagues at iPay Systems Limited. My gratitude extends to Dr. Abul Kashem Mia, Dr. A.B.M Alim Al

Islam, Abdus Salam Azad, Dr. Abu Wasif, and Mohsin Khan for recommending me to apply for a PhD program. I am thankful to Dr. B K Bose and Dr. Kanta Roy for inspiring me to pursue this academic path. Special thanks to Arafat Mahmood, Chowdhury Md Rakin Haider, Md Arifuzzaman, Akhter Al-Amin, Saadbin Khan, Muhammad Ahad Ul Alam, Raisul Islam Zaeem, Mohammad Salman Yasin, Arifur Reza, Ishriak Ahmed, Md Touhiduzzaman, Mostafiz Rahman, Rafi Kamal, Sheikh Shakib Ahmed, Abdullah Al Fahim, Nafisa Anzum, Tanzeer Hossain, Saiful Islam, Dipayan Banik, Ahsan Ayub, Bulbul Sharif, Golam Maula Mehedi Hasan, Nishan Biswas, Sk. Yasir Arafat, Nafiul Huda, Al Artat Bin Ali, Muntasir Maruf, Minarul Islam, Monir Hossain, Raisul Arefin, Shafiqul Islam, Mozahidul Islam, Muhammad Nafisur Rahman, Sk. Alimuzzaman, Ishita Islam, Fahmida Shabnam, Rakibul Hasan Reyad and Monzurul Quader for all their support and inspiration, from applying to the PhD program to defending my dissertation.

I would like to thank my colleagues from the PASER group at Auburn University for their constructive feedback and support. Special thanks to Farzana Ahamed Bhuiyan, Justin Murphy, Raunak Shakya, Yue Zhang, and Pemsith Mendis, my former colleagues from the PASER group, for their valuable feedback on my research. I am grateful to my internship supervisor, Dr. Seratun Jannat, at GEODIS, and to Hanyang Hu, under whose supervision my research and technical skills expanded significantly. My gratitude also extends to my graduate writing partner, Sidharth Suresh Gautam, for his valuable suggestions and feedback.

During my PhD tenure, I have authored or co-authored six publications with eight different authors. I am grateful to all my co-authors for their contributions, which enriched my research. I also express my gratitude to Dr. Xiao Qin and Dr. Hari Narayanan, the chair of the Computer Science and Software Engineering department at Auburn University. I would like to thank Dr. Sheikh Ghafoor, Dr. William Eberle, Dr. Manak Gupta, Dr.

Muhammad Ismail, and Dr. Doug Talbert for their support and feedback, and Dr. Jerry Gannod, the chair of Computer Science at Tennessee Tech University, for funding my PhD research. I am also thankful to the National Science Foundation (NSF) for their financial support of my PhD research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Container technologies, such as Docker and LXC are gaining popularity amongst information technology (IT) organizations for deploying software applications. For example, PayPal uses 200,000 containers to manage 700 software applications [74]. For managing these containers at scale, practitioners often use automated container orchestration, i.e, the practice of pragmatically managing the life-cycle of containers with tools, such as Kubernetes [73].

Since its inception in 2014, Kubernetes has established itself as the *de-facto* tool for automated container orchestration [93, 13]. According to Stackrox survey [104], 91% of the surveyed 500 practitioners use Kubernetes for container orchestration. As of Sep 2020, Kubernetes has a market share of 77% amongst all container orchestration tools [106]. Organizations, such as Adidas, Twitter, IBM, U.S. Department of Defense (DOD), and Spotify are currently using Kubernetes for automated container orchestration. Use of Kubernetes has resulted in benefits, e.g., using Kubernetes the U.S. DoD decreased their release time from 3∼8 months to 1 week [18]. In the case of Adidas, the load time for their e-commerce website was reduced by half, and the release frequency increased from once every 4∼6 weeks to 3∼4 times a day [60].

Kubernetes-based container orchestration, similar to every other configurable software, is susceptible to security misconfigurations. However, due to the pervasive nature of Kubernetes-based container orchestration, such misconfigurations can have severe security implications. According to the 2021 'State of Kubernetes Security Report', 94% of 500 practitioners experienced at least one Kubernetes-related security incident, majority of which can be attributed

to security misconfigurations [93]. The survey also states Kubernetes-related misconfigurations to "*pose the greatest security concern*" for Kubernetes-based container orchestration [93]. Anecdotal evidence attests to such perceptions: for example, a Kubernetes-related security misconfiguration resulted in a data breach that affected 106 million users of Capital One, a U.S.-based credit card company [54, 107]. We also observe cryptomining attack in electric car manufacturer company Tesla's Amazon Web Services (AWS) resources due to Kubernetes security misconfiguration [22]. In case of security breach, an organization often face project delays or disruptions that causes revenue or customer loss, financial loss such as fines as well as legal actions or lawsuits. [95]. The end user of the service of the organization also gets affected in the security breach which includes private data leak, data loss and service disruption [95].

Practitioners often report what security practices they use in Internet artifacts [32, 34] rather than in academic forums such as conferences. One strategy to address this problem is to systematize available knowledge regarding Kubernetes security practices that could support practitioners to secure their Kubernetes environment. Systematization of knowledge can be conducted by analyzing Internet artifacts, such as blog posts and video presentations. Such systematization of knowledge can be beneficial for practitioners to understand what practices need to follow to secure Kubernetes components and use the derived list of practices as a benchmark to compare their existing state of security practices. To systematically synthesize practitioner-reported security best practices for Kubernetes, we answer the following research question in Section 3:

- **RQ 3.1** What Kubernetes security practices are reported by practitioners?

Additionally, we observe anecdotal evidence in open-source software (OSS) repositories that provide clues on what security misconfigurations can occur for Kubernetes. In the case of Kubernetes, a pod is considered the most fundamental unit for performing container orchestration. 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. In Figure 1.1 we present a code snippet related to Kubernetes manifests, and mined from OSS repositories [24, 105] that has a security misconfiguration `allowPrivilegeEscalation:True`. When a practitioner configures a pod using this Kubernetes manifest, the `allowPrivilegeEscalation:True` misconfiguration allows a child process of a container to gain more privileges than its parent process. As a result, any malicious user can leverage this misconfiguration to gain unauthorized access to the underlying host machine [71].

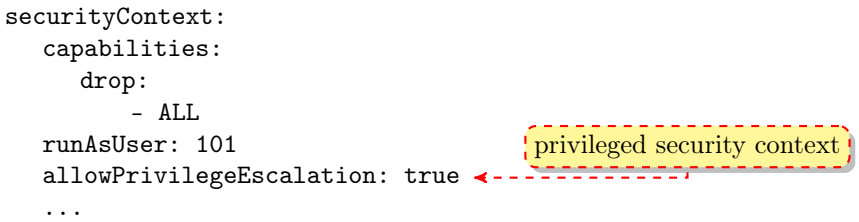

Figure 1.1: Anecdotal evidence of security misconfigurations in Kubernetes manifests that shows an example of a security misconfiguration related to privilege escalation in a Kubernetes manifest

Existing security analysis tools for Kubernetes [103], [58], [3], [23], [14] scan Kubernetes manifests, repositories and report one misconfiguration at a time and they do not provide any additional context how a misconfiguration can be leveraged by a malicious user with the combination of other security misconfiguration to conduct a security attack. As a result, the practitioners can not identify the pod-related configuration parameters that can facilitate a security attack. In order to secure the container orchestration process with Kubernetes, practitioners must identify pod-related configuration parameters that can facilitate security attacks. However, 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.

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 require- ments unique to pods need to be identified.

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.

To mitigate the above mentioned challenges and perform systematic investigation to determine which configuration parameters facilitate security attacks for Kubernetes pods we answer the following research questions in Section 5:

- **RQ 5.1** What configuration parameters facilitate security attacks for Kubernetes pods?
- **RQ 5.2** How frequently do identified configuration parameters appear in Kubernetes manifests?
- **RQ 5.3** What states in the pod lifecycle map with security attacks for Kubernetes pods?

As an open source software, Kubernetes codebase is large and complex with minimal documentation [108] and as of April 2024 Kubernetes GitHub repository has 2.2 million lines of code [46]. As a result, it becomes difficult for practitioners to learn and grow Kubernetes-related skills from the official codebase and documentation. A recent survey conducted by Cloud Native suggests that 48% (595) of the survey respondents among 1,240 participants reported "Lack of in-house skills/limited manpower" for running and maintaining their Kubernetes cluster [62]. According to the state of Kubernetes survey, among 247 participants 70% and 67% cited lack of experience and expertise as a top deployment and top management challenges [111] respectively. Moreover, practitioners often lack knowledge needed to mitigate security misconfigurations [71].According to red hat 2024 survey, among 600 practitioners 30% of them reported that they lack internal security talents for their Kubernetes security solutions [95]. Although the internet artifacts, and survey shows the lack of security talent in Kubernetes, academic researchers have not yet proposed any educational approach to train the Kubernetes practitioners. A Kubernetes practitioner is an individual who deploys, manages, and maintains applications using Kubernetes. Kubernetes practitioners also

include individuals such as graduate and undergraduate students who are learning Kubernetes in academic classrooms or as learners in a non-academic environment. Prior research has shown that students can be used as a surrogate measurement for experiments in software engineering [27], [28]. The researchers describe the drawback of using professionals compared to students in software engineering experiment such as compensation for time, low sample size, less commitment issue, lower internal validity for the experiment [28].

To mitigate the challenge of a lack of security experts in Kubernetes, we take an educational approach to train the practitioners in Kubernetes security. We use an authentic learning-based instructional approach as authentic learning exercises have proven effective in enhancing students' understanding of various subjects, such as mobile application security [83] and infrastructure-as-code (IaC) [89]. We answer the following research questions in Section 6:

- **RQ 6.1**: How to design authentic learning-based exercise to help students for secure development of Kubernetes Manifests?
- **RQ 6.2**: How does authentic learning help students to learn about the secure development of Kubernetes Manifests?
- **RQ 6.3**: What instructor-related attributes are useful for students in an authentic learning-based exercise used for Kubernetes security misconfiguration analysis?

This dissertation thesis will impact the state-of-the-art secure development of Kubernetes. We hypothesize that through systematically synthesizing the knowledge related to Kubernetes security best practices study, we can help the practitioners to integrate best security practices. All of the evidence mentioned above emphasizes the need for a security analysis of pod-related configuration parameters that can weaken the security posture of the pod at runtime. Such analysis can help practitioners understand the Kubernetes security attacks due to the pod-related configuration parameters. We leverage our understanding of Kubernetes pod lifecycles to construct finite state machines (FSM). We hypothesize that

FSM will be used to identify pod-related configuration parameters that can be used to conduct security attacks. Moreover, we create authentic learning-based exercises for the next generation of Kubernetes practitioners to learn about Kubernetes security misconfiguration and provide empirical analysis for the effectiveness of the exercise.

The goal of this doctoral dissertation is to help practitioners secure their Kubernetes-based container-orchestration process by adopting a techno-educational approach.

In this dissertation, we make the following contributions:

- A synthesized list of security practices for Kubernetes(Section 3.3);
- A curated dataset with a mapping between Internet artifact and identified security best practices [7];
- A list of 21 configuration parameters combinations of which facilitate 6 security attacks for Kubernetes pods (Section 5.4);
- An empirical validation of the identified attacks from Kubernetes formal verification using pod-related configuration parameters (Section 5.1);
- A mapping between of pod-related configuration parameters and Kubernetes-related security attacks (Section 5.5);
- An empirical analysis of open-source Kubernetes manifests to identify pod-related configuration parameters that can facilitate security attack (Section 5.5.1);
- A mapping between states and pod-related security attacks (Section 5.6);
- An evaluation of students' perception of authentic learning to design a more effective curriculum for students (Section 6.2.1);

- An evaluation of the authentic learning module's effectiveness in teaching students Kubernetes security misconfiguration (Section 6.2.2); and
- An evaluation of the students' perception on the usefulness of instructor-related attributes in an authentic learning-based exercise used for Kubernetes security misconfiguration analysis (Section 6.2.3).

Chapter 2

Background and Related Work

2.1 Background

In this section, we provide relevant background and discuss related academic works. First, we provide a brief background on Kubernetes and its architecture. Then we provide background on Kubernetes manifests, which are files used to define and deploy pods, the fundamental units of applications in Kubernetes. We also provide background on the life cycle of pod in Kubernetes. After that, we provide a brief description on authentic learning, an instructional approach that emphasizes real-world problem-solving activities. We end this section by describing related academic research.

2.1.1 Kubernetes Architecture

Kubernetes is an open-source software for automating management of computerized services such as containers [73]. A Kubernetes installation is colloquially referred to as a Kubernetes cluster [73]. Each Kubernetes cluster contains a set of worker machines defined as nodes. As shown in Figure 2.1, two types of nodes exist for Kubernetes: master nodes and worker nodes.

Each control-plane node includes the following components: 'API server', 'scheduler', 'controller', and 'etcd' [73]. The 'API server' is responsible for orchestrating all the operations within the cluster. Kubernetes serves its functionality through an application program interface from the 'API server'. The 'controller' is a component on the control-plane that watches the state of the cluster through the 'API server' and changes the current state towards the desired state. The 'scheduler' is the component in the control plane responsible for scheduling pods across multiple nodes. The 'etcd' is a key-value based database that

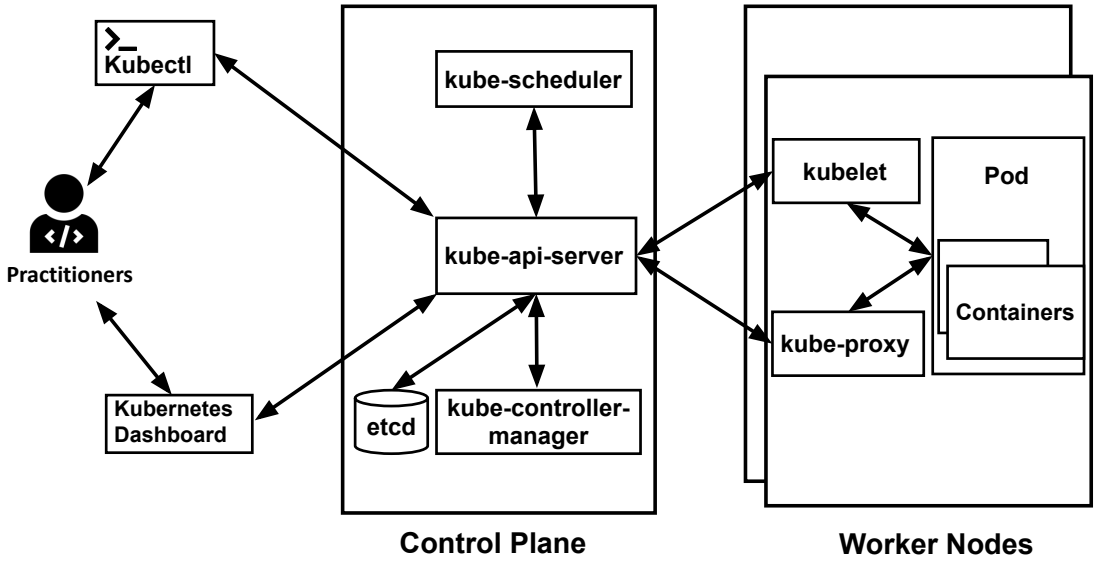

Figure 2.1: A brief overview of Kubernetes. Kubernetes users interact with the installation using the Kubernetes dashboard and 'kubectl'. The purpose of control-plane node is to maintain the desired cluster state and manage worker nodes. Worker nodes are used to run containerized applications inside the pod.

stores all configuration information for the Kubernetes cluster. Users use a command-line tool 'Kubectl' to communicate with the 'API server' in the control-plane node.

The worker nodes host the applications that run on Kubernetes [73]. The following components are included in the worker node: 'kube-proxy', 'kubelet' and 'pod'. 'kube-proxy' maintains the network rules on nodes. 'kubelet' is an agent that ensures containers are running inside a pod. The pod is the smallest Kubernetes entity, which includes at least one active container. A container is a standard software unit that packages the code and associated dependencies to run in any computing environment [73].

2.1.2 Kubernetes Manifest

Kubernetes allows practitioners to create persistent objects using declarative configurations [59]. Kubernetes provides a command line tool called "`kubectl`" that allows the

practitioners to communicate with the Kubernetes cluster to create, update, and delete Kubernetes objects with desired state using object configuration files called Kubernetes manifests [59]. Practitioners write Kubernetes manifests and use the "`kubectl apply`" command in the command line terminal using appropriate privilege to configure objects and update the live configuration of an object [59]. Kubernetes manifests are written as a YAML file that describes the desired state of a Kubernetes object in a Kubernetes cluster [88]. In Listing 1, we provide a sample example of a Kubernetes object `pod` defined by Kubernetes manifest [59].

2.1.3 Pod Life Cycle

In Kubernetes, the pod is the smallest deployable and manageable unit. Each pod goes through certain phases during their life cycle depending on the condition of the containers inside the pod. The 'kube-scheduler' in the control-plane node schedules each pod only once. After scheduling a worker node for the pod, the pod runs in the worker-node until the worker node stops or the pod terminates.

Once an authenticated and authorized user creates a valid pod creation request, the Kubernetes API server accepts the request and stores the information in 'etcd' database in control plane node. After Kubernetes API accepts the pod creation request the pod goes to 'pending' phase. Kube-scheduler assigns the pod to a node and Kubernetes API server stores that information to 'etcd'. Finally, the 'kubelet' agent in the worker node receives the pod specification, pulls the image from the container registry and provides the image to the container runtime to run the container. If container runtime starts at least one container or in the process of starting or restarting then the pod goes to 'running' phase. When the containers inside the pod terminates and at least one container ends with failure such as terminated by system or exited with non-zero status, the pod goes to 'failed' state. The failed container may restart based on restart-policy upon failure if it is created by other workloads such as replica sets. If the containers in a pod ends in success and will not restart then pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Listing 1: An example of Kubernetes manifest for Pod

reaches ‘succeeded‘ state. The figure 2.2 demonstrates the pod life cycle as described in this section.

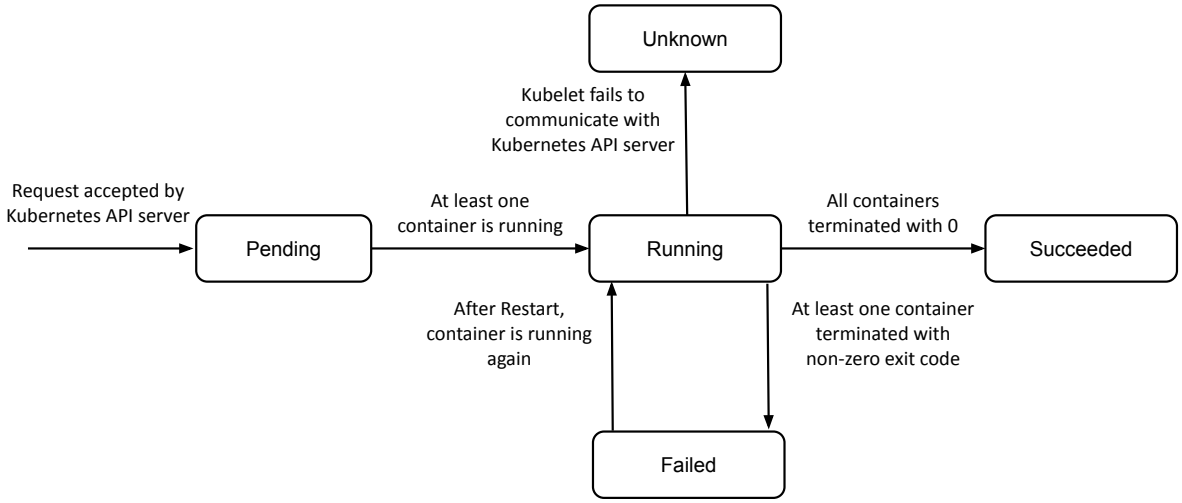

Figure 2.2: A simple graphical demonstration of a pod life cycle.

2.1.4 Background on Authentic Learning

Authentic learning is recognized as an instructional approach that prioritizes the engagement of students in problem-based activities that reflect real-world contexts [68]. Authentic learning is more of philosophy for exercise design rather than learning theory [37]. Herrington et al. [38] describes that Authentic learning comprises 9 design elements: (i) authentic context, (ii) authentic tasks and activities (iii) access to expert performances, (iv) multiple

roles and perspectives, (v) support collaborative construction of knowledge, (vi) reflection (vii) articulation, (viii) coaching and scaffolding and (ix) authentic assessment. When implementing authentic learning based exercise, the exercise follow the the elements for creating authentic learning environment for the students. The exercises also exhibit distinct characteristics that contribute to its effectiveness. These characteristics include [70]: (i) it focuses on hands-on exercises relevant to the real-world problems, (ii) it encourages students to have a diverse set of perspectives for the same exercise, and (iii) it utilizes available resources to solve exercises.

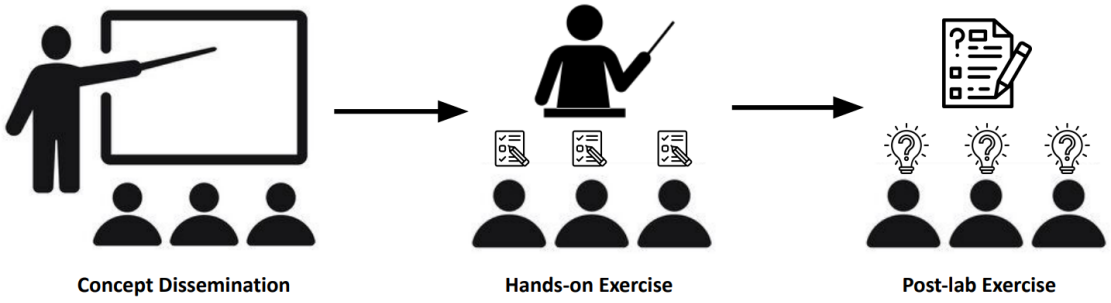

Figure 2.3: The diagram illustrates the three distinct steps of the authentic learning-based exercise, encompassing pre-lab content dissemination, hands-on exercise and active learning, and post-lab exercise with real-world scenarios.

The implementation of an authentic learning-based exercise typically involves three distinct steps. In Figure 2.3, we have demonstrated three steps of authentic learning steps. The inclusion of these three steps in the authentic learning-based exercise ensures a holistic and practical learning experience for students, promoting deeper engagement and mastery of the subject matter. In these three steps core authentic learning elements are incorporated to provide a supportive environment that motivates student to learn in relevant and real-word scenarios [38]. The three steps are as follows:

- Step 1: Pre-Lab Content Dissemination
- Step 2: Hands-On Exercise
- Step 3: Post-Lab Exercise

Step 1: Pre-Lab Content Dissemination: In this step, the instructor introduces the students to the fundamental concepts related to the topic at hand. Through various teaching methods, such as lectures or presentations, the instructor imparts the necessary theoretical knowledge and background information to the students. In this step, the instructor provides authentic context to the student by disseminating the knowledge related to the subject matter [38]. This phase sets the foundation for the subsequent hands-on exercises.

Step 2: Hands-On Exercise: In this step of the authentic learning-based exercise involves providing students with hands-on exercises that are directly relevant to the real-world application of the subject matter. Through active learning strategies, students engage directly with the material and apply their theoretical knowledge in practical scenarios. The instructor guides and supports the students during this hands-on exercise phase, facilitating their learning and understanding of the subject matter through active participation. In this step, the students learn to conduct authentic activities, and get access to expert assessment and coaching support from the instructor to create a supportive learning environment [37]. The students also get the opportunity to get perspective on multiple roles. Following the completion of the hands-on exercise, the authentic learning-based exercise progresses to the post-lab exercise stage.

Step 3: Post-Lab Exercise: In this phase, the instructor presents the students with exercises based on real-world scenarios that reinforce and deepen their understanding of the subject matter. These exercises challenge students to apply their acquired knowledge and skills to solve complex problems or address practical challenges. By working through these real-world scenarios, students develop a more comprehensive understanding of the subject matter and enhance their problem-solving abilities in authentic contexts. In the post lab step, the students get the opportunity to reflect and articulate their learning to perform authentic assessment [37].

Authentic learning and experiential learning are two distinct instructional approaches that are often compared in the context of education. While authentic learning emphasizes

real-world problem-solving activities, experiential learning follows a four-phase model consisting of design, conduct, evaluation, and feedback [33].

In experiential learning, the instructor plays a pivotal role in creating a structured and supportive environment for students throughout the design and conduct phases [33],[72]. The learning experience is carefully designed to facilitate active engagement and experiential opportunities [33], [53]. Subsequently, in the evaluation phase, the instructor assesses the specific learning outcomes achieved through the experience, followed by providing feedback to the students [33], [72].In contrast, authentic learning focuses on exposing students to real-world problem-solving scenarios based on their in-class experiences. By engaging in authentic tasks, students have the opportunity to develop and refine both soft and hard employable skills that are aligned with market demands [81].In our research, we choose to adopt the authentic learning approach instead of experiential learning to prepare a highly employable cybersecurity workforce with expertise in Kubernetes security misconfiguration analysis.

Herrington et al. described the usefulness of engaging students in reflective and intentional learning [38].Researchers applied authentic learning in learning security threats in machine learning models [4], improving competency in real cybersecurity incidents [49], improving students competency geospatial information system(GIS) skills [6]. Prior research has successfully integrated authentic learning-based exercises into various domains, such as secure software development in mobile computing [83], resulting in improved self-efficacy and confidence among students. Authentic learning has also been applied to enhance learning in secure infrastructure-as-code (IaC) development, enabling students to gain insights into secure IaC practices [89]. Researchers applied authentic learning in learning security threats in machine learning models [4], improving competency in real cybersecurity incidents [49], improving students competency geospatial information system(GIS) skills [6]. We take motivation from prior work on authentic learning in various domains and design authentic learning exercises for practitioners to learn Kubernetes security misconfigurations.

2.2 Related Works in the Domain of Kubernetes

We follow Garousi et al.'s [31] recommendations to conduct a systematic literature review study on Kubernetes. To identify necessary peer-reviewed publications, we use five scholar databases, namely, (i) ACM Digital Library, (ii) IEEE Xplore, (iii) Springer Link, (iv) Science Direct, and (v) Wiley Online Library. We use these five scholar databases for our MLR study because Kuhrmann et al. [63] recommend these databases to use in systematic mapping studies and systematic literature reviews. Following Garousi et al.'s [31] guidelines, we apply an inclusion and exclusion criteria to filter irrelevant search results we identify a set of 105 publications from 3,856 peer-reviewed articles. Each of the publications' names are listed in Table A1 of the Appendix. We index each publications as 'P#', for example the index '1' refers to the publication 'Modelling performance & resource management in kubernetes'. We identify the topics that have been researched in the area of Kubernetes by applying qualitative analysis on the content of the 105 publications. Through our qualitative analysis, we identify 14 research topics. A publication can belong to multiple topics implying that the identified topics are not orthogonal to each other.We provide a mapping between the research topics and publications in Table 2.1. The description of each of the research topic is given below:

Performance Evaluation (50): Performance evaluation is the category of peer-reviewed publications that investigates performance issues in Kubernetes-based deployments. We observe this category of publication to include two sub-categories:

(i) *Technique for Performance Improvement*: Publications that belong to this category proposes and evaluates techniques that can improve a Kubernetes-based deployment. For example, in P18, the authors propose a technique called 'AlloX', and evaluated the performance improvement obtained by Allox for TensorFlow [1]. In P19, the authors proposed and evaluated a configuration tuning tool called 'Accordia' that generates configurations so that performance overhead is reduced for resource-intensive software.

Table 2.1: Mapping Between Publications and Research Topics

Topic	**Publication Index**	**Count**
Performance Evaluation	P1, P2, P6, P7, P14, P17, P18, P19, P21, P22, P23, P24, P27, P35, P37, P39, P41, P42, P43, P46, P47, P51, P52, P53, P55, P57, P58, P59, P60, P61, P67, P68, P69, P70, P73, P75, P77, P78, P79, P84, P86, P87, P91, P95, P96, P99 ,P101, P103, P104, P105	50
Resource Allocation	P1, P2, P4, P5, P6, P8, P16, P18, P19, P21, P22, P23, P24, P26, P27, P28, P32, P41, P50, P51, P52, P54, P55, P57, P58, P62, P70, P73, P77, P78, P79, P80, P81, P82, P83, P84, P86, P88, P91, P95, P99, P102, P104, P105	44
Internet of Things (IoT)	P5, P14, P34, P35, P38, P45, P46, P47, P54, P57, P62, P63, P76, P86, P102, P104	16
Networking	P6, P7, P15,P45, P47, P54, P56, P64, P65, P76, P85, P86, P87, P100	14
Data Mining & Machine Learning	P9, P11, P14, P18, P25, P51, P71, P79, P80, P93, P98	11
Microservice Orchestration	P42, P59, P66, P72, P74, P90, P92, P94	8
Security	P3, P33, P47, P66, P92, P93, P100	7
Fault tolerance	P10, P17, P44, P59, P96, P97	6
High Performance Computing	P68,P71	2
Logging & Monitoring	P30, P60	2
Configuration Abstraction	P20	1
Database Management	P12	1
Electronic Vehicle	P49	1
Discrete Time System Simulation	P36	1

Similarly, in P53, the authors propose 'ConfAdvisor' to improve container performance. In P23, the authors propose a tool called 'KubeShare' that allows graphics processing units (GPU)-based deployments using Kubernetes. In P52, the authors propose a technique to minimize CPU consumption when the CPU resources are shared among co-located containerized software. In P91, the authors propose a technique that uses

the non-dominated sorting genetic algorithm II (NSGA II) to optimize container CPU and memory.

(ii) *Performance Benchmarks*: Publications that belong to this category investigate and compare performance of Kubernetes-based deployments using curated data benchmarks. For example, in the publication P37, P41 and P43 the authors investigated performance comparison of Kubernetes with other deployments tools, performance of Kubernetes in AWS, Azure and GCP platforms and performance comaparison with Docker Swarm and Kubernetes respectively.

Resource Allocation (44): Resource allocation is the category of peer-reviewed publications that proposes and evaluates techniques on how Kubernetes can be configured so that resources are efficiently allocated for one our multiple software deployments using Kubernetes. We observe prior research to apply a diverse set of algorithms, such as search-based algorithms, graph algorithms, and machine learning algorithms to efficiently allocate resources. For example, in P8, the authors use search-based algorithms, namely, the ant colony algorithm [26], and the particle swarm optimization algorithm [102] to develop a scheduling model for Kubernetes-based deployments. In P21, authors use BestConfig algorithm [112] and Bayesian optimization [5] to find cost-effective resource allocation policies for SLOs in Kubernetes. In P26, the stable marriage algorithm [75] is used to find compatible hosts and containers in order to achieve the best deployment with respect to deployment speed. In P55, the authors provide an effective resource allocator for containers running on the Kubernetes cluster. In P80, authors use deep reinforcement learning [65] to allocate resources for deployments in Kubernetes. In P105, authors use a graph algorithm called the minimum cost flow algorithm [35] where resources are allocated by representing each container request with a graph.

Internet of Things (16): Internet of things (IoT) is the category of peer-reviewed publications that investigates how Kubernetes can be used for IoT-based software applications. Peer-reviewed publications that belong to this category focus on improving network latency,

scheduling, and fault tolerance of IoT applications. For example, In P38, the authors propose a fault-tolerant architecture for IoT applications in the cloud. In P45, the authors propose an extension to Kubernetes called 'KubeEdge' architecture with a network protocol stack called 'KubeBus' for IoT applications. In P62, the authors propose a custom Kubernetes scheduler where the nodes decide scheduling for IoT agents.

Networking (14): Networking is the category of peer-reviewed publications that investigates networking-related challenges in Kubernetes-based deployments. For example, in P64, the authors propose a remote direct memory access (RDMA) architecture to control network bandwidth in Kubernetes. In P65, the authors propose a framework to automatically configure virtualized networks with Kubernetes. In P85, the authors propose a solution for monitoring vehicular networks provisioned using Kubernetes. In P87, the authors analyze performance bottlenecks for container network interface (CNI) plugins used in Kubernetes.

Data Mining & Machine Learning (11): Data mining & machine learning is the category of peer-reviewed publications that investigates how software projects that use data mining and machine learning algorithms can be deployed in Kubernetes. For example, in P71, the author uses Kubernetes to design and deploy experiments for a data mining application used in particle imaging [11]. In P98, the authors propose 'JOVIAL' a cloud-based data mining platform that can be used for astronomical data analysis with JupyterHub and Kubernetes.

Microservice Orchestration (8): Microservice orchestration is the category of peer-reviewed publications that investigates techniques on how to orchestrate microservice-based software applications while maintaining availability. For example, in P59, the authors propose a strategy to improve availability of microservices that relies on the state of the service by implementing state controller support for Kubernetes. In P72, the authors propose a new framework to support synchronization among microservices in Kubernetes/Openstack and test various use cases. In P74, the authors compare the deployment of microservices in

CI/CD pipelines with Rundeck, Docker, Kubernetes and report that Kubernetes provides the most efficient way to achieve highly available and scalable microservices.

Security (7): Security is the category of peer-reviewed publications that investigates techniques to mitigate security weaknesses for Kubernetes. Anomaly detection is one security-related topic that has been addressed by researchers. In P3, the authors propose an anomaly detection tool for detecting anomalies in astronomy data analysis tools that are deployed with Kubernetes. In P93, the authors implement 'KubeAnomaly' a tool for anomaly detection in the Kubernetes cluster, using neural network approaches. Security-focused frameworks have also garnered interest: in P33, the authors propose an automated threat mitigation architecture for Kubernetes that continuously scan containers for vulnerabilities to quarantine and isolate vulnerable containers. In P92, the authors built a security framework for integrity protection for microservices-based systems. In P100, the authors propose a zero-trust secure design for a Kubernetes-based data center. Zero-trust refers to the concept that requires all users to be authenticated, authorized, and continuously validated before being granted or keeping access to software and data [51].

Fault Tolerance (6): Fault tolerance is the category of peer-reviewed publications that proposes frameworks to increase reliability for Kubernetes. For example in P96, the authors propose a Kubernetes Multi-Master Robust (KMMR) platform to facilitate robust fault tolerance of Kubernetes.

High Performance Computing (2): High performance computing (HPC) is the category of peer-reviewed publications that investigates techniques on how to efficiently provision HPC applications on Kubernetes. For example in P68, the authors discuss how Kubernetes can be used to deploy HPC applications. The authors further compare Kubernetes-based deployments with Docker Swarm, and bare metal deployments with respect to memory and network bandwidth. The authors of P68 observe Docker Swarm to outperform Kubernetes.

Logging & Monitoring (2): Logging & monitoring is the category of peer-reviewed publications that investigates how logging can integrated in Kubernetes-based deployments.

For example, in P30, the author proposes a technique to mitigate challenges related to logging in pods and containers.

Configuration Abstraction (1): Configuration abstraction is the category of peer-reviewed publications that investigates how novel configuration abstractions can be conducted for Kubernetes. The only publication belonging to this category is P20, where the authors propose 'Isopod' that directly identifies and abstracts Kubernetes objects using the Kubernetes API instead of using Kubernetes manifests. The authors of P20 reported that YAML-based Kubernetes manifests are untyped, can contain wrong indents, and miss important fields, which necessitates abstractions of Kubernetes objects using the Kubernetes API.

Database Management (1): Database management is the category of peer-reviewed publications that investigates how database management tools can be provisioned using Kubernetes. The only publication belonging to this category is P12, where authors propose the Greenplum Database for Kubernetes (GP4K) tool to aid database administrators in automatically deploying databases in Kubernetes.

Electronic Vehicle (1): Electronic vehicle is the category of peer-reviewed publications that investigates how Kubernetes can be used to simulate behaviors of electronic vehicles. The only publication belonging to this category is P49, where the authors use Kubernetes to simulate electric vehicle fleet behavior in a distributed manner.

Discrete Time System Simulation (1): Discrete time system simulation is the category of peer-reviewed publications that investigates how Kubernetes can be used to simulate discrete time systems. The only publication belonging to this category is P36, where the authors use Kubernetes to simulate a linear multi-variable discrete time system. A discrete-time system is a system that takes a discrete time signal as input and generates a discrete time signal as output [80].

Based on the discussion mentioned above we identified the following under-investigated research areas in the domain of Kubernetes and presented three empirical studies in this dissertation.

- **Systemization of knowledege related to Kubernetes Security**: We do not find any publication that focus on systematization of security practices in Kubernetes in our 105 Kubernetes-related publications. Systematizing available knowledge regarding Kubernetes security practices could support practitioners in securing their Kubernetes installations. In addition, such systematization of knowledge can be beneficial for practitioners who (i) want to understand what activities need to be executed to secure Kubernetes components and (ii) can use the derived list of practices as a benchmark to compare their state of security practices.

- **Pod-related configuration parameters for security attack**: In our set of 105 Kubernetes related publications, we observe researchers use anomaly detection approach and security analysis approach to address security related challenges in Kubernetes [88]. However, such security analysis tools do not provide adequate context to the practitioners how the configuration parameters can cause a security attack in Kubernetes. To address that challenge, we conducted systematic investigation to determine which configuration parameters 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 [12], [88], 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.

- **Authentic learning based exercise for Kubernetes security**: Although, lack of security expert is reported by the practitioners [111] , [62], [71], we observe no academic publication address that challenge and propose an educational approach on how we can train the practitioners to learn Kubernetes security. By utilizing authentic learning, we aim to provide students with practical, hands-on experiences in addressing real-world challenges in Kubernetes security. This approach aligns with our objective of equipping students with the necessary skills and knowledge to meet the demands of the industry in the field of Kubernetes security.

Chapter 3

Systemization of Kubernetes Security-related knowledge

In this chapter, we describe our research study to systematize Kubernetes-related security knowledge by synthesizing Kubernetes security best practices.

3.1 Kubernetes-related Security Best Practices

Systematization of knowledge in Kubernetes can be conducted by analyzing Internet artifacts, such as blog posts and video presentations. Practitioners often report what practices they use in Internet artifacts [32, 34] rather than in academic forums such as conferences. In this study, we synthesize Kubernetes security practices by conducting a grey literature review [39]. A grey literature review is the process of reviewing and synthesizing content included in Internet artifacts, such as blog posts and video presentations [39]. A grey literature review differs from a systematic mapping study or systematic literature review, as in these types of literature reviews, researchers use peer-reviewed scientific articles indexed in scholarly databases. In prior work, researchers have reported that practitioners use Internet artifacts, such as blog posts to report their experiences, recommendations, and the practices they follow. Previously, researchers have systematically studied Internet artifacts to identify challenges in microservices development, identify practices used in continuous deployment [90], identify security practices used in organization who have adopted DevOps [110], and software testing [30].

To systematically synthesize practitioner-reported security best practices for Kubernetes, we answer the following research question:

- **RQ 3.1** What Kubernetes security practices are reported by practitioners?

3.2 Methodology

A brief overview of the methodology of this study is demonstrated in Figure 3.1. We synthesize Kubernetes security practices by conducting a grey literature review [39]. A grey literature review is the process of reviewing and synthesizing content included in Internet artifacts, such as blog posts and video presentations [39]. A grey literature review is different from a systematic mapping study or systematic literature review, as in these types of literature reviews, researchers use peer-reviewed scientific articles indexed in scholar databases. In prior work, researchers have reported that practitioners use Internet artifacts, such as blog posts to report their experiences, recommendations, and the practices they follow. Previously, researchers have systematically studied Internet artifacts to identify challenges in microservices development, identify practices used in continuous deployment [90], identify security practices used in organization who have adopted DevOps [110], and software testing [30]. Our hypothesis is that by systematically analyzing Internet artifacts we can synthesize Kubernetes security practices reported by practitioners.

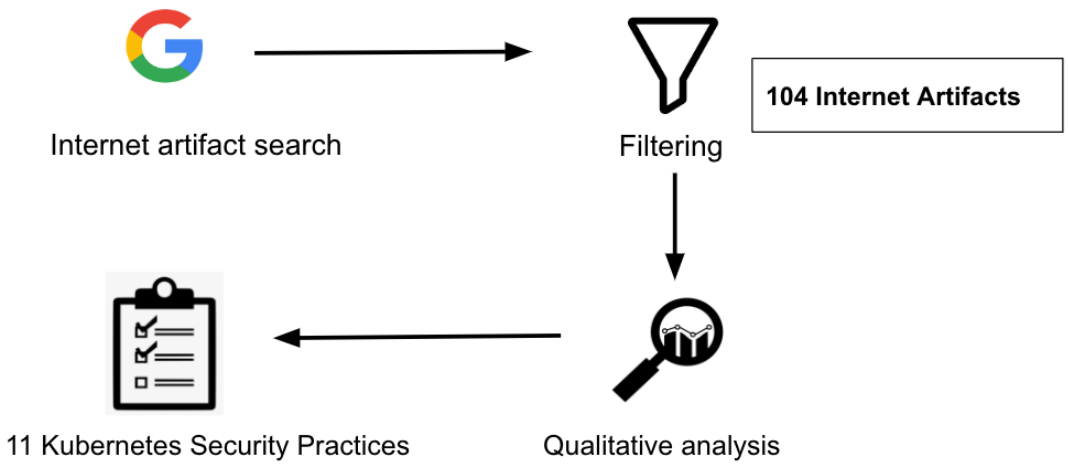

Figure 3.1: A brief overview of the methodology to derive security best practices related to Kubernetes from Internet artifacts.

We use the Google search engine to collect our Internet artifacts. We use three search strings: 'kubernetes security practices', 'kubernetes security good practices', and 'kubernetes security best practices'. After performing the search, we collect the first 100 search results, as Google displays the results in a sorted order based on relevance. We apply inclusion criteria on the collected search results to identify Internet artifacts that discuss security practices for Kubernetes. The inclusion criteria are listed below:

- The Internet artifact is not a duplicate;
- The Internet artifact is available for reading; and
- The Internet artifact discusses security practices for Kubernetes;

We use open coding [98], a qualitative analysis technique, to determine the security practices for Kubernetes. In open coding, a rater observes and synthesizes patterns within unstructured text [98]. Figure 3.2 shows an example of open-coding to derive a category of security best practices in Kubernetes. The first rater is a PhD student with 1 year of experience in Kubernetes. The identified practices are also susceptible to biases of the rater who identified the practices by applying open coding. We mitigate this bias by allocating another rater, who apply closed coding [21] on a randomly selected set of 50 Internet artifacts. Closed coding is the technique of mapping an entry to a predefined category [21]. For each of the 50 Internet artifacts, the second rater examined whether the artifact of interest includes a discussion related to the security practices identified by the first rater. The second author has 3 years of experience in software security. We calculate the agreement rate between the first and second author for the 50 Internet artifacts using Cohen's Kappa [19]. The Cohen's Kappa between the two raters is 0.8, which is substantial [64]. After the closed coding exercise, the first rater and second rater discuss each of their disagreements and resolve conflicts.

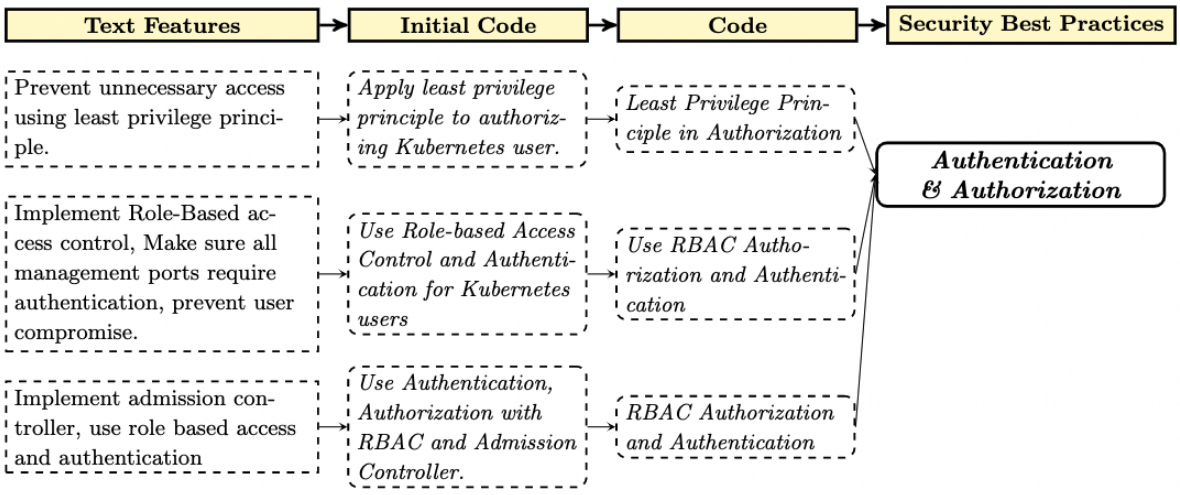

Figure 3.2: An example of open-coding to derive a category of security best practices in Kubernetes

3.3 Results

After applying open coding and closed coding exercise on 104 Internet artifacts we derive 11 best practices for Kubernetes security. Of the 104 Internet artifacts 90.38%, 4.81%, and 4.81% are respectively blog posts, videos and presentations. Among the 11 Kubernetes security best practices mostly discuss about ensuring Authentication and Authorization and Kubernetes-specific policies. In figure 3.3, we have listed 11 security best practices and their occurrences in the curated 104 Internet artifacts. We describe the 11 identified Kubernetes security best practices as follows where the number between parentheses indicates their occurences in the Internet artifacts:

1. **Authentication and Authorization (82)**: The practice of applying authentication and authorization rules to prevent malicious users from getting access and performing unauthorized activities inside the Kubernetes cluster. Authentication in Kubernetes refers to the authentication of API requests through authentication plugins[59]. Authorization in Kubernetes refers to the evaluation of each authenticated API request against all policies to allow or deny the request[59].

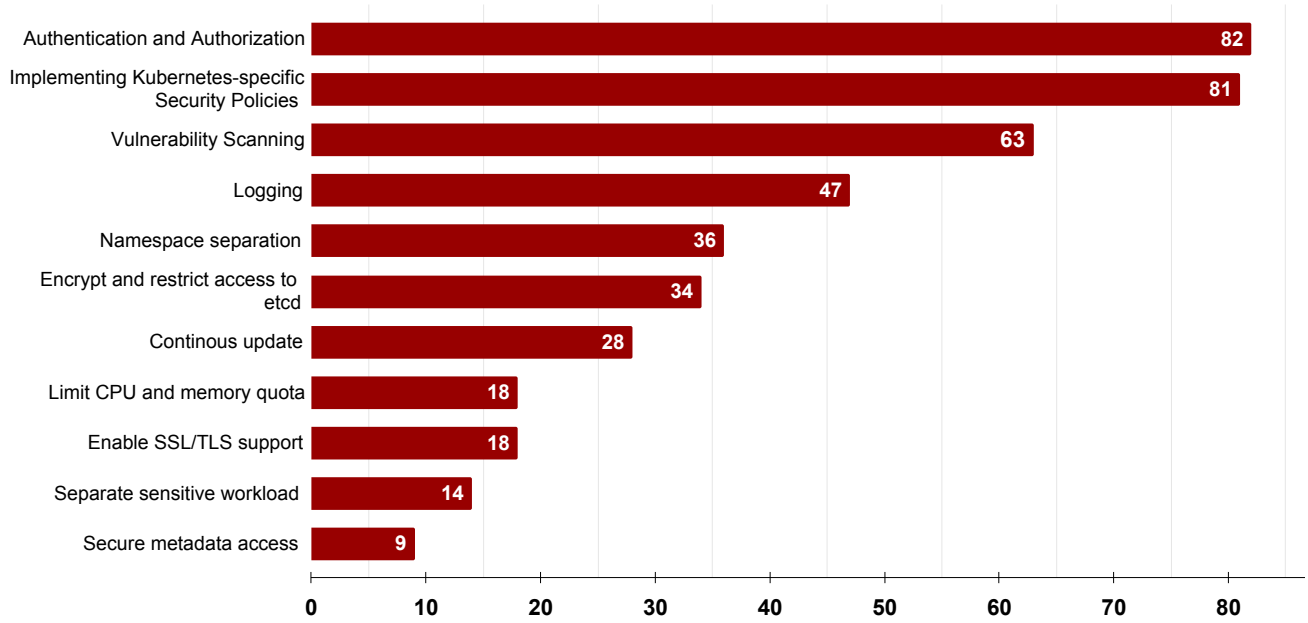

Figure 3.3: The occurrences of security best practices in Kubernetes in the Internet Artifacts

2. **Implementing Kubernetes-specific Security Policies (81):** The practice of applying policies to secure Kubernetes components, pods, and networks of Kubernetes clusters to prevent security breaches.

- *Network-specific policies*: The practice of applying a network policy to protect communication between Kubernetes pods from undesirable network communications. By default, all Kubernetes pods can communicate with other pods. Practitioners recommend policies to restrict traffic between pods, restrict API server access and reducing network exposure to secure the network.

- *Pod-specific policies*: The practice of implementing a policy for pods to apply security context to pods and containers. Pod policies determine how the workloads should run in the Kubernetes cluster. Without defining a secure context for the pod, a container may run with root privilege and write permission into the root file system, which can make the Kubernetes cluster vulnerable. Practitioners recommend containers inside a pod must run as a non-root user with read-only permission and enabling Linux security modules.

- *Generic policies*: The practice of applying a generic security policy to protect Kubernetes cluster components from external malicious users. TCP ports for kubelet, API server, etcd, and network plugins should not be left open and should require authentication to have visibility. Every user in the system should have the least privilege by default.

3. **Vulnerability scanning (63):** The practice of scanning Kubernetes components and continuous delivery (CD) components for vulnerabilities.

- Kubernetes components, such as containers can contain vulnerabilities and malicious malware. If vulnerabilities are present in a Kubernetes cluster, then the entire container orchestration system, and the provisioned applications, become susceptible to attacks. For example, in 2017, researchers found Docker images embedded with malicious malware. Practitioners recommended scanning containers for vulnerabilities with tools,such as 'Dockscan' [1] and 'CoreOS Clair' [2].

- If images and deployment configurations within CD components are not inspected, then it can make the Kubernetes cluster vulnerable to malicious users. The malicious users can gain access at a later point when these images are deployed and may exploit the latent vulnerabilities in Kubernetes production environments. Practitioners recommend pulling images from a trusted private registry and checking for the vulnerability of code and images.

4. **Logging (47):** The practice of enabling and monitoring logs for the Kubernetes cluster. Practitioners recommend that logging should be enabled for (i) applications, (ii) the containers within each pod, and for (iii) Kubernetes clusters for system health checking.

5. **Namespace separation (36):** The practice of separating namespaces so that the resource of one namespace are not shared with another. A 'namespace' in Kubernetes is a

[1] https://github.com/kost/dockscan
[2] https://github.com/quay/clair

logically isolated virtual cluster within the same physical cluster.[59] Creation of separate namespaces enables resources to be isolated between namespaces. If a separate namespace is not created for a resource then the resource gets 'default' namespace.

6. **Encrypt and restrict access to etcd (34):** The practice of encrypting and restricting access to 'etcd', the internal database used by Kubernetes[59]. By default, Kubernetes stores secret data as plaintext in 'etcd'[3]. Practitioners recommend using secret management tools for additional security[59], such as 'Vault'[4] for encryption.

7. **Continuous update (28):** The practice of applying security patches to keep the Kubernetes cluster updated with latest security fixes. Practitioners recommend that Kubernetes users apply updates as well as conducting continuous updates for the deployed applications within the Kubernetes pods.

8. **Limit CPU and memory quota (18):** The practice of limiting CPU and memory to a pod or a namespace so that malicious attacks can be mitigated. By default, all resources in Kubernetes start with unbounded memory requests/limits and unbounded CPU access.

9. **Enable SSL/TLS support (18):** The practice of enabling secure sockets layer (SSL) or transport layer security (TLS) protocol to ensure secure and encrypted communication between Kubernetes components. Enabling TLS between kubernetes api server, etcd, kubelet and kubectl ensures secure communication between cluster components. Practitioners suggest enabling TLS and SSL certificates for Kubernetes components.

10. **Separate sensitive workload (14):** The practice of running sensitive applications on a dedicated set of machines to limit the potential impact of a security breach.

11. **Secure metadata access (9):** The practice of securing the sensitive metadata of the Kubernetes cluster. Practitioners state that the Kubernetes metadata APIs provide a gateway to expose 'kubelet' admin credentials.

[3]https://ubuntu.com/kubernetes/docs/encryption-at-rest
[4]https://www.vaultproject.io

Chapter 4

Motivating Example

We motivate our empirical study further by using Figure 4.1, where we present an example Kubernetes manifest. The manifest is used to specify configurations for a pod called 'sample' with 'nginx' container images, using the namespace 'sample-app-space'. We also observe the manifest to include specifications for role-based access control (RBAC) using `kind: RoleBinding`, `kind: ServiceAccount`, and `kind: Role` objects. In the case of configurations, such as `name` and `namespace`, a practitioner can assign any strings so that the pod 'sample' is deployed with adequate RBAC configurations, 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 values can yield security attacks. Of these nine configurations, (i) 5 are Boolean, (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 nine configurations cause attacks, a Kubernetes user can manually explore all possible combinations for the 9 configurations by accounting for the semantics of pods, RBAC policies, and their interactions. However, such manual exploration is practically impossible as the user has to provision the pod for 1.2×10^{22} possible configuration combinations. Hence, an automated approach is required that can aid in automated determination of what pod-related configurations can cause security attacks. As the focus is on identifying configurations that can cause pod-related attacks, the automated approach should also account for pod states, i.e., the states that a pod traverse upon execution. In the context of Figure 4.1, prior to executing the pod with the provided configurations, a pod will undergo the through following states: 'request initiated', 'request authenticated', and

'request authorized' [59]. Therefore, the automated approach must account for these states unique to pods to determine attack-akin configurations.

To that end, we use model checking to determine attack-akin configurations. Model checking leverages finite state machines, which will allow us to account for the pod-related states [17]. Our hypothesis is that use of model checking will be useful to determine: (i) if the 9 configuration combinations can lead to a security attack, and (ii) what configuration values can be used to demonstrate the attacks. We describe our model checking-based approach and findings in Section 5.

```
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
...
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: sample-app-space
...
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: sample-app-space
roleRef:
  kind: Role
  name: app-role
subjects:
- namespace: sample-app-space
  kind: ServiceAccount
  name: app-service-account
```

Figure 4.1: Example code snippet to demonstrate attack-akin configurations.

Chapter 5

Configuration Parameters that Facilitate Security Attacks for Kubernetes Pods

Despite the reported benefits, security is identified as one of the prime concerns for practitioners who utilize containers for the construction and deployment of applications in Kubernetes. According to the State of Kubernetes and Container Security Report 2020, published by Stackrox, it is suggested that containerized application deployment was delayed by 44% of organizations due to security concerns. Furthermore, it is stated in the report that security incidents were experienced by 94% of organizations in the last 12 months, with 69% of these incidents being attributed to misconfiguration-related security issues.[92].

To identify security misconfigurations in source code, researchers use static analysis tools to detect code smells, such as in Infrastructure as Code (IaC) scripts [85], [86] and misconfiguration in Kubernetes manifests [88]. However, the precision required for evaluating source code soundness and completeness cannot be achieved by static analysis tools alone [48]. The generation of potential attack scenarios or the demonstration of a realizable attack path when identifying a security misconfiguration in a source code file is not within the capabilities of security analysis tools [14], [58], [3], [23] as they report one misconfiguration at a time without any additional context for the practitioners. Researchers use model checking to identify configurations that can cause attacks in cellular network protocols [45], [48]. In this research, we combine model checking with static analysis tool to identify the pod-related configuration parameters that can cause security attacks. While empirical research related to Kubernetes have addressed topics related to quality assurance [13], [88] there is a lack of investigation on what configuration parameters facilitate security attacks. We provide the

overview of our methodology in Figure 5.1.

In this research, we answer the following questions:

- **RQ 5.1** What configuration parameters facilitate security attacks for Kubernetes pods?
- **RQ 5.2** How frequently do identified configuration parameters appear in Kubernetes manifests?
- **RQ 5.3** What states in the pod lifecycle map with security attacks for Kubernetes pods?

5.1 Methodology for RQ 5.1

5.1.1 Threat Model

In our threat model, we assume that Kubernetes manifests are developed by system administrators without malicious intent, but still can include security misconfigurations. Our assumption is consistent with prior research that shows the existence of known security misconfigurations [88]. In our threat model, a malicious user, i.e., an attacker attempts to launch attacks against Kubernetes-based containers by leveraging one or multiple combinations of these misconfigurations. The goal of the attacker is to (a) gain unauthorized access and/or (b) disrupt availability for any Kubernetes-based container infrastructure. If successful, the attacker 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.

Pod Security Requirement Derivation from Internet Artifacts

The phase of a pod during its life cycle is influenced by cluster configurations and environment variables, as described in Section 2.1.3. To replicate the behavior of a pod

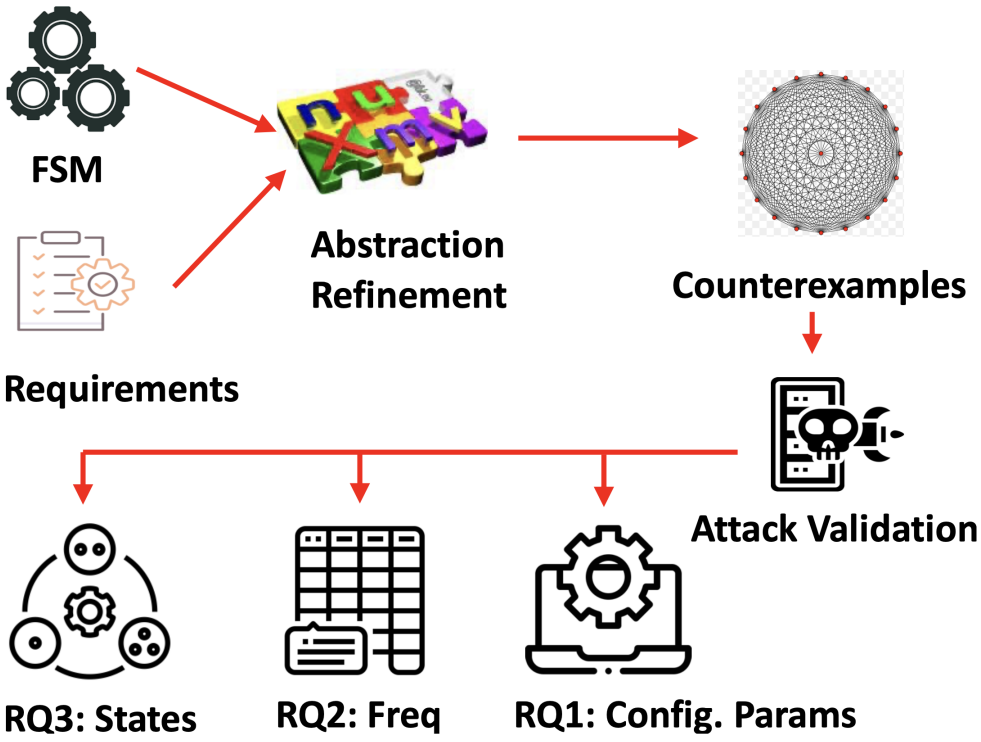

Figure 5.1: An overview of methodology to identify pod-related configuration parameters that can facilitate security attacks

throughout its life cycle, we develop a finite state machine using NuXMV. In our model in NuXMV, we incorporate the Kubernetes cluster configurations and environment variables as pod properties, which define the pod's behavior. We use open coding to identify codes for pod properties related to pod security requirements from the Internet artifacts. Open coding is a qualitative analysis technique that identifies the underlying code from unstructured text data [99].

Pod Security Guideline Extraction from Internet Artifacts

We conduct a grey literature review [31] on available Internet artifacts that discuss the pod security requirements. We use the Google search engine to collect the internet artifacts in incognito mode. We use two search strings: "Kubernetes pod security guidelines" and "Kubernetes pod security rules". We start our Internet artifact search with the initial search

string "Kubernetes pod security guidelines". We add the later search string as we observe the practitioners often refer pod security rules instead of pod security guidelines. Then, we search with the search string "Kubernetes pod security rules". We collect the first 100 search results for each of the search strings. To perform the filtering for the Internet artifacts, we apply the following exclusion and inclusion criteria according to the guideline of Garousi et al. [31].

Exclusion Criteria: We adhere following guideline to exclude an Internet artifact.

- The Internet artifact is not written in English.
- The Internet artifact is published before 2014, as the initial version of Kubernetes was released in 2016. [73].

Inclusion Criteria: We apply the following criteria to include an Internet artifact.

- The Internet Artifact is available for reading.
- The Internet artifact is not a duplicate.
- The content of the Internet artifact explicitly describes the Kubernetes security guideline that includes pod related security guidelines.

After combining two search results for our search strings, we remove the duplicates. We read each of the Internet artifact, and filter 21 Internet artifacts to gather pod properties related to pod security requirements. Figure 5.2 illustrates our Internet artifact collection process.

We apply an open coding technique to derive pod properties related pod security requirements from the Internet artifacts. In Figure 5.3, we illustrated our open coding process. First, we collect the text from the Internet artifacts that discuss pod-related security and form initial code. In Figure 5.3, we create initial code *"Admission controller can scan images and block insecure images"* and *" Admission controller can scan images*

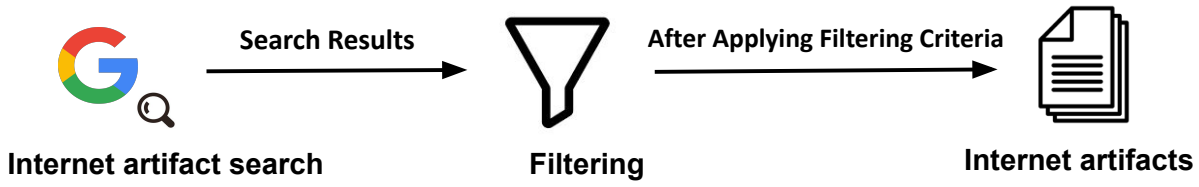

Figure 5.2: Internet artifact search and filtering process

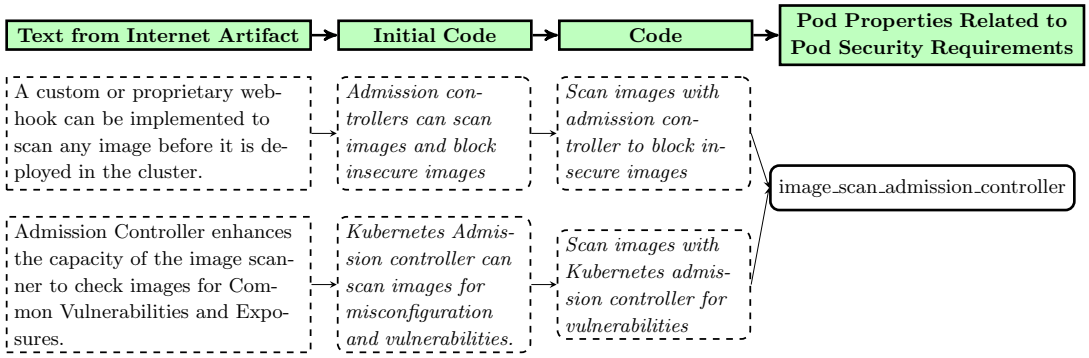

Figure 5.3: A description of our open coding process to derive pod properties related parameters from Internet artifacts.

for misconfiguration and vulnerabilities." respectively. In the next step, we identify codes *"Scan images with Kubernetes admission controller to block insecure images"* and *"Scan images with Kubernetes admission controller for vulnerabilities"* from the initial code. Finally, we construct the pod properties as parameters related to pod security requirement as a `boolean` variable, such as `image_scan_admission_controller`. If the value of the `image_scan_admission_controller` is `true`, then the admission controller for scanning container image is present in the Kubernetes cluster. Altogether, we derive 71 pod security related properties from the Internet artifacts. We represent our derived pod properties with their appearance frequency in Table 5.1.

We also define the pod security requirements from the 21 Internet artifacts. To construct the pod security requirement, we translate the pod property to corresponding NuXMV LTL

formulas for pod security 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 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'. Table 5.2 describes our pod security requirements and corresponding NuXMV LTL formulas. We use the parameters related to pod security requirements gathered from the Internet artifacts to construct the NuXMV LTL formulas for the pod security requirements. For instance, in the first row of the Table 5.2 we describe pod security requirement, "*containers with unnecessary privilege cannot be executed inside a pod*". We construct corresponding LTL formula in NuXMV such as if the container has `CAP_SYS_ADMIN` privilege then there will not be a `over_privileged_container` while the `pod_state = pod_running`.

We translate the pod security requirements to the NuXMV LTL property as follows:

```
LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN)
-> G X(!over_privileged_container & pod_state=pod_running))
```

In the propositional LTL formula, we use `TC_18` and `TC_19` & `new_pod_creation_request` as constants. Here `TC_18` and `TC_19` are transition conditions and both are set to `False` and state variable `new_pod_creation_request` is set to `True`.

5.1.2 Translation of Pod Life Cycle to Finite State Transitions

A pod is considered as the unit entity in the Kubernetes-based container orchestration. In Kubernetes, each pod has a definite life cycle as described in Section 2.1 in Figure 2.1.3. When the Kubernetes API server accepts pod creation requests, it creates a pod object, and the pod goes to the `Pending` phase. The scheduler in the Kubernetes API server schedules the pod object to a node. In the first step, the scheduler finds a set of candidate nodes and

assigns ranks to the candidate nodes to find the most suitable node for the pod object. In the second step, the scheduler binds the feasible node for a pod object. A pod remains in the `Pending` phase until the kubelet in the assigned node receives the pod object specification and provides the container runtime engine with the image to start a container. The pod has its IP address in the `Running` phase and can communicate with all other pods on the node or any other node in the Kubernetes cluster. The pod can be accessed outside the Kubernetes cluster as a service with a service IP address managed by kube-proxy. Each pod is assigned storage while in the `Running` phase, and the Kubernetes volume abstracts the storage of a pod. Kubernetes destroys the ephemeral volume of a pod when a pod terminates. However, a pod can have a persistent volume that exists beyond the life cycle of a pod. If at least one container terminates with a non-zero exit code, the pod goes to `Failed` phase. If the container runs again after the restart, the pod goes to the `Running` phase. If the kubelet fails to communicate with the Kubernetes API server from the node, then the pod goes to `Unknown` phase. The pod controller replaces the pod in the node in case of pod failure or another node in case of node failure. If all the container terminates with zero exit code the pod terminates in `Succeeded` phase.

When a practitioner sends a request to the Kubernetes API server to create a pod, we observe a temporal ordering of events for a pod when there is a transition of the pod phase. Each event depends upon pod configurations, conditions and Kubernetes cluster states. In the sequence diagram in Figure 5.4, we illustrate only the sequence of events at the pod creation time. In section 2.1, we discuss the events while a pod reaches its `Running` phase from the `Pending` phase. We list the temporal events for the scenario as follows:

Event 1: The practitioner initiates a pod creation request to the API server.

Event 2: The API server authenticates the request. This event can lead to two events: successful or failed authentication.

Event 3: Upon successful authentication, the API server can authorize or fail to authorize the request.

Event 4: The API server writes the information to the etcd database and returns a response to the practitioner.

Event 5: Upon successful authentication, and authorization, the Kubernetes API server creates a pod object and sends the pod specification to Scheduler, which watches for a new pod.

Event 6: Upon successful node allocation to the pod, the Kubernetes API server binds the pod to a node and stores the desired pod state in the etcd database.

Event 7: The kubelet agent in the worker node watches for the pod bound to it.

Event 8: The API server sends the pod specification to the kubelet worker node.

Event 9: Upon receiving the pod specification, the kubelet attempts to pull the container image from the registry. If the kubelet in the worker node can pull the image from the registry, it sends the image to a container runtime such as Docker engine. Upon failure to pull the image from the registry, kubelet reports an error to the Kubernetes API server.

Event 10: If the container runtime, such as the Docker engine, can create a container from the image or encounter any issue, the kubelet updates the API server regarding the pod update.

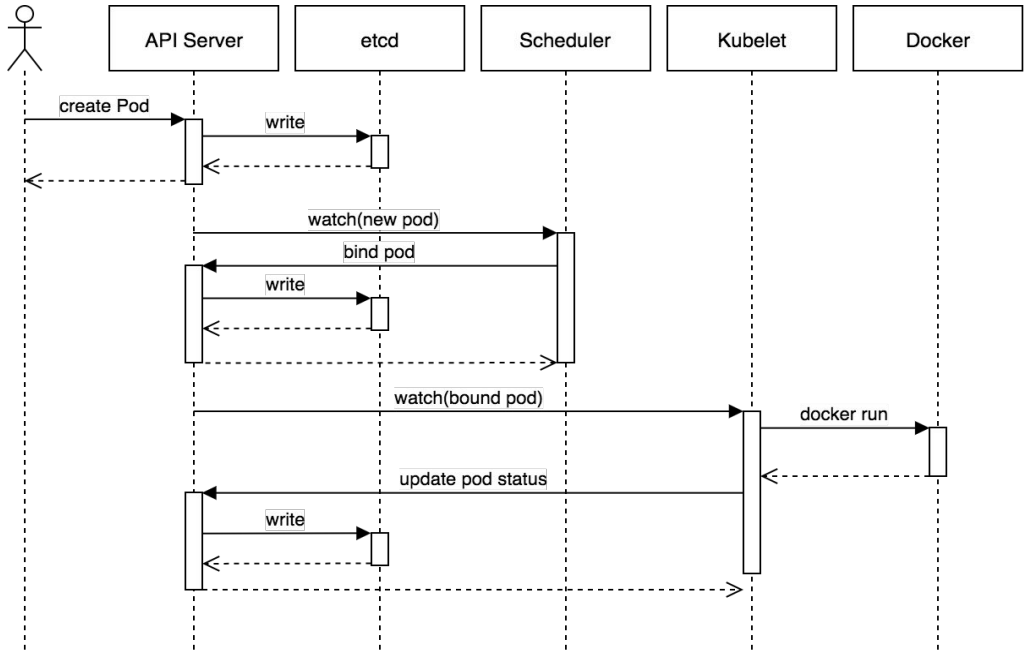

Figure 5.4: The sequence diagram of pod events after a practitioner requests for pod deployment

Table 5.1: Identified Pod Properties from the Internet Artifacts

Pod Properties	Count
Privilege escalation (privilege_escalation)	10
System admin capability (CAP_SYS_ADMIN)	10
Run as user (run_as_user)	10
Privileged security context (security_context_privileged)	9
Drop container capabilities container_DropCapabilities	9
Allow container capabilities (container_AllowedCapabilities)	9
Default container capabilities (container_DefaultCapabilities)	9
Host IPC enabled (hostIPC_enabled)	8
Linux security module SELinux enabled (lsm_SELinux_enabled)	8
Linux security module Seccomp enabled (lsm_SECCOMP_enabled)	8
Read only root file system (read_only_root_file_system)	8
Running as non root (running_as_NON_ROOT)	8
Pod Restricted Admission (pod_admission_RESTRICTED)	8
Host PID enabled (hostPID_enabled)	7
Host Network enabled (hostNetwork_enabled)	7
Default network policy deny everything (default_network_policy_all_ns_deny_everything)	7
FS group (fsGroup)	7
Supplemental group (supplementalGroup)	7
Run as group(run_as_group)	7
Host path enabledhost_path_enabled	6
Admission controller image scan (admission_controller_image_scan)	6
Default namespace (default_namespace)	6
Pod admission baseline (pod_admission_BASELINE)	6
Pod admission privileged (pod_admission_PRIVILEGED)	6
Enforce pod admission controller (pod_admission_ENFORCE)	6
Namespace resource quota enabled (namespace_resource_quota_enabled)	6
Pod CPU and memory request limit enabled (pod_cpu_memory_request_limit_enabled)	6
Pod CPU memory limit enabled (pod_cpu_memory_limit_enabled)	6
Namespace resource quota enabled (namespace_resource_quota_enabled)	6
Host process enabled (hostprocess_enabled)	5
Host port enabled (host_port_enabled)	5
Linux security module apparmor enabled (lsm_APPArmor_enabled)	5
Use of base container images (use_of_base_container_images)	5
Avoid tags and latest image tags (avoid_tags_and_latest_tags)	5
Use sha256 digest for image (use_sha256_digest_for_image)	5
Admission image policy webhook (admission_image_policy_webhook)	5
Avoid default service account (avoid_default_service_account)	5
Default proc mount (default_proc_mount)	4
Avoid environment variables in images (avoid_env_variables_images_images)	4
Use secret in images (use_secret_in_images)	4
Service account automount token (service_account_automount_token)	4
Container network interface supports network policy (cni_supports_network_policy)	4
Pod security exemption for user (pod_security_exemption_user)	4
Pod security exemption for workload pod (pod_security_exemption_workload_pod)	4
Pod security exemption for namespace (pod_security_exemption_namespace)	4
Minimal distroless image (minimal_distroless_image)	3
Unprivileged user for image build (unprivileged_user_for_build_image)	3
Capability NET_RAW (CAP_NET_RAW)	2
Docker socket enabled (docker_socket_enabled)	2
Volume usage permission (volume_usage_permission)	2
Sysctl namespaced (sysctl_namespaced)	2
Image pull policy (image_pull_policy)	2
Use external secret storage (use_external_secret_storage)	2
Network policy between pods (network_policy_between_pods)	2
File system (FS) group change policy (fsGroupChangePolicy)	2
Admission namespace lifecycle (admission_namespace_lifecycle)	1
Get secret (GET_secret)	1
List secret (LIST_secret)	1
Watch secret (WATCH_secret)	1
All verb secret (ALL_verb_secret)	1
All verb role (ALL_verb_role)	1
All verb resources (ALL_verb_resources)	1
Cluster admin (ClusterRole_cluster_admin)	1
Security context enabled (security_context_enabled)	1
Admission always pull images (admission_always_pull_images)	1
Liveness probe enabled (livenessprobe_enabled)	1
Readiness probe enabled (readinessprobe_enabled)	1
Limit node PID (limit_node_PID)	1
Limit pod PID (limit_pod_PID)	1
Pod eviction policy (pod_eviction_policy)	1

Table 5.2: Pod Security Requirements and Corresponding LTL formula

Pod Security Requirements	**Corresponding LTL Formula in NuXMV**
Containers with unnecessary privilege cannot be executed inside a pod	`LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN) -> G X(!over_privileged_container & pod_state=pod_running))`
Admission controller must be enabled for pods	`LTLSPEC G (!TC_18 & !TC_11_2 & !TC_19) -> G(!admission_image_signature_verification) -> G X (!admission_control_bypass & pod_state = host_system_access)`
TLS encryption must be enabled for pod-related communication	`LTLSPEC G (TC_18 = FALSE & !TC_15 & !TC_25 & !TC_26 & !TC_19) -> G((!mTLS_encryption) -> G X(!network_misconfiguration & pod_state=host_network_access))`
Network policies must be enabled for pods	`LTLSPEC G (TC_18 = FALSE & !TC_15 & !TC_25 & !TC_26 & !TC_19) -> G((!default_network_policy_all_ns_deny_everything) -> G X(!network_request_other_workload & pod_state=service_exposed))`
Restrict malicious users in obtaining secrets from the container inside a pod	`LTLSPEC G( !TC_11_2 & !TC_19) -> G (!container_secret_exfiltration)`
Restrict permission to read/watch secret	`LTLSPEC G (!TC_26 & !TC_15 & !TC_19 & !TC_25) -> G(WATCH_secret) -> (G X (!host_secret_exfiltration))`
Unnecessary permission to host file system need to be revoked	`LTLSPEC G (!TC_18 & !TC_19 & !security_context_run_as_user) -> G ((fsGroup) -> G X(!host_file_system_access))`
Images with incorrect configurations can not be pulled from an unauthorized registry	`LTLSPEC G (!TC_15 & !TC_11_2) -> G(!use_sha256_digest_for_image) -> G (!misconfigured_image)`
Any container running in a pod must specify resource limits	`LTLSPEC G (!TC_18 & node_resource_quota_enabled) -> G (!pod_eviction_from_node)`

To answer RQ 5.1, we need to represent above-mentioned pod phases in a manner so that the each pod events belong to pod phase. 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 life- cycle 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.

5.1.3 Encoding Logic Formula for Finite State Transitions and Requirements Mitigation of Challenges

We address two primary challenges while encoding a finite state model and constructing logic formula for the pod life cycle. We organize our challenges as below:

Challenge #1: Search-space explosion: Kubernetes pod-related events depend upon the configurations of the pod and the Kubernetes cluster. Each pod event in Kubernetes occurs due to a specific combination of configurations. In the Kubernetes cluster, the number of combinations of configurations of the pod and the Kubernetes cluster makes the search space computationally expensive. We construct 125 state variables to abstract pod and Kubernetes cluster configurations. Each state variable is a configuration in Kubernetes cluster that helps in describing the pod behaviour a pod state. Among 125 state variables, 71 of the state variables are the pod properties related to pod security requirements as described in Table 5.1. The remaining 54 state variables helps describe pod behaviour in Kubernetes cluster. In total, our FSM for a pod has 33 states, 125 state variables, and 43 transition conditions. The search space of our FSM for a pod has 2^{125} search space. Hence, verifying the pod security requirements as a propositional formula built from a set of 125 state variables relates to boolean satisfiability problem (SAT) [20]. The SAT problem is an NP-complete problem. NP-complete problems can not be solved in polynomial times but can be verified in polynomial time. To verify SAT problem, the SAT solver is used as a verification tool. The SAT solver uses approximation algorithms to verify a boolean formula

in polynomial time. We use NuXMV model checker that uses SAT solver to reduce the search space of our FSM for a pod to verify pod security requirements in polynomial time.

Challenge #2 Intertwined component interactions: Identifying the pod events due to the complex interaction between the Kubernetes components and pods is one of our primary challenges in building a finite state model (FSM) for a pod. Each pod phase in the pod life cycle depends on the container state, Kubernetes components, configurations and pod conditions. We mitigate this challenge by identifying the transition conditions between the FSM states for a pod as a form of propositional logic.

Construction of Finite State Machine

We model the events of the pod life cycle as a deterministic finite state machine. Our state machine is 3-tuple (S, Σ, Γ) where S is the finite nonempty set of states, Σ is a finite nonempty set of transitions and Γ is a finite nonempty set of transition conditions. We define $s_i \epsilon S$ is the initial state of the pod,and $s_o \epsilon S$ is the final state of the pod. transition action $\alpha \epsilon \Sigma$ is a finite set of transitions and transition conditions $\gamma \epsilon \Gamma$ is a set of transition condition.

Our FSM-based approach alleviates the challenge of accounting for the stateful nature of configuration parameters through the usage of state variables, and transition conditions for the derived FSM. We use Figure 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 'Pending' phase. In this particular FSM has five states, where initial state is `request_initiated`. Σ represents as a set of the four transitioning conditions in forms of variable assignments, namely, `authorized_bootstrapping`, `valid_API_request`, `valid_admission_controller`, and `unsuccessful_authorization`. 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`.

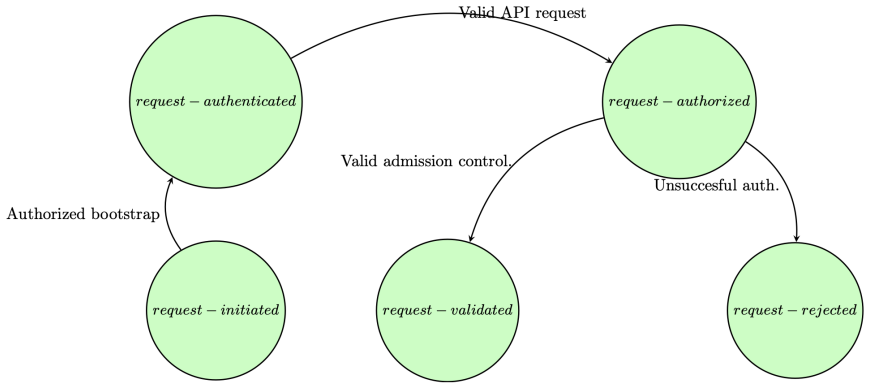

Figure 5.5: A finite state machine representing the a subset of the states related to the 'Pending' phase of Pod

We construct the FSM model for a pod and transition condition from one state to another state with the combination of state variables defined as propositional logic formula. Model checking is a method to check if a system's finite state model (FSM) fulfils a particular set of specifications [10]. Model checking method explores all possible states of a system and all possible values for the state variables [10]. We define a pod state as the status of a pod in the Kubernetes cluster. In addition, we define state variables as configurations that can describe a pod state. A transition condition is an expression where a combination of state variables allows the transition from a state to a subsequent state. A valid transition triggers a pod-related event to transition into a subsequent pod state. For instance, when the Kubernetes API server starts the pod creation request, the pod enters into `request_pod_creation_initiated` state. If the pod stays in the `request_pod_creation_initiated` state and the state variable `request_accepted_k8s_api_server`, then the transition condition for the subsequent state `desired_pod_state_stored_in_etcd` will be `true`. In this case, a valid transition and pod event will occur, as Figure 5.6 describes. If a state in the FSM is found under the property or specification that violates the property then the model generates a counter-example. A counter-example describes the execution step from the initial step to where the system violates the specific property. We

use 71 pod properties related to pod security requirements in our pod state model as described in Table 5.1. Apart from these 71 properties related to pod security requirements, we use 54 additional state variables as a parameter to define the transitions and transition conditions of our FSM for a pod. We grouped the 71 pod properties related to pod security requirements into 16 groups.

Each parameters related to pod security requirement can belong to multiple groups. The intuition behind grouping the parameters related to pod security requirements is to cluster them into a similar group so that we can construct propositional logic formula to verify the pod security requirements in the NuXMV [16]. For instance, `hostPID_enabled`, `hostIPC_enabled`, `CAP_SYS_ADMIN`, `host_path_enabled`, `host_port_enabled`, `hostprocess_enabled`, and `hostNetwork_enabled` belongs to one single group `host_namespace_access`, because any of the two parameters can be used to access host namespace. We define the relationships as `hostPID_enabled | hostIPC_enabled |hostprocess_enabled| hostNetwork_enabled | CAP_SYS_ADMIN | host_path_enabled | host_port_enabled` If a container can access the shared namespace, it can potentially extract underlying host information such as host process id, host network, and even host file system. Similarly, all of the variables also belongs to `over_privileged_container`. The privilege to access host namespace gives the container unnecessary privilege to compromise the underlying host.

Model checking is a method to check if a system's finite state model (FSM) fulfils a particular set of specifications [10]. Model checking method explores all possible states of a system and all possible values for the state variables [10]. In the NuXMV, we verify the pod security requirements in our FSM for a pod as a safety property [10]. A safety property dictates that under certain condition bad events will never happen [10]. For model checking, we use counter example guided abstraction refinement (CEGAR) principle [10]. According to the CEGAR principle, we initially use the pod security requirements and our FSM for verification in the NuXMV [16]. If the NuXMV generates erroneous counter-example then

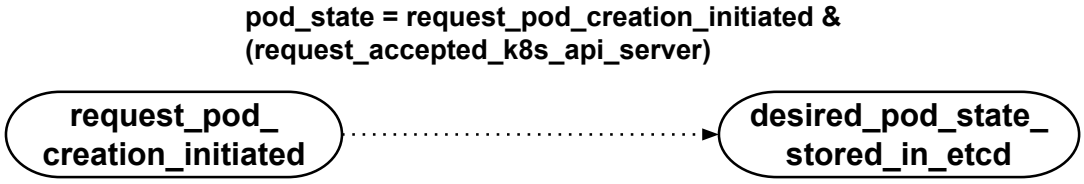

Figure 5.6: A pod state transition of event

we revise our pod security requirements to prevent the erroneous counter-examples. We continue this process for violation of each safety property until we find understandable counter-example. Depending on counter-example output, we set the values of some state variables, transition conditions as constant to prevent erroneous counterexamples. The value of the state variables and transition conditions we set as constant in LTL formula do not change during the execution for verification in NuXMV. For instance, in the following NuXMV LTL formula as described in Table 5.2, we use `TC_18` and `TC_19` & `new_pod_creation_request` as constants. Here `TC_18` and `TC_19` are transition conditions and both are set to `False`.

```
LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN)
-> G X(!over_privileged_container & pod_state=pod_running))
```

5.1.4 Mapping of Pod Events to Finite State Machines

Kubernetes provides support for practitioners to manage containerized applications at scale [59] [15]. Practitioners can install Kubernetes on-premise, on cloud platforms, or a combination of both. A Kubernetes installation is also colloquially referred to as a Kubernetes cluster [59]. A pod is the most fundamental unit of a Kubernetes cluster [59]. A pod groups one or more containers with shared network and storage resources according to the specifications practitioners provide in their Kubernetes manifest. Kubernetes allows namesapces, which provide a mechanism to isolate groups of resources within a Kubernetes cluster.

Using Figure 4.1, we also provide background information on pod lifecycle and states, as our model checking-based approach to answer RQ1 accounts for pod-related states. The

lifecycle of a pod consists of five phases. Each phase consists of one or multiple states, which we encode as described in Section 2.1.3. Transition from one state to another is dependent Boolean conditions, which we refer to as transition conditions.

The first phase of a pod in Kubernetes is 'Pending' with the 'request_initiated' and 'request_pod_creation_initiated' states. The 'kubectl' command line interface is used to create a pod. Next, with two states namely, 'request_authenticated' and 'request_authorized', the Kubernetes API server authenticates and authorizes the request upon receiving the request. If the requests are validated by the admission controller with the 'request_admission_controller_validated' state, then a pod is created and all the configurations for the pod will be stored in etcd with the 'desired_pod_state_stored_in_etcd', 'kubelet_receives_podspec', and 'etcd_updated' states. 'etcd' is a database that uses a key-value mechanism to store all pod-related data [59]. In the context of Figure 4.1, at this phase all configurations for 'sample' will be stored in etcd. Using the 'pod_schedule_bind_phase' state, the Kubernetes scheduler schedules the pod, and the container runtime engine pulls the container image for running the container using the 'image_pulled_from_registry' state. For example, for Figure 4.1, the container runtime engine will pull the 'nginx' image. A container runtime engine is the software that run containers in a host machine [59]. Kubernetes scheduler is a component of the control plane node responsible for selecting and binding a node for a newly created pod [59]. Upon completion, the pod will reach 'pod_starting' state. When the Kubernetes scheduler binds the pod to a worker node, and at least one container has started, the pod reaches the 'Running' phase. A pod in the 'Running' phase can communicate with all other pods on the node or any other node in the Kubernetes cluster. As part of the 'Running' phase the following states are executed: 'image_provided_to_cri', 'volume_mounted', 'host_network_access', 'host_system_access', and 'pod_running'.

A container inside a pod can terminate after completing its task or terminate at runtime. If at least one container terminates in failure at runtime, such as terminated by the system,

then the pod reaches the 'Failed' phase via the 'pod_failed' state. Upon termination, a container can be restarted based on the container restart policy. A pod reaches the 'Succeeded' phase if all its containers terminate after successful execution via the 'pod_succeeded' state. If the 'kubelet' component of the worker node where the pod is running fails to communicate with the Kubernetes API server in the control plane node, the pod reaches 'Unknown' phase. For both succeeded and failed phases, the 'pod_terminate' state is executed.

5.1.5 Counterexample Generation

Model checking is a method to check if a system's finite state model (FSM) fulfils a particular set of specifications [10]. Model checking method explores all possible states of a system and all possible values for the state variables [10]. In the NuXMV, we verify the pod security requirements in our FSM for a pod as a safety property [10]. A safety property dictates that under certain condition bad events will never happen [10]. For model checking, we use counter example guided abstraction refinement (CEGAR) principle [10]. According to the CEGAR principle, we initially use the pod security requirements and our FSM for verification in the NuXMV [16]. If the NuXMV generates erroneous counter-example then we revise our pod security requirements to prevent the erroneous counter-examples. We continue this process for violation of each safety property until we find understandable counter-example. Depending on counter-example output, we set the values of some state variables, transition conditions as constant to prevent erroneous counterexamples. The value of the state variables and transition conditions we set as constant in LTL formula do not change during the execution for verification in NuXMV. Execution for each of the 9 pod security 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.

For instance, in the following NuXMV LTL formula as described in Table 5.2, we use `TC_18` and `TC_19` & `new_pod_creation_request` as constants. Here `TC_18` and `TC_19` are transition conditions and both are set to `False`.

```
LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN)
-> G X(!over_privileged_container & pod_state=pod_running))
```

Table 5.3: Implementation of Each Transition Condition (T)

Transition Condition	**Propositional Logic Formula for Transition**
T_1	(pod_state = request_initiated ∧ (Valid_Kubeconfig ∨ Auth_Bootstrap_Token))
T_2	(pod_state = request_initiated ∧ (¬(Valid_Kubeconfig ∨ Auth_Bootstrap_Token)))
T_3	(pod_state = request_authenticated ∧ (((API_request_verb_GET ∨ API_request_verb_LIST ∨ API_request_verb_DELETE ∨ API_request_verb_UPDATE ∨ API_request_verb_PATCH ∨ API_request_verb_CREATE) ∧ API_resource_pod) ∨ Cluster-Role_cluster_admin))
T_4	(pod_state = request_authenticated ∧ (¬(API_request_verb_GET ∨ API_request_verb_LIST ∨ API_request_verb_DELETE ∨ API_request_verb_UPDATE ∨ API_request_verb_PATCH ∨ API_request_verb_CREATE) ∧ ¬API_resource_pod))
T_{5_0}	pod_state = request_authorized ∧ (((mutating_validating_admission_controller_available)))
T_{5_1}	pod_state = request_admission_controller_validated ∧ (((mutating_validating_admission_controller_available ∧ mutating_validating_admission_validation)))
T_{5_2}	pod_state = request_authorized ∧ ((¬mutating_validating_admission_controller_available))
T_6	pod_state = request_authorized ∧ ((authentication_authorization_successful ∧ default_admission_controller_pass ∧ mutating_validating_admission_controller_available) ∧ ¬(mutating_validating_admission_validation))
T_7	pod_state = request_pod_creation_initiated ∧ (request_accepted_k8s_api_server)
T_{7_2}	pod_state = desired_pod_state_stored_in_etcd ∧ (desired_state_written_at_etcd_by_api_server ∧ pod_object_created_by_resource_controller)
T_{7_3}	pod_state = pod_schedule_score_phase ∧ (pod_creation_pending_state ∧ pod_schedule_initiated)
T_8	pod_state = pod_schedule_bind_phase ∧ (scheduler_binds_node_to_pod ∧ podspec_poll_by_kubelet)
T_9	pod_state = kubelet_receives_podspec ∧((¬image_present ∧ image_pull_policy=NEVER) ∨ ¬imagepull_registry_valid_credentials)
T_{10}	pod_state = kubelet_receives_podspec ∧ (image_present ∧ (image_pull_policy=NEVER ∨ image_pull_policy=IFNOTPRESENT))
T_{11}	pod_state = kubelet_receives_podspec ∧ ((image_pull_policy=ALWAYS ∨ image_pull_policy =IFNOTPRESENT) ∧ imagepull_registry_valid_credentials)
T_{11_2}	pod_state = image_pulled_from_registry ∧ ((image_pull_policy=ALWAYS ∨ image_pull_policy =IFNOTPRESENT) ∧ imagepull_registry_valid_credentials)
T_{12}	pod_state = image_provided_to_cri ∧ (persistent_volume ∧ persistent_volume_claim ∧ secret_configmap_volume_mount)
T_{13}	pod_state = volume_mounted ∧ (container_initializing_or_ready ∧ pod_has_network)
T_{14}	pod_state = pod_starting ∧ container_poststart_webhook
T_{15}	pod_state = pod_running ∧ (missing_dependency_for_pod ∨ pod_runtime_error)
T_{16}	pod_state = error_crash_loop_backoff ∧ (pod_runtime_error ∧ ¬(livenessprobe_enabled ∨ startupprobe_enabled))
T_{17}	pod_state = pod_failed ∧ (¬sigkill_zero_exit ∧ ((livenessprobe_enabled ∨ readinessprobe_enabled)))
T_{18}	pod_state = pod_running ∧(((pod_disruption_budget ∧ (pod_disruption_allowed ∧ pod_disruption_budget_max_available_min_unavailable_condition_satisfie ∧ container_prestop_webhook ∧ pod_termination_grace_period_default_30s ∧ ¬pod_termination_force_NO_grace_period ∧ API_request_verb_DELETE ∧ API_resource_pod))
T_{19}	pod_state = pod_running ∧ (((¬pod_cpu_memory_limit_enabled ∧ ¬pod_cpu_memory_request_limit_enabled) ∧ node_resource_quota_enabled) ∨ (¬readinessprobe_enabled ∧ pod_eviction_pod_preemtion ∧ node_resource_quota_enabled) ∨ (¬limit_node_PID ∨ ¬limit_pod_PID ∨ ¬pod_eviction_policy) ∨ (namespace_resource_quota_enabled ∧(¬pod_cpu_memory_request_limit_enabled ∧ ¬pod_cpu_memory_request_limit_enabled)))
T_{20}	pod_state = pod_running ∧ (pod_CNI_enabled ∧ clusterIP_NodePort_exposed)
T_{22}	pod_state = pod_running ∧ (CAP_NET_RAW ∨ CAP_SYS_ADMIN ∨ network_request_other_workload ∨security_context_privileged ∨ ¬security_context_run_as_user ∨ lsm_SECCOMP_enabled ∨ lsm_APPArmor_enabled ∨ lsm_SELinux_enabled ∨ hostprocess_enabled ∨ hostNetwork_enabled ∨ hostPID_enabled ∨ hostIPC_enabled ∨ host_path_enabled ∨ ¬container_DropCapabilites ∨ ¬running_as_NON_ROOT ∨ pod_admission_BASELINE ∨ ¬pod_admission_RESTRICTED ∨ pod_admission_PRIVILEGED ∨ ¬pod_admission_ENFORCE ∨ pod_security_exemption_user ∨ pod_security_exemption_namespace ∨ pod_security_exemption_workload_pod ∨ ¬network_policy_between_pods ∨ docker_socket_enabled ∨ default_namespace ∨ service_account_privileged ∨ service_account_automount_token)
T_{23}	pod_state = service_exposed ∧ (¬CAP_NET_RAW ∧ ¬CAP_SYS_ADMIN ∧ ¬security_context_privileged ∧ security_context_run_as_user ∧ lsm_SECCOMP_enabled ∧ lsm_APPArmor_enabled ∧ lsm_SELinux_enabled ∧ ¬hostprocess_enabled ∧ ¬hostNetwork_enabled ∧ ¬hostPID_enabled ∧ ¬hostIPC_enabled ∧ ¬host_path_enabled ∧ container_DropCapabilites ∧ running_as_NON_ROOT ∧ ¬pod_admission_BASELINE ∧ pod_admission_RESTRICTED ∧ ¬pod_admission_PRIVILEGED ∧ pod_admission_ENFORCE ∧ pod_security_exemption_user ∧ pod_security_exemption_namespace ∧ pod_security_exemption_workload_pod ∧ network_policy_between_pods ∧ ¬default_namespace ∧ ¬docker_socket_enabled ∧ ¬service_account_privileged ∧ ¬service_account_automount_token)
T_{24}	pod_state = service_exposed ∧ (CAP_NET_RAW ∨ CAP_SYS_ADMIN ∨ network_request_other_workload ∨ security_context_privileged ∨ security_context_run_as_user ∨ lsm_SECCOMP_enabled ∨ lsm_APPArmor_enabled ∨ lsm_SELinux_enabled ∨ hostprocess_enabled ∨ hostNetwork_enabled ∨ hostPID_enabled ∨ hostIPC_enabled ∨ host_path_enabled ∨ ¬container_DropCapabilites ∨ ¬running_as_NON_ROOT ∨ pod_admission_BASELINE ∨ ¬pod_admission_RESTRICTED ∨ pod_admission_PRIVILEGED ∨ ¬pod_admission_ENFORCE ∨ pod_security_exemption_user ∨ pod_security_exemption_namespace ∨ pod_security_exemption_workload_pod ∨ ¬network_policy_between_pods ∨ docker_socket_enabled ∨ default_namespace ∨ service_account_privileged ∨ service_account_automount_token)
T_{25}	pod_state = pod_running ∧ new_pod_creation_request ∧ (container_breakout ∨ malicious_container ∨ pod_admission_PRIVILEGED ∧ (¬admission_namespace_lifecycle ∧ admission_control_bypass))
T_{26}	pod_state = host_system_access ∧ (container_secret_exfiltration)
T_{28}	pod_state = image_pulled_from_registry ∧ unscanned_container_image
T_{29}	pod_state = container_registry_poisoned ∧ (misconfigured_image ∨ unscanned_container_image)
T_{30}	pod_state = pod_running ∧ network_misconfiguration
T_{31}	pod_state = pod_terminated ∧ sigkill_zero_exit
T_{32}	pod_state = pod_terminated ∧ ¬sigkill_zero_exit
T_{33}	pod_state = pod_unrestricted_communication ∧ (network_misconfiguration ∨ remote_service_connected_from_cluster)
T_{35}	pod_state = host_system_access ∧ (RCE_vulnerability ∨ misconfigured_image)
T_{36}	pod_state = host_system_access ∧ container_breakout
T_{38}	pod_state = host_system_access ∧ service_account_privileged
T_{40}	pod_state = host_network_access ∧ (malicious_container ∨ over_privileged_container ∨ network_request_other_workload)
T_{41}	pod_state = host_system_access ∧ network_misconfiguration
T_{42}	pod_state = remote_service_connected ∧ remote_service_connected_from_cluster ∧ malicious_container
T_{43}	pod_state = remote_service_connected ∧ service_account_privileged
T_{44}	pod_state = kube_api_server_is_updated_by_kubelet ∧ container_breakout
T_{45}	pod_state = kubelet_receives_podspec ∧ container_state = WAITING ∧ pod_schedule_completed

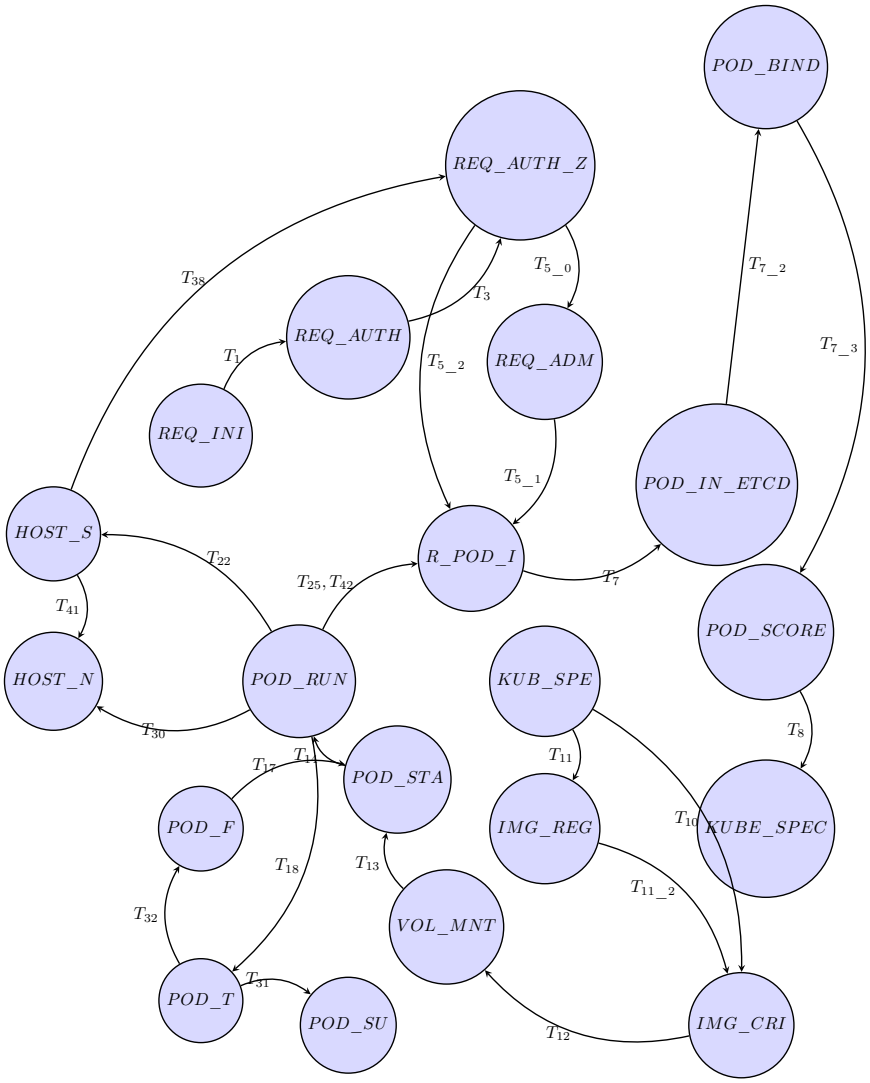

Figure 5.7: A finite state machine representing the states of a pod.

Figure 5.7 summarizes the states that a pod encounters. For example, 'REQ_INI' represents the 'request_initiated' state, which transitions to the next state called 'request_authentication' ('REQ_AUTH') if the transition condition T_1 is true. Here, T_1 corresponds to the condition of (Valid_Kubeconfig $\vee$ Auth_Bootstrap_Token)), which means either the pod has a valid configuration or authentication bootstrap token. A description of the transition conditions is available in Table 5.3. Certain configurations are only applicable during certain states of the pod. For example, the configuration `hostNetwork` is applicable when the state of the pod is 'host_network_access'. The context of states for pod configurations requires encoding and analyzing these states in forms of a finite state machine.

5.1.6 Attack Validation

Execution Environment

We validate the attacks for misconfigurations using `kubeadm` installation of Kubernetes cluster. The `kubeadm` installation provides multi-node vanilla Kubernetes cluster. We use vagrant to set up a three node Kubernetes cluster with one control plane and two worker nodes using virtual box VMs for nodes in our local workstation. The configuration of the local machine is Mac OS Intel Core i7, 8 core CPU with 16GB of memory. We assign the control plane 4GB of memory and 2 virtual central processing unit (vCPU) and the worker nodes have 3GB of memory and 2 virtual central processing unit each. The virtual central processing unit(vCPU) represents the central processing unit (CPU) of a virtual machine. We deploy the microservice-demo [36] application to demonstrate our attack which has 11 microservices. We use this application as Google uses this application to demonstrate the use of Kubernetes [36]. We edit the Kubernetes manifests of `cartservice` and `checkoutservice` of the microservice-demo by providing root privilege, privilege escalation, host namespaces and container privileges so that we can exploit the misconfigurations to conduct an attack. We assume that the attacker has an "initial access" into the Kubernetes cluster [41]. In

each of our attack validation set up, the attacker has an access to a valid Kubeconfig file and uses the kubectl CLI interface to communicate with the Kubernetes API server.

Description of Attacks

We verify the pod security requirements using NuXMV model checker with our FSM model for a pod. We validate attacks against each of the counter examples generated by NuXMV. We demonstrate 6 attacks for 9 pod security requirements. We describe the attacks as follows:

Attack #1 **Access cluster secrets:** In this attack, an attacker exploits the unnecessary privilege of a container running inside a pod to access cluster secrets.

Detection and attack description:

We verify the pod security requirement *"Any container running inside a pod with unnecessary privilege will not be in an unsafe state at runtime"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can generate a new pod creation request from a running pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker gets access to a privileged pod that starts with missing admission controller misconfiguration [42] [79]. The pod has the privileges: `hostNetwork: true , hostIPC: true, hostPID: true, allowPrivilegeEscalation: true`, active hostPath , Capability abuse, privileged `securityContext`, `automountServiceAccountToken: true, readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False`. The pod is associated with the `serviceAccount: default`. The privileges of the pod allows the attacker to escalate pod isolation boundary in the host [100].

Step-2: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true`, privileged `securityContext`, active hostPath and `automountServiceAccountToken: true` to get the service account token of `serviceAcco unt: default` [100].

Step-3: The attacker has access to the service account token of `serviceAccount: default` as privileged `ServiceAccount` that has privileged role. The attacker uses the privileged `ServiceAccount` which has over-privileged permission as a `cluster-admin` role.

Step-4: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-5 The attacker leverages the misconfigurations and creates a new malicious privileged pod with the service account token of privileged `ServiceAccount`. The malicious pod can run as cryptominer to disrupt other running pods. The attacker can also read Kubernetes cluster secrets, modify/delete running pods in the Kubernetes cluster from the malicious pod.

Implication: An attacker can run malicious applications such as a cryptominer and consume excessive resource to disrupt running pods, get secrets from the Kubernetes API server or modify/delete any running pods causing service disruption to the victim organization. Hence, with this "workload privilege escalation attack", an attacker can violate the confidentiality, integrity and availability of the Kubernetes cluster.

Mitigation: We mitigate this attack by eliminating our identified privileged pod misconfigurations: `hostNetwork:true , hostIPC:true, hostPID:true`, privileged `securityContext`, `allowPrivilegeEscalation:true, readOnlyRootFileSystem:False, runAsNonRoot:False, runAsUser: False`, active hostPath, Capability abuse, so that the attacker can not access the pod to get the root privilege with write permission and get unnecessary access to the host machine. We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if available: privileged role, privileged `ServiceAccount`, `automountServiceAccountToken: true, serviceAccount: default`.

Attack #2 **Dashboard maneuver:** In this attack, an attacker can exploit over-privileged RBAC permission in Kubernetes cluster.

Detection and attack description:

We verify the pod security requirement *"An over-privileged RBAC permission in Kubernetes cluster will not lead a pod to an unsafe state."*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can create a new pod creation request from the Kubernetes dashboard using a default service account with over-privileged RBAC permission.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker has access to the kubernetes dashboard with the service account token of `serviceAccount: default` as privileged default `ServiceAccount` that has `cluster-admin` role to access the Kubernetes dashboard.

Step-2: The attacker can access the Kubernetes dashboard with the `cluster-admin` and privileged default `ServiceAccount`. The attacker leverages the misconfigurations missing admission controller to deploy malicious pod. The attacker can also modify/delete nodes, expose secrets and sensitive applications with the `cluster-admin` privilege.

Implication: An attacker with over-privileged RBAC permission, such as `cluster-admin` privilege can provide an attacker a complete control over the Kubernetes cluster of an organization and violate confidentiality, integrity, availability.

Mitigation: To mitigate the attack, We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if available: privileged role, privileged `ServiceAccount`, privileged Default `ServiceAccount, serviceAccount: default, cluster-admin`.

Attack #3 **Databse tampering:** In this attack, an attacker leverages misconfigurations related to network segmentation and exploits unrestricted sensitive applications.

Detection and attack description:

We verify pod security requirement *"Misconfigurations in network segmentation will not lead a pod to an unsafe state"*. We observe a counterexample for this pod security

requirement and detect an attack where an attacker can access a sensitive pod, such as a database, from a malicious pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller misconfiguration [42] [79]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine. The pod has the misconfigurations: `hostNetwork: true, readOnlyRootFileSystem: False, namespace: default, runAsNonRoot:False, runAsUser: False`

Step-2: The attacker uses the ncat [78] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [100].

Step-3: The attacker again uses the ncat tool using privilege `hostNetwork: true`, missing network policy in `namespace: default` to establish a connection with the Redis database in port 6379.

Step-4: The attacker can update the data in the Redis database as Redis does not require any authentication by default [96], [109]. The attacker can modify or delete sensitive information from the redis database. As a result the pods may get tampered data when required. This attack violates confidentiality, integrity and availability of Kubernetes cluster.

Implications: Any attacker can leverage the network misconfigurations to connect with sensitive applications such as database and get sensitive data or modify existing data. The attacker can also cause a network-related denial of service attack to business-critical applications.

Mitigation: To mitigate this attack, we apply network policy for the sensitive applications such as redis database and avoid using default namespace. We apply non root user with read only permission and eliminate access to host network privilege from the pods as well. We eliminate the following misconfigurations if available: missing network policy, `namespace: default, hostNetwork: True, readOnlyRootFileSystem: False, runAsUser: False`.

Attack #4 **Denial of service (DOS):** In this attack, an attacker leverages the lack of centralized policy and missing resource limit specification for pods in Kubernetes cluster. In the attack, attacker deploys the pod without specifying resource limits to create denial of service attack.

Detection and attack description:

We verify pod security requirement *"Lack of centralized policy such as missing admission controller will not lead a pod to an unsafe state."* We observe a counterexample for this pod security requirement and detect an attack where an attacker can cause a denial of service attack from a running pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller and missing resource limit misconfigurations [42] [79]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine [100]. The pod has the misconfigurations: `runAsNonRoot:False, runAsUser: False`

Step-2: The attacker uses the ncat [78] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [100]

Step-3: The attacker runs a malicious program inside the running container of the pod with `readOnlyRootFileSystem: False`, absent resource limit misconfigurations. The pod consumes the entire CPU and memory limit of the node and disrupts the availability of running pods in the Kubernetes cluster.

Implications: Lack of centralized policy can allow outdated, vulnerable images, dependencies to run as containers. As a result, any attacker can leverage the underlying image vulnerability to cause critical service disruption, such as a resource-related denial of service attack in the Kubernetes cluster. This attack violates the availability of the Kubernetes cluster.

Mitigation: To mitigate the attack, We eliminate missing resource limit, disable root user and enable read only file system in pod configuration. We also recommend using "ValidatingAdmissionPolicy" to eliminate missing admission controller so that any pod without resource limit never gets deployed at the Kubernetes cluster. We eliminate the following misconfigurations if available: missing admission controller, absent resource limit, `runAsUser:False, readOnlyFileSystem:True`.

Attack #5 **Etcd takeover:** In this attack, an attacker can leverage secret management-related misconfigurations to steal secrets from Kubernetes cluster. By default, the etcd database in the control plane is not encrypted, and access to an unencrypted etcd database can leak sensitive cluster information.

Detection and attack description:

We verify pod security requirement, *"If any secrets are accessible and stored without encryption in Kubernetes cluster, it can lead a pod to an unsafe state"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can access a pod with a privileged service account, deploy a pod in the control plane node, and extract the unencrypted etcd database contents.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker gets access to a privileged pod that starts with missing admission controller misconfiguration [42] [79]. The pod has the privileges: `hostNetwork:true , hostIPC:true, hostPID: true, allowPrivilegeEscalation: true`, privileged `securityContext, automountServiceAccountToken: true, readOnlyRootFileSystem: False`, active hostPath, Capability abuse, `runAsNonRoot:False, runAsUser: False`. The pod is associated with the `serviceAccount: default`. The privileges of the pod allows the attacker to escalate pod isolation boundary in the host [100].

Step-2: The attacker uses the ncat [78] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [100].

Step-3: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true`, active hostPath, privileged `securityContext, automountServiceAccountToken: true` to get the service account token of `serviceAccount: default` [100].

Step-4: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-4: The attacker creates a privileged pod in the control plane node with misconfigurations missing admission controller, `nodeName:master`. The privileged pod has the permissions: `hostNetwork: true , hostIPC: true, hostPID: true, allowPrivilegeEscalation: true, runAsNonRoot: false`, privileged `securityContext, readOnlyRootFileSystem: False`, Capability abuse and the pod runs in the control-plane node.

Step-5: The attacker installs etcd client and can access the key and certificate for etcd using `hostNetwork: true , hostIPC: true, hostPID: true, allowPrivilegeEscalation: true, runAsNonRoot: false` privileged `securityContext, readOnlyRootFileSystem: False`, and Capability abuse misconfigurations.

Step-6: The attacker can use the key and certificate to access the unencrypted etcd database that contains all cluster secrets and information. This action from the attacker violates the confidentiality of the Kubernetes cluster.

Implication: Any attacker who has access to Kubernetes API server can extract unencrypted cluster secrets such as database credentials and cluster information and violate confidentiality of the Kubernetes cluster.

Mitigation: To mitigate the attack, we avoid the configuration `nodeName:master` so that the pods are not deployed in the control-plane node. We also eliminate the following misconfiguration if present in the pod configurations: `hostNetwork:true , hostIPC:true,`

`hostPID:true, privileged:true, allowPrivilegeEscalation:true`, active hostPath, Capability abuse, `readOnlyRootFileSystem:False, runAsNonRoot:False, runAsUser: False` privileged role, privileged `ServiceAccount` and missing admission controller.

Attack #6 **Pod disruption:** In this attack, an attacker exploits any vulnerabilities in the library, dependencies, container images of the pod deployment manifest to disrupt the availability running pod.

Detection and attack description:

We verify the pod security requirement *"Any unscanned or misconfigured container image will not lead a pod to an unsafe state"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can remotely connect to a running pod and create a new pod creation request.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller misconfiguration [42] [79]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine. The pod has the misconfigurations: `hostNetwork: true, hostIPC: true, hostPID: true, allowPrivilegeEscalation: true, automountServiceAccountToken: true`, privileged `securityContext`, Capability abuse, `readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False`.

Step-2: The attacker uses the ncat [78] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [100].

Step-3: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true, automountServiceAccountToken: true`, privileged `securityContext`, and active hostPath to get the service account token of `serviceAccount: default` [100].

Step-4: The attacker has access to the service account token of `serviceAccount: default` as privileged `ServiceAccount` that has privileged role. The attacker uses the privileged `ServiceAccount` which has over-privileged permission as a `cluster-admin` role.

Step-5: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-6 The attacker leverages the misconfiguration missing admission controller and creates a new malicious privileged pod with the service account token of privileged `ServiceAccount`. The malicious pod can run as cryptominer to disrupt other running pods. The attacker can also read Kubernetes cluster secrets, modify/delete running pods in the Kubernetes cluster from the malicious pod.

Implication: An attacker can exploit the remote code execution vulnerability to get access to the Kubernetes cluster and leverage the misconfigurations in pod manifests to create malicious applications such as cryptominers. The attacker can also leak secrets, capture the network communication, and modify/delete pods without getting noticed by the cluster administrator. Hence, this "remote code execution attack" can violate confidentiality, integrity and availability of the Kubernetes cluster.

Mitigation: We mitigate this attack by applying "ValidatingAdmissionPolicy" [59] that checks container image origin and eliminates missing admission controller misconfiguration so that any malicious container with remote code execution vulnerability never gets into Kubernetes cluster. After that, we eliminate our identified privileged pod misconfigurations: `hostNetwork:true , hostIPC:true, hostPID:true`, privileged `securityContext`, active hostPath, capability abuse, `allowPrivilegeEscalation:true, readOnlyRootFileSystem:False, runAsNonRoot:False, runAsUser: False` so that the attacker can not access the pod to get the root privilege with write permission and get unnecessary access to the host machine. We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if

available: privileged role, privileged `ServiceAccount, automountServiceAccountToken: true, serviceAccount: default`.

In Table 5.4, we list our attacks, configuration sequences and dependencies for insecure pod provisioning.

Table 5.4: Misconfigurations that invoke insecure provisioning

Attack Name	Configuration Sequence	Dependencies
Access cluster secrets	missing admission controller --> (hostIPC:True, hostPID:True, hostNetwork:True, active hostPath, allowPrivilegeEscalation:True, Capability abuse, privileged securityContext) --> (serviceAccount: default,runAsNonRoot:False, runAsUser: False, readOnlyRootFileSystem: False, automountServiceAccountToken: True, privileged role) --> privileged ServiceAccount, missing admission controller	kubectl exec checkoutservice /bin/sh serviceaccount -> ca.crt namespace token wget kubectl
Dashboard maneuver	serviceAccount: default, automountServiceAccountToken: True, runAsNonRoot:False, runAsUser: False, --> privileged default ServiceAccount, cluster-admin, missing admission controller	kubernetes dashboard access serviceaccount -> ca.crt namespace token dashboard service account
Denial of service(DOS)	missing admission controller, absent resource limit --> readOnlyRootFileSystem: False, runAsNonRoot: False	ncat -vlp <PORT> revshell-pod.yaml DoS-Daemonset.yaml
Database tampering	missing network policy, missing admission controller --> (hostNetwork:True, runAsNonRoot:False, runAsUser:False) --> namespace: default	revshell.yaml ncat -vlp <PORT> ncat -vn redis:6379
Etcd takeover	missing admission controller -->(hostIPC: True, hostPID:True, hostNetwork:True, allowPrivilegeEscalation: True, Capability abuse, privileged securityContext) --> readOnlyRootFileSystem: False, automountServiceAccountToken: False, runAsNonRoot:False, runAsUser: False, --> (serviceAccount: default, privileged ServiceAccount)	revshell.yaml ncat -vlp <PORT> apt-get curl kubectl serviceaccount -> ca.crt namespace token nodeselector-controlplane hack-control plane.yaml wget etcd /etc/kubernetes/pki/etcd/ca.crt, healthcheck-client.crt etcdctl IP:2379 ca.crt healthcheck-client.crt
Pod disruption	missing admission controller --> (hostIPC: True, hostPID:True, hostNetwork:True, allowPrivilegeEscalation: True, Capability abuse, privileged securityContext) --> readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False, serviceAccount: default, automountServiceAccountToken: True, privileged role --> privileged ServiceAccount, missing admission controller	ncat -vlp <PORT> revshell-pod.yaml serviceaccount -> ca.crt namespace token apt-get curl kubectl

5.2 Methodology for RQ 5.2

Prior survey among practitioners demonstrated the prevalence of the misconfigurations in Kubernetes [93]. Rahman et al. identified misconfigurations in Kubernetes manifests in open source software(OSS) repositories [88]. We extend the open source tool SLIKUBE [88]

and build the tool (SLIKUBE+) to detect the presence of additional 13 security misconfigurations in Kubernetes OSS repositories. As an input the user will provide the directory path with Kubernetes manifests and SLIKUBE+ will output the count of misconfigurations for each of the Kubernetes manifests in the directory. In this section, we describe our methodology to construct SLIKUBE and extended the implementation of SLIKUBE to construct SLIKUBE+.

5.2.1 SLIKUBE

We used the qualitative analysis technique - open coding [99] to derive security misconfiguration categories for the tool SLIKUBE. Open coding is well-suited to identify insights in an under-explored domain, such as Kubernetes security misconfigurations. Furthermore, open coding provides a systematic way to surface similarities across textual artifacts, and group such similarities into categories [99].

Identification Misconfiguration Categories for SLIKUBE

As part of the open coding process, *first*, the rater identifies configurations in a Kubernetes manifest. *Second*, the rater inspects the values for each identified configuration to determine if the configuration is in fact a security misconfiguration. While determining misconfigurations, the rater uses the following definition of security misconfiguration provided by the U.S. National Institute of Standards and Technology (NIST) [77] "*A setting within a computer program that violates a configuration policy or that permits unintended behavior that impacts the security posture of a system*". Both raters, who are well-versed on Kubernetes (having used them in practice) initially came up with a list of security misconfigurations that can potentially cause unintended behaviors based on their experience.

Third, the rater derives categories based on similarities between the identified instances of security misconfigurations. For each identified security misconfiguration category, the

rater further checks if the category violates any of the Kubernetes-related security best practices as documented in Section 3.1. We use our study because it (i) systematically synthesizes security-related best practices from multiple Internet artifacts, and (ii) is peer-reviewed and leveraged a grey literature review with 101 Internet artifacts including multiple artifacts that came out of Snyk [103], where practitioners have discussed the security best practices for Kubernetes. Other artifact sources that we leverage include artifacts authored by practitioners from Google Cloud, Cloud Native Computing Foundation (CNCF), VMWare, Tech Republic, DZone, SonaType, IBM, and Microsoft.

Upon completing the aforementioned three steps, we derive a list of security misconfiguration categories. In this manner, our identified security misconfigurations convey the message that if identified security misconfigurations are not mitigated, they can permit unintended behaviors.

Rater Verification for Misconfiguration Categories for SLIKUBE

The first and second authors act as raters, and conduct the open coding process. The first author and second author respectively, has experience in working with Kubernetes for one and two years. Both rater individually manually inspects 1,796 Kubernetes manifests provided by Brinto et al. [13]. Brinto et al. [13]'s dataset includes 1,796 Kubernetes manifests that are modified in 5,193 commits, and collected from 38 OSS repositories. Of the 1,796 Kubernetes manifests, 90% and 10% are respectively, `Kind` and `Helm` manifests. For each Kubernetes manifest, both raters individually apply the aforementioned open coding process.

Upon completion of the open coding process, the first and second authors respectively, identify 8 and 6 categories of security misconfigurations. We compute Krippendorff's α [55] to quantify agreement, similar to prior work in software engineering [8, 91, 29]. The Krippendorff's α is 0.45, indicating *'unacceptable'* agreement [55]. Both raters discussed their disagreements and observed that root cause of their disagreements occur due to the second

author missing five categories, identified by the other author. These categories are: activation of `hostIPC`, activation of `hostNetwork`, activation of `hostPID`, capability misuse, and Docker socket mounting. The second rater missed categories because of being unaware of these configurations. Upon discussion, both raters conduct the inspection process again. After completing the inspection process, we calculate Krippendorff's α to be 1.0, indicating *'perfect'* agreement [55]. We use Krippendorff's α instead of Cohen's κ, because Krippendorff's α: (i) emphasizes disagreement leading to more reliability on the achieved agreement rate, and (ii) handles multiple categories [55]. Furthermore, qualitative analysis experts have advocated for the use of Krippendorff's α over Cohen's κ [56, 66]

Altogether, we identify 8 categories of security misconfigurations in Kubernetes manifests. An example of each category with a mapping to the violated security practice is presented in Table 5.5. All the examples presented in Table 5.5 are obtained from Kind manifests. 'Count' corresponds to the count for the Brinto et al. [13] dataset for each category.

Methodology to Construct SLIKUBE

Step-1: Parsing: SLIKUBE parses Kubernetes manifests into key-value pairs. For each key, a value can be a nested dictionary, or a list, or a single value. In the case of nested dictionaries, SLIKUBE preserves the hierarchy of the extracted keys for Kubernetes manifest.
Step-2: Rule Matching: From the parsed content of Kubernetes manifests, SLIKUBE applies rule matching to identify security misconfigurations. The rules needed to identify categories are listed in Table 5.6. The rules are derived by abstracting code snippets for each misconfiguration category. The rules presented in Table 5.6 leverage pattern matching similar to prior research [85, 87]. The string patterns used by each rule in Table 5.6 is provided in Table 5.7.

Table 5.5: Examples of Security Misconfiguration Categories of SLIKUBE

Category (Count)	Violated Practice	Example Code Snippet
Active `hostIPC`	Implementing Pod-specific Policies [101]	`spec:` `  hostIPC: true`
Active `hostNetwork`	Implementing Pod-specific Policies [101]	`spec:` `  hostNetwork: true`
Active `hostPID`	Implementing Pod-specific Policies [101]	`spec:` `  hostPID: true`

Capability Misuse	Implementing Pod-specific Policies [101]	`capabilities:`
		`  add:`
		`  - CAP_SYS_ADMIN`
		`  - CAP_SYS_MODULE`

Escalated Privileges for Child Container Processes	Implementing Pod-specific Policies [101]	`allowPrivilegeEscalation: true`
Missing SSL/TLS for HTTP	Enable SSL/TLS Support [101]	`value: http://elastisearch-logging:9200`

Missing resource limit	Limit CPU and Memory Quota [101]	`spec:`
		`  containers:`
		`  - name: employee`
		`  image: piomin/employee-service`

Privileged `securityContext`	Implementing Pod-specific Policies [101]	`securityContext:` `  privileged: true`

Rule Derivation Process: We identify the commonalities in patterns capable of expressing security misconfigurations, and abstract such commonalities as rules to detect misconfigurations. For instance, `privileged` keyword is used to specify the coding pattern. SLIKUBE can parse both coding patterns as key value pairs, where `privileged` is the key and `true` is the value. In both coding patterns we notice commonality in the key value pairs, which can be abstracted to a rule $isKey(x) \wedge isSecuirtyContext(x) \wedge isPrivileged(x) \wedge isEnabled(x.value)$. We repeat the same abstraction process for other misconfiguration categories.

5.2.2 SLIKUBE+

In Section 5.4.1, we identify 13 additional security misconfigurations along with 8 security misconfigurations that can be identified with SLIKUBE described in Table 5.6. To identify the 13 security misconfigurations we extend SLIKUBE to derive 13 additional rules for parsing and rule matching in Kubernetes manifest by following the technique described in

Table 5.6: Rules Used by SLIKUBE

Category	Rule
Activation of `hostIPC`	$(isKey(x) \wedge isHostIPC(x) \wedge isEnabled(x.value)$
Activation of `hostPID`	$(isKey(x) \wedge isHostPID(x) \wedge isEnabled(x.value)$
Activation of `hostNetwork`	$(isKey(x) \wedge isHostNetwork(x) \wedge (isEnabled(x.value)$
Capability Misuse	$(isKey(x) \wedge isContainer(x) \wedge hasCapability(x) \wedge (isCAPSYSADMIN(x.value) \vee isCAPSYSMODULE(x.value))$
Escalated Privileges for Child Container Processes	$isKey(x) \wedge isPrivEscalation(x) \wedge isEnabled(x.value)$
Missing SLL/TLS for HTTP	$isKey(x)(\wedge isProtocol(x.name) \vee isHTTP(x.value))$
Missing Resource Limit	$(isKey(x) \wedge (isSpec(x) \vee isContainer(x)) \wedge \neg(isLimitResources \wedge (isLimitMemory \wedge isLimitRequests))))$
Privileged `securityContext`	$isKey(x) \wedge isSecuirtyContext(x) \wedge isPrivileged(x) \wedge isEnabled(x.value)$

Section 5.2.1. We describe the rules to extend SLIKUBE in Table 5.8. We also describe the string pattern for the rules in Table 5.9. In total, we identify 21 types of misconfigurations that are related pod security requirements described in Section 5.4.

Table 5.7: String Patterns Used for Rules in SLIKUBE

Function	**String Pattern**
$hasCapability()$	'capabilities'
$isCAPSYSADMIN()$	'CAP_SYS_ADMIN'
$isCAPSYSMODULE()$	'CAP_SYS_MODULE'
$isContainer()$	'container'
$isDockerSocket()$	'/var/run/docker.sock'
$isEnabled()$	'true'
$isHostIPC()$	'hostIPC'
$isHostNetwork()$	'hostNetwork'
$isHostPID()$	'hostPID'
$isHTTP()$	'http:'
$isLimitMemory()$	'limits'
$isLimitRequests()$	'requests'
$isLimitResources()$	'resources'
$isPath()$	'path'
$isPassword()$	'password'
$isPrivEscalat()$	'allowPrivilegeEscalation'
$isProtocol()$	'protocol'
$isPriviledged()$	'privileged'
$isSecurityContext()$	'securityContext'
$isSpec()$	'spec'
$isToken()$	'key'
$isUser()$	'user'

Table 5.8: Additional Rules for SLIKUBE+ to Extend SLIKUBE

Misconfiguration Name	**Rule**
Missing Admission Controller	$isKind(x) \wedge (isApiVersion(x) \wedge (\neg isAdmissionConfiguration(x.value) \vee \neg isAdmissionReview(x.value) \vee \neg isValidationAdmission(x.value) \vee \neg isMutatingAdmission) \vee \neg isApiVersionAdmission(x.value))$
Missing Network Policy	$isKind(x) \wedge (isApiVersion(x) \wedge (\neg isNetworkPolicy(x.value) \vee \neg isIngress(x.value) \vee \neg isEgress(x.value))$
Privileged Role	$(isKind(x) \wedge (isRole(x.value) \vee isClusterRole(x.value))) \wedge ((isVerb(x) \wedge hasRiskyVerbPrivilege(x)) \wedge ((isResource(x) \wedge hasRiskyResourcePrivilege)) \wedge isName(x) \wedge$
Privileged `ServiceAccount`	$(isKind(x) \wedge (isRole(x.value) \vee isClusterRole(x.value))) \wedge ((isVerb(x) \wedge hasRiskyVerbPrivilege(x)) \wedge ((isResource(x) \wedge hasRiskyResourcePrivilege(x)) \wedge isName(x) \wedge (isKind(x) \wedge isClusterRoleBinding(x) \vee isRoleBinding(x) \wedge (isRole(x) \wedge isKindClusterRole(x) \wedge isKindServiceAccount(x)))$
Privileged `RoleBinding`	$(isKind(x) \wedge (isClusterRoleBinding(x) \vee isRoleBinding(x) \wedge (isClusterAdmin())))$
Inactive read-only root filesystem	$isKind(x) \wedge ((isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x)) \wedge isVolumeMount(x) \wedge (isReadOnlyRootFileSystem(x) \vee isReadOnlyFileSystem(x) \wedge (isEnabled(x) \vee (\neg isReadOnlyRootFileSystem(x) \wedge \neg isReadOnlyFileSystem(x))$
Auto mounted token	$isKind(x) \wedge (isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x) \vee isKindService(x)) \wedge ((isAutomountServiceToken(x) \wedge (isEnabled(x))) \vee \neg(isAutomountServiceToken(x)))$
Default `ServiceAccount`	$isKind(x) \wedge (isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x) \vee isKindService(x)) \wedge \neg isKeyServiceAccountName(x)$
Default `Namespace`	$isKind(x) \wedge (isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x) \vee isKindService(x)) \wedge \neg isKeyNamespace(x)$
Privileged default `ServiceAccount`	$isKind(x) \wedge (isRole(x.value) \vee isClusterRole(x.value)) \wedge ((isVerb(x) \wedge hasRiskyVerbPrivilege(x)) \wedge (isResource(x) \wedge hasRiskyResourcePrivilege(x)) \wedge (isName(x) \wedge (isDefault(x)))$
Active `hostPath`	$isKind(x) \wedge ((isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x)) \wedge isHostPath(x) \wedge (isReadOnlyRootFileSystem(x) \vee isReadOnlyFileSystem(x) \wedge (isEnabled(x) \vee (\neg isReadOnlyRootFileSystem(x) \wedge \neg isReadOnlyFileSystem(x))$
Inactive `runAsNonRoot`	$isKind(x) \wedge ((isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x)) \wedge (isRunAsNonRoot(x) \wedge (isEnabled(x) \vee (\neg isRunAsNonRoot(x))$
Inactive `runAsUser`	$isKind(x) \wedge ((isKindPod(x) \vee isKindDeployment(x) \vee isKindDaemonSet(x) \vee isReplicaSet(x)) \wedge (\neg(isRunAsUser(x)) \vee (\neg isRunAsGroup(x)))$

Table 5.9: Additional String Patterns Used for Functions in SLIKUBE+ Rules

Function	**String Pattern**
isImagePullSecrets()	'imagePullSecrets'
isImagePullPolicy()	'imagePullPolicy'
isAlways()	'Always'
isNodeName()	'nodeName'
hasNameMaster()	'master', 'control-plane', 'controlplane'
isKind()	'kind'
isApiVersion()	'apiVersion'
isApiVersionAdmission()	'admissionregistration.k8s.io/v1'
isAdmissionConfiguration()	'AdmissionConfiguration'
isAdmissionReview()	'AdmissionReview'
isValidationAdmission()	'ValidatingWebhookConfiguration'
isMutatingAdmission()	'MutatingWebhookConfiguration'
isRole()	'Role'
isClusterRole()	'ClusterRole'
isVerb()	'verbs'
isResource()	'resources'
hasRiskyVerbPrivilege()	'list', 'watch', 'create', 'update', 'patch', 'delete', '*'
hasRiskyResourcePrivilege()	'pod', 'daemonset', 'secrets', '*'
isClusterRoleBinding	'ClusterRoleBinding'
isRoleBinding	'RoleBinding'
isServiceAccount	'ServiceAccount'
isClusterAdmin()	'cluster-admin'
isDefault()	'default'
isKindPod()	'Pod'
isKindDeployment()	'Deployment'
isKindService()	'Service'
isKindDaemonSet	'daemonset'
isKindReplicaSet()	'ReplicaSet'
isVolumeMount()	'volumeMounts'
isReadOnlyRootFileSystem()	'readOnlyRootFileSystem'
isReadOnlyFileSystem()	'readOnlyFileSystem'
isEnabled()	'True', 'true'
isAutomountServiceToken()	'automountServiceAccountToken'
isKeyServiceAccountName()	'serviceAccountName'
isVerb()	'verbs'
isResource()	'resources'
isName()	'name'
isRunAsNonRoot()	'runAsNonRoot'
isRunAsUser()	'runAsUser'
isRunAsGroup()	'runAsGroup'

5.2.3 Evaluation of SLIKUBE+

We evaluate SLIKUBE+ with the dataset used to evaluate SLIKUBE [88]. We describe the dataset filtering criteria in Table 5.10 and attributes of the dataset in Table 5.11. Prior research describe that software projects in GitHub and GitLab often do not represent professional software projects [76]. To mitigate this challenge, researchers used criteria to curate professional software engineering projects such as commits per month, count of contributors [84], [2], [57]. Based on these attributes we use the following criteria to filter our open source repositories:

- **Criterion-1:** The repository must have at least 10% files that are Kubernetes manifests.
- **Criterion-2:** The repository is available to download.
- **Criterion-3:** The repository is not a clone to another repository.
- **Criterion-4:** The repository has at least 2 commits per month. We set this criteria to filter out repository that has little activity.
- **Criterion-5:** The repository has at least five contributors. We set this criteria so that we can eliminate projects for personal use.
- **Criterion-6:** The repository does not contain projects that is used to demonstrate examples, conduct course works and used as book chapter.

5.2.4 Metrics for Frequency 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 5.1 and 5.2 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%.

$$\text{Manifest Coverage } (a) = \frac{\text{\# of configuration parameters used for attack } ia \text{ in the manifest}}{\text{total of configuration parameters needed for attack } a} * 100\% \quad (5.1)$$

$$\text{Repository Coverage } (a) = \frac{\text{\# of configuration parameters used for attack } ia \text{ in the manifests of the repository}}{\text{total of configuration parameters needed for attack } a} * 100\% \quad (5.2)$$

Table 5.10: Dataset for SLIKUBE+

	GitHub	**GitLab**
Initial Repo Count	3,405,303	546,000
Criterion-1 ≥10% YAML files)	6,633	8,194
Criterion-2 (Available)	6,512	7,914
Criterion-3 (Non-duplicates)	4,317	5,871
Criterion-4 (Commit/month≥2.0)	1,325	671
Criterion-5 (Contrib. ≥5)	189	44
Criterion-6 (Not Toy Project)	51	14
Final Repo Count	**51**	**14**

Table 5.11: Dataset Attributes

Attribute	GitHub	GitLab
Repositories	71	21
Kubernetes Objects	3,630	827
Kind Manifests	1,508	369
Helm Charts	82	80
Kubernetes Manifests	1,590	449
Contributors	1,187	977
Commits	37,184	15,870
Size (LOC)	148,588	51,512
Duration	9/2015-12/2021 (75 months)	10/2015-12/2021 (74 months)

5.3 Methodology for RQ 5.3

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. For each configuration parameter, we identify the correspond- ing 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.

5.4 Answer to RQ 5.1

5.4.1 Identification of Pod-related Configuration Parameters

If the NuXMV generates counter-examples, we analyze the counter-examples. We trace the sequence of configurations and events that lead to the violation of the property given an initial configuration of pod. For example, the pod security requirement "*An over privileged running pod will never go to an unsafe state*" can be represented as `!over_privileged_container & pod_state = pod_running)` is `false`. We illustrate the

generated counter-example for the requirement and only represent its relevant configurations in Figure 5.8.

In the initial `state 1.1`, the pod state is `request_initiated`. We find that the state variables in our model related to pod security requirements `hostPID_enabled`, `service_account_automount_token`, `pod_security_exemption_workload_pod`, `security_context_privileged`, and `service_account_privileged` are all set to `TRUE`. In the next `state 1.2`, the pod state is `request_authenticated` and state variables in our model related to pod security requirements `ClusterRole_clusteradmin`, `host_namespace_accesss` and `RCE_vulnerability` becomes `TRUE`. In the next `state 1.3`, the pod state is `request_authorized` and the state variables in our model related to pod security requirements `API_request_verb_UPDATE`, `API_request_verb_PATCH`, `API_request_verb_DELETE` and `API_resource_pod` becomes `TRUE`. In the `state 1.7`, the pod state is `pod_schedule_bind_phase` and the state variables in our model related to pod security requirements `host_secret_exfiltration`, `host_file_system_access` becomes `TRUE`. Eventually, in the `state 1.12`, the pod state becomes `pod_starting`, which symbolizes that the pod has started. In `state 1.13`, the pod state becomes `pod_running`, and the state variable `container_prestop_webhook` becomes `TRUE`. In prior states, the pod has got access to host secret with `host_secret_exfiltration` and `privileged_service_account`. Hence in the next `state 1.14`, the pod again initiates a pod creation request and reaches state `request_pod_creation_initiated`. Finally, in `state 1.22`, the pod reaches the state `pod_running`, which means the over privileged pod initiated another pod with privileged service account.

We identify 21 attack-akin configuration from our counterexample-based analysis. Definitions for each configuration is provided below. The configurations that are common across all six 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 default `serviceAccount`.

1. **Active** `hostIPC` - This configuration provides a container within a pod with the privilege to intercept all IPC communications of the host machine.

2. **Active** `hostNetwork` - The configuration to activate `hostNetwork` with `hostNetwork: true`. This configuration provides applications the privilege to ping and intercept all network interfaces of the host machine.

3. **Active** `hostPID` - The configuration to activate `hostPID` with `hostPID: true`. This configuration allows a pod to access the namespace and find all the processes running on the host.

4. **Capability abuse** - The configuration to activate Linux capabilities. This category includes configurations that allow a pod to gain root-level access. This category includes two sub-categories: `CAP_SYS_ADMIN` and `CAP_SYS_MODULE`.

5. **Escalated child process** - The configuration that allows a child process in a container to gain more privilege than its parent process.

6. **Privileged** `securityContext` - The configuration that allows a privileged `securityContext` by disabling all security features of the pod or container.

7. **Missing resource limit** - The configuration that allows a container within a pod to run without CPU and memory limit, in turn consuming all available resources. The `limits` and `resources` keywords are used to specify resource limits for containers within a pod.

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

9. **Missing network policy** - The configuration of not using the `NetworkPolicy` object to specify network policies for pods. This configuration facilitates unrestricted traffic between pods as `NetworkPolicy` is used to control the flow of traffic between pods.

10. **Missing SSL/TLS for HTTP** - The configuration 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.

11. **Missing admission controller** - The configuration of not using an admission controller with `AdmissionConfiguration`. An admission controller intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized. This configuration allows an authenticated and authorized request to bypass the security compliance for pod creation.

12. **Privileged role** - The configuration that allows excessive permission to a role. A role is a code construct that allows permissions for a particular namespace [59]. Using the `verbs: ["*"]` configuration for role-related rules, a role becomes privileged. Verbs in Kubernetes are used to specify all possible permissions for a Kubernetes pod.

13. **Privileged** `ServiceAccount` - The configuration of creating a privileged `ServiceAccount`. A service account is a non-human account that provides an identity for processes that run in a pod. In Figure 5.9, 'sample-sa' is a privileged `ServiceAccount` as it uses a `Role` and `RoleBinding` with a default namespace.

14. **Privileged** `RoleBinding` - The configuration that 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.

15. **Inactive read-only for root filesystem** - The configuration that allows the mounting of the container's root file system to be writable with `readOnlyRootFileSystem: False`.

16. **Auto mounted token** - The configuration that allows a pod to automatically activate tokens used for service accounts inside the pod with `automountServiceAccountToken: false`. These tokens are used to authenticate requests from processes within the cluster to the Kubernetes API server.

17. **Default** `ServiceAccount` - The configuration that allows a pod to use the default service account.

18. **Privileged default** `ServiceAccount` - The configuration that allows a privileged role to be used by a default service account.

19. **Active** `hostPath` - The configuration that allows mounting of a file or a directory from the host node filesystem into a pod with `active hostPath`.

20. **Inactive** `runAsNonRoot` - The configuration that allows unauthorized write permissions for the filesystem inside a container of a pod using `runAsNonRoot: false`.

21. **Inactive** `runAsUser` - The configuration that allows a container to run as a root user inside a pod using `runAsUser: false`.

Table 5.4 provides a mapping between pod-related security attacks and the derived configuration parameters. We observe, each attack to include a combination configuration parameters, e.g., executing the 'access cluster secret' requires 15 configuration parameters. The implication of this finding is that a single configuration in a Kubernetes manifest cannot lead to any of the studied attacks listed in Section 5.1.6.

Table 5.12: Pod Configuration Parameters that Invoke Security Attacks

Current state	Expected State	Unsafe State	Transition Conditions
pod_running	service_exposed , pod_terminated	pod_creation_request_initiated	(request_authorized & !mutating_validating_admission_controller_available) --> (new_pod_creation_request & (container_breakout | malicious_container | pod_admission_PRIVILEGED) --> ((!admission_namespace_lifecycle & admission_control_bypass)) --> (remote_service_connected_from_cluster & malicious_container)
pod_running	service_exposed, pod_terminated	pod_evicted	(((!pod_cpu_memory_limit_enabled & !pod_cpu_memory_request_limit_enabled) & node_resource_quota_enabled) | (!readinessprobe_enabled & pod_eviction_pod_preemtion & node_resource_quota_enabled) | (!limit_node_PID |!limit_pod_PID | !pod_eviction_policy) | (namespace_resource_quota_enabled & (!pod_cpu_memory_request_limit_enabled & !pod_cpu_memory_request_limit_enabled)))
service_exposed	pod_terminated	pod_unrestricted_communication	(request_authorized & !mutating_validating_admission_controller_available) --> (new_pod_creation_request & (container_breakout | malicious_container | pod_admission_PRIVILEGED) --> ((!admission_namespace_lifecycle & admission_control_bypass)) --> (remote_service_connected_from_cluster & malicious_container) --> (CAP_NET_RAW | CAP_SYS_ADMIN | network_request_other_workload | security_context_privileged | security_context_run_as_user | lsm_SECCOMP_enabled | lsm_APPArmor_enabled | lsm_SELinux_enabled | hostprocess_enabled | hostNetwork_enabled | hostPID_enabled | hostIPC_enabled | host_path_enabled | !container_DropCapabilites | !running_as_NON_ROOT | pod_admission_BASELINE | !pod_admission_RESTRICTED | pod_admission_PRIVILEGED | !pod_admission_ENFORCE | pod_security_exemption_user | pod_security_exemption_namespace | pod_security_exemption_workload_pod | !network_policy_between_pods | docker_socket_enabled | default_namespace | service_account_privileged | service_account_automount_token)

```
-> State: 1.1 <-
  hostPID_enabled = TRUE
  service_account_automount_token = TRUE
  pod_security_exemption_workload_pod = TRUE
  security_context_privileged = TRUE
  admission_controller_image_scan = FALSE
  service_account_privileged = TRUE
-> State: 1.2 <-
  ClusterRole_cluster_admin = TRUE
  host_namespace_access = TRUE
  RCE_vulnerability = TRUE
  pod_state = request_authenticated
-> State: 1.3 <-
  API_request_verb_UPDATE = TRUE
  API_request_verb_PATCH = TRUE
  API_request_verb_DELETE = TRUE
  API_resource_pod = TRUE
-> State: 1.4 <-
  pod_object_created_by_resource_controller = TRUE
  remote_service_connected_from_cluster = TRUE
  pod_state = request_pod_creation_initiated
  ...
-> State: 1.5 <-
  pod_state = desired_pod_state_stored_in_etcd
  ...
-> State: 1.6 <-
  pod_state = pod_schedule_score_phase
  ...
-> State: 1.7 <-
  host_secret_exfiltration = TRUE
  host_file_system_access = TRUE
  pod_state = pod_schedule_bind_phase
  ...
-> State: 1.8 <-
  pod_state = kubelet_receives_podspec
  ...
-> State: 1.9 <-
  pod_state = image_pulled_from_registry
  ...
-> State: 1.10 <-
  pod_state = image_provided_to_cri
  ...
-> State: 1.11 <-
  pod_state = volume_mounted
  ...
-> State: 1.12 <-
  pod_state = pod_starting
  ...
-> State: 1.13 <-
  pod_state = pod_running
  ...
-- Loop starts here
-> State: 1.14 <-
  pod_state = request_pod_creation_initiated
  ...
-> State: 1.15 <-
  pod_state = desired_pod_state_stored_in_etcd
  ...
-> State: 1.16 <-
  pod_state = pod_schedule_score_phase
  ...
-> State: 1.17 <-
  pod_state = pod_schedule_bind_phase
-> State: 1.18 <-
  image_pull_policy = IFNOTPRESENT
  pod_state = kubelet_receives_podspec
  ...
-> State: 1.19 <-
  pod_state = image_provided_to_cri
  ...
-> State: 1.20 <-
  pod_state = volume_mounted
  ...
-> State: 1.21 <-
  pod_state = pod_starting
  ...
-> State: 1.22 <-
  pod_state = pod_running
  ...
-> State: 1.23 <-
  pod_state = request_pod_creation_initiated
  ...
```

Figure 5.8: A counter-example for over privileged pod

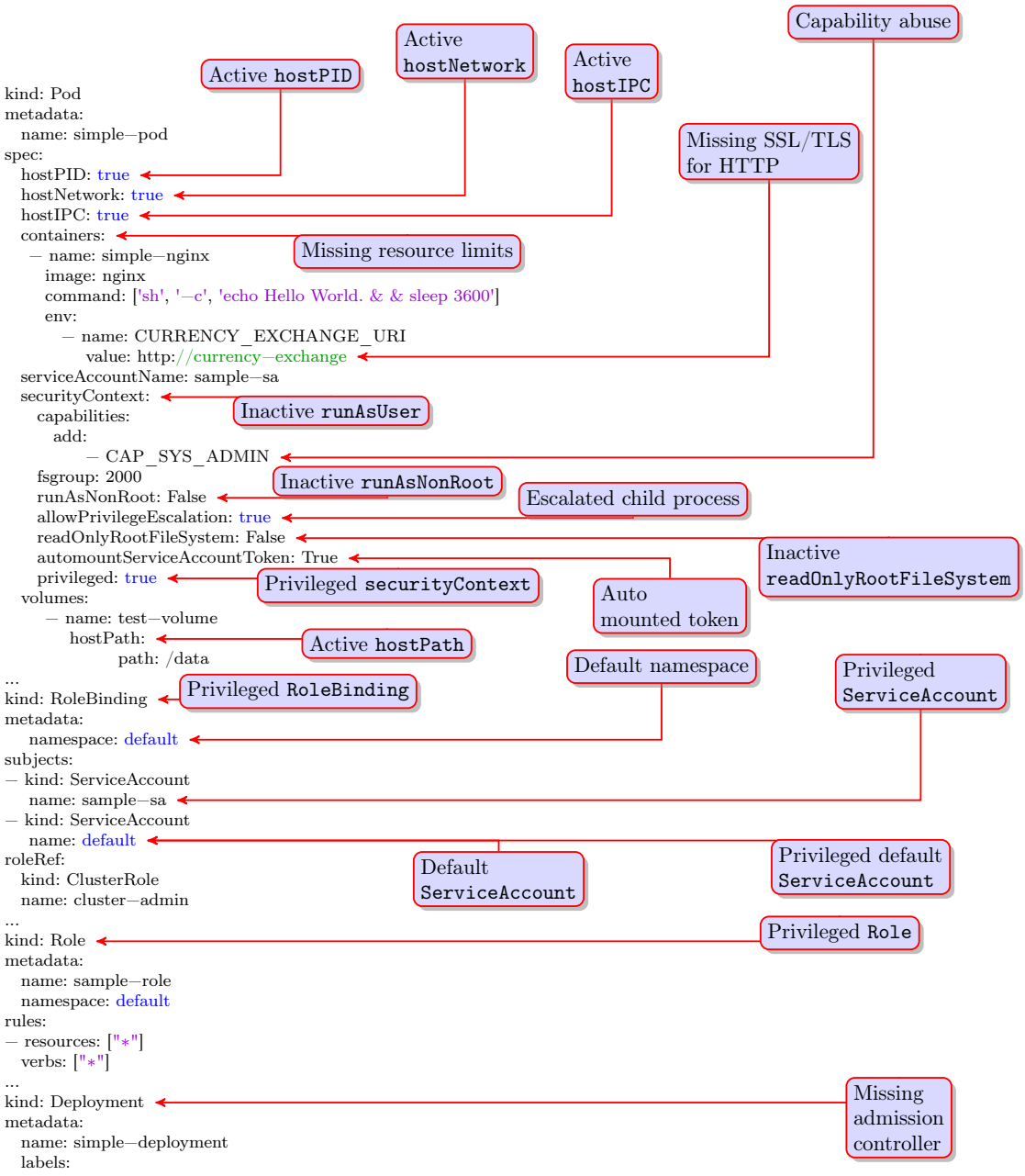

Figure 5.9: Example code snippet to demonstrate attack-akin configurations.

5.5 Answer to RQ 5.2

5.5.1 Frequency of Pod Configuration Parameters

We report 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. We describe the manifest-based attack coverage (MAC) and repository-based attack coverage (RAC) for GitHub and GitLab dataset for 6 attacks in Table 5.13 and Table 5.14 respectively. 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. We observe missing network policy is the most frequently occurring category. We describe the evaluation result of SLIKUBE+ on the dataset in Table 5.15. We also describe the pod configuration parameters related to security attacks in Table 5.16.

Table 5.13: Manifest-based Attack Coverage (MAC)

Attack Name	MAC							
	GitHub				GitLab			
	Min	Median	Avg.	Max	Min	Median	Avg.	Max
Access cluster secrets	0.0	16.6	17.4	55.5	0.0	5.5	12.9	55.5
Dashboard maneuver	0.0	16.6	17.4	55.5	0.0	5.5	12.9	55.5
Database tampering	0.0	17.6	21.6	58.8	0.0	17.6	17.6	47.0
Denial of service	0.0	13.3	17.5	53.3	0.0	6.6	13.2	46.6
Etcd takeover	0.0	16.6	17.4	55.5	0.0	5.5	12.9	55.5
Pod disruption	0.0	16.6	17.4	55.5	0.0	5.5	12.9	55.5
Combined	0.0	16.6	18.1	58.8	0.0	5.5	13.7	55.5

Table 5.14: Repository-based Attack Coverage (RAC)

Attack Name	RAC							
	GitHub				GitLab			
	Min	Median	Avg.	Max	Min	Median	Avg.	Max
Access cluster secrets	0.0	16.6	17.7	55.5	0.0	16.6	21.5	55.5
Dashboard maneuver	0.0	16.6	17.7	55.5	0.0	16.6	21.5	55.5
Database tampering	0.0	17.6	21.9	58.8	0.0	23.5	25.3	47.0
Denial of service	0.0	13.3	17.5	58.8	0.0	20.0	22.5	46.6
Etcd takeover	0.0	16.6	17.4	55.5	0.0	16.6	21.5	55.5
Pod disruption	0.0	16.6	17.4	55.5	0.0	16.6	21.5	55.5
Combined	0.0	16.6	18.3	58.8	0.0	16.6	22.3	55.5

Table 5.15: Kubernetes Security Misconfigurations in OSS

Misconfigurations	**GitHub**	**GitLab**
Active `hostIPC`	1	0
Active `hostPID`	5	0
Active `hostNetwork`	11	3
Escalated child process	3	0
Capability abuse	0	20
Privileged `securityContext`	3	9
Missing Resource Limit	69	10
Missing Network Policy	1,508	396
Default Namespace	95	2
Missing SSL/TLS for HTTP	395	217
Missing admission controller	1509	395
Privileged Role	47	35
Privileged `ServiceAccount`	73	51
Privileged `RoleBinding`	11	1
Inactive read-only root file system	227	43
Auto mount token	1	0
Privileged default `ServiceAccount`	0	1
Default `ServiceAccount`	784	125
Active `hostPath`	86	12
Inactive `runAsNonRoot`	560	72
Inactive `runAsUser`	556	72

Table 5.16: Mapping of Configuration Parameters to Pod-related Attacks

Attack Name	Config. Params.	Count
Access cluster secrets	cluster-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	serviceAccount, 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

5.5.2 Comparison of SLIKUBE+ with Existing Tools

We compare our SLIKUBE+ with SLIKUBE tool and existing static analysis tools for identifying Kubernetes security misconfigurations. SLIKUBE+ reports 12 more security misconfigurations than SLIKUBE [3]. Similar to SLIKUBE, compare our tool SLIKUBE+ with four state-of-the-art static analysis tool as follows: Checkov [14], KubeLinter [58], Datree [23], and Snyk [103]. We inspect the policy or rules of each the static analysis tool from their online documentation and idnetify which of the categories of our SLIKUBE+ tool are identified by each of these tools. We observe that only SLIKUBE+ detects all 23 category of security misconfigurations. Checkov, KubeLinter, Datree and Snyk do not identify 7, 3, 4 and 7 categories of Kubernetes security misconfigurations respectively.

Table 5.17: Comparison of SLIKUBE+ with Existing Tools

Misconfiguration Name	**SLIKUBE+**	**Checkov**	**KubeLinter**	**Datree**	**Snyk**
Active `hostIPC`	✓	✓	✓	✓	✓
Active `hostNetwork`	✓	✓	✓	✓	✓
Active `hostPID`	✓	✓	✓	✓	✓
Capability Abuse	✓	✓	✓	✓	✓
Escalated child process	✓	✓	✓	✓	✓
Privileged `securityContext`	✓	✓	✓	✓	✓
Missing Resource Limit	✓	✓	✓	✓	✓
Missing SSL/TLS for HTTP	✓	×	×	×	×
Default namespace	✓	✓	✓	✓	✓
Missing network policy	✓	×	✓	×	✓
Missing admission controller	✓	×	×	×	×
Privileged role	✓	✓	✓	✓	✓
Privileged `ServiceAccount`	✓	✓	✓	✓	✓
Privileged `RoleBinding`	✓	×	✓	✓	×
Inactive read-only for root filesystem	✓	✓	✓	✓	✓
Auto mounted token	✓	✓	×	✓	×
Privileged default `ServiceAccount`	✓	✓	✓	✓	✓
Active `hostPath`	✓	×	✓	✓	×
Default `ServiceAccount`	✓	✓	✓	✓	✓
Inactive `runAsNonRoot`	✓	×	✓	✓	✓
Inactive `runAsUser`	✓	×	✓	✓	✓

Table 5.18: Comparison between SLIKUBE+ and SLIKUBE

Misconfiguration Name	SLIKUBE+	SLIKUBE
Active `hostIPC`	✓	✓
Active `hostPID`	✓	✓
Active `hostNetwork`	✓	✓
Escalated child process	✓	✓
Capability Abuse	✓	✓
Privileged `securityContext`	✓	✓
Missing Resource Limit	✓	✓
Missing SSL/TLS for HTTP	✓	✓
Missing network policy	✓	×
Default namespace	✓	×
Missing admission controller	✓	×
Privileged role	✓	×
Privileged `ServiceAccount`	✓	×
Privileged `RoleBinding`	✓	×
Inactive read-only root filesystem	✓	×
Auto mount token	✓	×
Privileged default `ServiceAccount`	✓	×
Active `hostPath`	✓	×
Default `ServiceAccount`	✓	×
Inactive `runAsNonRoot`	✓	×
Inactive `runAsUser`	✓	×

5.6 Answer to RQ 5.3

5.6.1 Identification of Pod States Related to Security Attacks

NuXMV generates counterexamples when there is a pod security requirement violation in our FSM for a pod. We observe the change of individual state variables in the counterexamples. We construct transition conditions as propositional logic. The change of state variables causes the change in the transition condition. When the transition condition changes, such as `True` from `False`, we observe a transition to an event from the previous event. From Figure 5.8, we observe that the change of state variables in every `pod_state` of the FSM allows transition to the subsequent event. We extract the events from the counterexample, the change in the value of state variables, and `pod_state`.In Table 5.12, in the leftmost column "Current State", we specify the state of FSM for a pod from before transition such as `pod_running` state. Then, in the next column, "Expected State," we list all the safe states of the FSM for a pod transitioning from "Current State" state. For instance, a pod should be in `service_exposed` or `pod_terminated` state as the next state of the `pod_running` state. In the third column "Unsafe state," we provide the state of FSM for a pod for transition from "Current State" due to a combination of misconfigurations. For instance, a transition

from pod_running state to pod_creation_request_initiated state can happen due to combination of misconfigurations in transition conditions. In the rightmost column " Transition Conditions", we provide the combination of misconfigurations which lead a pod to unsafe state such as from pod_running state to pod_creation_request_initiated state.

We identify the following pod events that occur prior to insecure pod events from the counterexamples as follows:

Request authenticated (request_authenticated): This is the pod event where the Kubernetes API server authenticates the pod creation request from the practitioner.

Request authorized (request_authorized): This is the pod event where the Kubernetes API server authorizes the pod creation request.

Pod creation request initiated (request_pod_creation_initiated:): This is the pod event where the Kubernetes API server creates a pod object from the authenticated and authorized pod creation request.

Desired pod state stored in etcd (desired_pod_state_stored_in_etcd): This is the pod event where the Kubernetes API stores the pod object specification in the etcd database as a desired pod state.

Pod scheduling score phase(pod_schedule_score_phase): This is the pod event where the Kubernetes scheduler calculates scores for scheduling a pod.

Pod scheduling bind phase(pod_schedule_bind_phase): This is the pod event where the Kubernetes scheduler binds the pod to a specific worker node.

Kubelet receives pod specification (kubelet_receives_podspec): This is the pod event where the kubelet agent in the worker node receives the pod specification from the Kubernetes API server.

Image pull from registry (image_pulled_from_registry): This is the pod event where the kubelet agent in the worker node pulls image from the container image registry.

Container registry poisoned (`container_registry_poisoned`): This is the pod event where a malicious agent provides malicious image to kubelet and pushes malicious image to the container image registry with stolen credentials from kubelet agent.

Image provided to container runtime interface (`image_provided_to_cri`): This is the pod event where the kubelet agent provides the container image to the container runtime interface (CRI).

Volume mounted(`volume_mounted`): This is the state of pod event where the pod mounts volume for the container and allocates persistent storage for stateful pods.

Pod starting(`pod_starting`): This is the pod event where the pod has a network, has storage, and at least one container started running inside the pod.

Pod running(`pod_running`): This is the pod event where the container state is `RUNNING`.

Pod evicted(`pod_evicted`): This is the pod event where one misconfigured pod evicts other pods from the worker node due to high resource consumption causing a resource-related denial of service attack.

Service exposed(`service_exposed`): This is the pod event where the running pod creates an endpoint and exposes the IP address so that other services can access it.

Pod unrestricted communication (`pod_unrestricted_communication`): This is the pod event where the pod or exposed service has some misconfiguration that allows open network communication and lateral movement inside the Kubernetes cluster.

Remote service connected (`remote_service_connected`): This is the pod event where the pod or exposed service has exposed shell to an unauthorized remote machine.

Among the states involved in the counterexamples, we observe the following states are involved in security attacks as described in Table 5.19.

Table 5.19: 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

Chapter 6

Authentic Learning for Learning Kubernetes Security Misconfiguration Analysis

As per the 2021 CNCF annual survey, 96% of the 19,000 practitioners surveyed are either using or evaluating Kubernetes for their respective organizations [43]. Furthermore, the survey highlights that approximately 5.6 million developers globally are utilizing Kubernetes [43]. However, practitioners also acknowledge that Kubernetes has evolved into a complex software platform with a steep learning curve, emphasizing the need for a skilled workforce proficient in Kubernetes. [44] [40]. According to the Redhat survey conducted in 2021, 94% of practitioners reported experiencing incidents related to Kubernetes security misconfigurations [94]. To address this issue and cultivate a more skilled cybersecurity workforce in the industry with expertise in Kubernetes, one potential solution is to educate students on Kubernetes security misconfigurations

Previous research has demonstrated that authentic learning exercises have proven effective in enhancing students' understanding of various subjects, such as mobile application security [83] and infrastructure-as-code (IaC) [89]. Building upon this, we formulate the hypothesis that an authentic learning-based exercise will help students in comprehending Kubernetes security misconfigurations.

- **RQ 6.1**: How to design authentic learning-based exercise to help students for secure development of Kubernetes Manifests?
- **RQ 6.2**: How does authentic learning help students to learn about the secure development of Kubernetes Manifests?

- **RQ 6.3**: What instructor-related attributes are useful for students in an authentic learning-based exercise used for Kubernetes security misconfiguration analysis?

6.1 Methodology

In this section, we describe our methodology by discussing the authentic learning exercise design for Kubernetes security misconfiguration analysis. Next, we describe our questionnaire design and deployment process. After that, we discuss the analysis of our questionnaire results.

6.1.1 Authentic Learning Exercise Design

We designed our authentic learning-based exercise for the Kubernetes security misconfiguration and deployed it in the "Software Quality Assurance" course in the fall 2022 semester at Auburn University. After collecting the feedback from the students, we redeployed our authentic learning-based exercise into two different classes at Auburn University and Tuskegee University in the spring 2023 semester, respectively. After that, we again deployed the exercise in fall 2023 and spring 2024 semester at Auburn University. We follow Herrington et al.'s guidelines for creating authentic learning based exercise [38], [37]. We construct the three steps of our authentic learning-based exercise as follows:

Concept Dissemination

In this step, we provide authentic context to the student by disseminating the knowledge related to Kubernetes security misconfigurations [38]. In the class, we introduce students to containers and tools to automate the management of containers. We specifically focus on one container management and orchestration tool, Kubernetes. Practitioners use configuration files known as manifests and the 'kubectl' tool to execute the manifests to manage containers in the Kubernetes cluster. Practitioners develop the manifests using a language called Yet Another Markup Language (YAML) with .yaml or .yml extension. We introduce the students

to Kubernetes security misconfigurations and the use of static security analysis tools to identify Kubernetes security misconfigurations in Kubernetes manifests. National Institute of Standards and Technology (NIST) defines a security misconfiguration as a setting within a computer program that violates a configuration policy, or that permits unintended behavior that impacts the security posture of a system [77]. In Figure 6.1, we demonstrate a sample Kubernetes Manifest with security misconfigurations for concept dissemination in our authentic learning-based exercise in class.

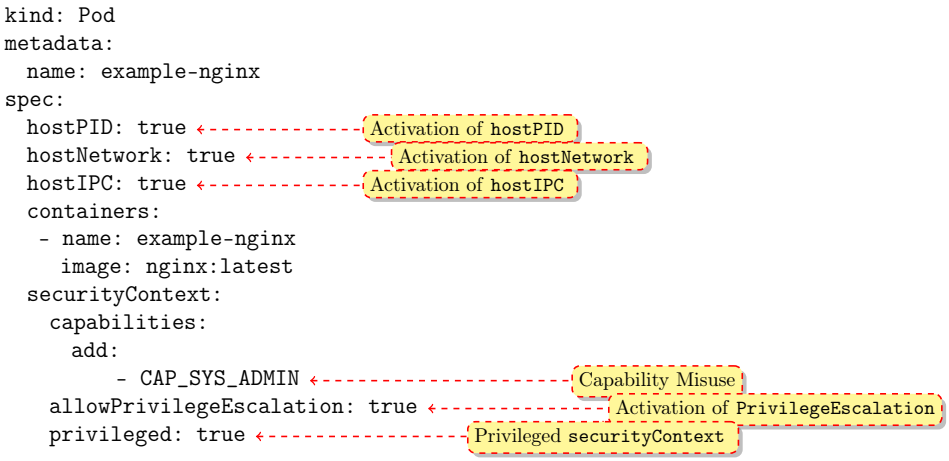

Figure 6.1: A Sample Kubernetes Manifest (example-nginx.yaml) with Security Misconfigurations for Concept Dissemination in the Authentic Learning-based Exercise

Hands-on Exercise

In this step, the students learn to conduct authentic activities, and get access to expert assessment and support from the instructor while participating in the hands-on exercise taught in the class [38]. The students also get the opportunity to get perspective on multiple roles both as a practitioner who develops the manifest and who scans the Kubernetes manifest for security misconfiguration [37]. In the hands-on exercise, we instructed each students to participate in the exercise individually. We asked the students install 'Docker' on their computers. We introduce the students to an open-source static security analysis tool called SLIKUBE [3]. We conduct a live demonstration for the students on using SLIKUBE to detect

security misconfigurations in Kubernetes. We demonstrated how to download the tool from DockerHub [25] and instructed them to run it inside the Docker container. We explain to the students the detailed output of the SLIKUBE presented as a CSV file, such as the directory column, the path of the manifests column, specific misconfiguration columns, and the total column that reports the total occurrences of misconfiguration in Kubernetes manifest. For instance, when students scan the 'example-nginx.yaml' Kubernetes manifest in Figure 6.1 as demonstrated by the instructor, the security analysis tool SLIKUBE reports 6 security misconfigurations: activation of `hostPID`, activation of `hostNetwork`, activation of `hostIPC`, capability misuse, activation of `privilege escalation`, privileged `securityContext` with 1 occurrence for each of the misconfigurations.

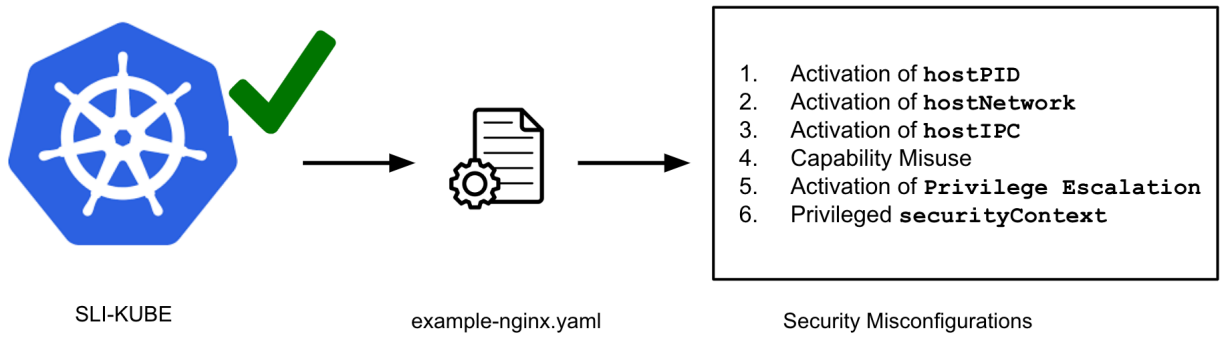

Figure 6.2: Overview of in-class experience to detect Kubernetes security misconfigurations in Kubernetes manifests

Post-lab Exercise

In the post-lab exercise, we provide students with Kubernetes manifests from open-source repositories(OSS) such as GitHub and GitLab. We ask the students to execute the security static analysis tool SLIKUBE on the provided Kubernetes manifests. After running the SLIKUBE on provided manifests, we ask the students to analyze the output of SLIKUBE and report the top 3 most frequent Kubernetes security misconfigurations with the definition and consequences for the corresponding Kubernetes security misconfigurations. Moreover,

we asked students to complete a survey on authentic learning-based exercises. In Figure 6.3, we demonstrate all three steps of our authentic learning-based exercise.

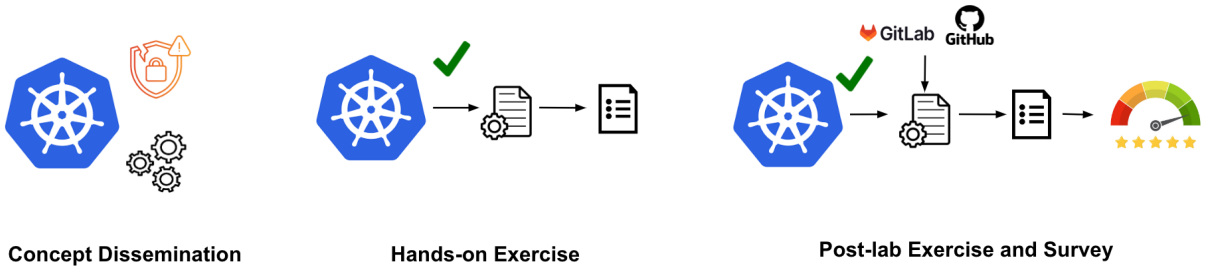

Figure 6.3: Overview of Authentic Learning-based Kubernetes Security Misconfiguration Analysis

6.1.2 Questionnaire Design and Deployment

Previous research has demonstrated that authentic learning exercises have proven effective in enhancing students' understanding of various subjects, such as mobile application security [83] and infrastructure-as-code (IaC) [89]. We take motivation from prior work and expanded their questionnaires and included in our questionnaire set. We use questionnaires to collect feedback from the students after the post-lab exercise on the usefulness of the authentic learning exercise for Kubernetes security misconfiguration analysis. We design our questions related to students' backgrounds with the following questions:

(i) students' academic background and prior experience in software engineering,

(ii) students' experience of the authentic learning-based exercise,

(iii) students' perception of the usefulness of the exercise, and

(iv) students' perception of the instructor.

We deploy the survey using the online Qualtrics platform. We describe our survey design and deployment in this section.

Question-Related to Students' Background and Experience in Software Engineering

As part of our study, we administered a questionnaire consisting of three questions aimed at assessing the students' background in the class. The participating students are from "Software Quality Assurance" course. As part of our study, we ask the following question to assess the students' academic background in the class. We also ask questions to gather information regarding the students' prior experience in cybersecurity, software quality assurance activities, and program analysis tools before participating in the workshop exercise. The specific questions were as follows:(i) How would you rate your experience in cybersecurity prior to the workshop? (ii) How would you rate your experience in software quality assurance activities prior to the workshop? (iii) How would you rate your experience with program analysis tools prior to the workshop?

We follow the recommendations of Kitchenham [52] and create a five-item Likert scale as follows: 'Expert', 'Somewhat Expert', 'Knowledgable', 'Little knowledge' and 'No knowledge'.

Question Related to Usefulness of Authentic Learning-based Exercise

As part of our evaluation process, we administered a six-question questionnaire to the students to assess the perceived usefulness of our authentic learning-based motivated from the prior work in IaC [89]. Additionally, we included one question specifically targeting the workshop's usefulness in facilitating learning about Kubernetes misconfigurations. The questions and response options were as follows:

(i) Which part of the authentic learning experience was useful for you? - Pre-stage
(ii) Which part of the authentic learning experience was useful for you? - In-class experience
(iii) Which part of the authentic learning experience was useful for you? - Post-stage
(iv) Which part of the authentic learning experience was useful for you? - Pre-stage and

in-class experience

(v) Which part of the authentic learning experience was useful for you? - Pre-stage and post-stage

(vi) Which part of the authentic learning experience was useful for you? - All three steps

We ask the participating students to rate the usefulness of each aspect of the authentic learning experience using a five-item Likert scale, with response options ranging from 'Extremely useful', 'Useful', 'Moderately useful', 'Little useful', to 'Not at all useful'.

Question Related to Student's Perception on Instructor Attributes

Prior research revealed that instructor attribute is correlated with students' learning experience [67], [97]. Researchers also report that instructor background [47], [82], enthusiasm [50], and engagement [69]are correlated with student's learning experiences. In our study we considered five instructor-related attributes as follows: 'academic background', industry background', 'conducted research', 'enthusiasm', and 'in-person engagement'. As part of our study, we provided a set of questionnaires to the students, consisting of six questions that aimed to assess the perceived usefulness of our instructor attributes for authentic learning-based exercises. We use a five-item Likert scale: 'Extremely Useful', 'Useful', 'Moderately Useful', 'Little Useful' and 'Not at all Useful' to answer each question.

The questions and response options were as follows:

(i) Which attributes of the instructor was useful for you ? - Academic background

(ii) Which attributes of the instructor was useful for you ? - Industry background

(iii) Which attributes of the instructor was useful for you ? - Conducted research

(iv) Which attributes of the instructor was useful for you ? - Enthusiasm

(v) Which attributes of the instructor was useful for you ? - In-person engagement

(vi) Which attributes of the instructor was useful for you ? - All of the above

Question Related to Student's Perception on Usefulness of Authentic-learning Exercise

We provide two questions that aimed to assess the perceived usefulness of our exercise in learning Kubernetes misconfigurations and secure automated Kubernetes configuration management for authentic learning-based exercise. We use the questions based on the usefulness of authentic learning in IaC [89]. The questions and response options were as follows:

(i) Did the workshop help you to learn about Kubernetes misconfigurations?

(ii) Did the workshop help you to learn about automated configuration management tools and how they work?

We use a five-item Likert scale: 'Extremely Helpful', 'Helpful', 'Somewhat Helpful', 'Little Helpful' and 'Not at all Helpful' to answer each question. Moreover, we ask students for additional

Questionnaire Deployment

We deployed the questionnaires using the online Qualtrics platform. The students provided answers in the questionnaires after completing the post-lab exercise. We asked for consent from the students before their participation.

6.1.3 Questionnaire Analysis

We use the responses from the students participating in the online questionnaire to answer our research questions.

Methodology to Answer RQ 6.1

We answer RQ1 by analyzing the responses from the questionnaire described in Sections 6.1.2 and 6.1.2. We consider the educational background of the students in the class, whether undergraduates or graduates. We also consider students' prior experiences in software engineering, namely (i) students' experience in software quality assurance, (iii) students' experience in cybersecurity, and (iv) students' experience in program analysis tools. Regarding students' prior experience, we recorded their responses using a five-item Likert scale: 'Expert,' 'Somewhat Expert,' 'Knowledgeable,' 'Little Knowledge,' and 'No Knowledge.' We also consider the usefulness of the students' authentic learning experience on each step of the authentic learning exercise, namely 'Pre-stage', 'In-class experience', 'Post-stage', 'Pre-stage and in-class experience', 'Pre-stage and post-stage', and 'All three steps'. We record their responses using a five-item Likert scale: 'Extremely Useful', 'Useful', 'Somewhat Useful', 'Little Useful' , and 'Not Useful'. Furthermore, we address the students' comments as part of their feedback. We report the analysis of students' feedback for authentic learning. Based on the students' feedback and comments, we modify and design our authentic-learning-based exercise to deploy them in the subsequent semester for their effective learning.

Methodology to Answer RQ 6.2

We answer RQ2 by analyzing the responses from the questionnaire described in Section 6.1.2. We report the usefulness of our authentic learning-based exercise by getting the students' responses on (i) whether the exercise helps the students in learning Kubernetes security misconfiguration and (ii) whether the exercise helps the students learn about automated configuration management tools and how they work. We record their responses using a five-item Likert scale: 'Extremely Useful', 'Useful', 'Somewhat Useful', 'Little Useful' and 'Not Useful'. We report the student's perception of the usefulness of the authentic learning exercise.

Methodology to Answer RQ 6.3

We answer the RQ3 by analyzing the responses from the questionnaire described in Section ref survey-design-instructor-attributes. We consider five instructor-related attributes: 'Academic background', 'Industry background', 'Conducted research', 'Enthusiasm' and 'In-person engagement'. Additionally, we include an option for students to indicate whether all the attributes are useful by providing 'All of the above' option. We record their responses using a five-item Likert scale: 'Extremely Useful', 'Useful', 'Somewhat Useful', 'Little Useful' and 'Not useful'. We report the perceived usefulness of instructor attributes for authentic learning.

6.2 Results

During our first deployment of the exercise in the fall 2022 semester, we conducted data collection by administering a survey to students enrolled in the "Software Quality Assurance" course at the Auburn University. A total of 61 responses were collected from the students, providing valuable feedback on the authentic learning-based exercise. Based on the feedback received in the fall 2022 semester, we redeployed the authentic learning-based exercise in Spring 2023. We repeat the process and deploy in the Fall 2023 and Spring 2024 semesters. In total, we collect 246 responses from the students. Each student who participated in the exercise completed all 3 steps and completed the survey. To provide further insights into the survey participants, Figure 6.4 showcases the distribution of students based on their educational background during their participation in the survey. Notably, we observe that 82.93% of the students identified themselves as undergraduate seniors, while 15.45% were graduate master's students. Furthermore, we notice that a minority of the students, comprising less than 2% of the total, consisted of graduate PhD students and junior undergraduates. In Table 6.1, we provide the educational background of the participating students per semester.

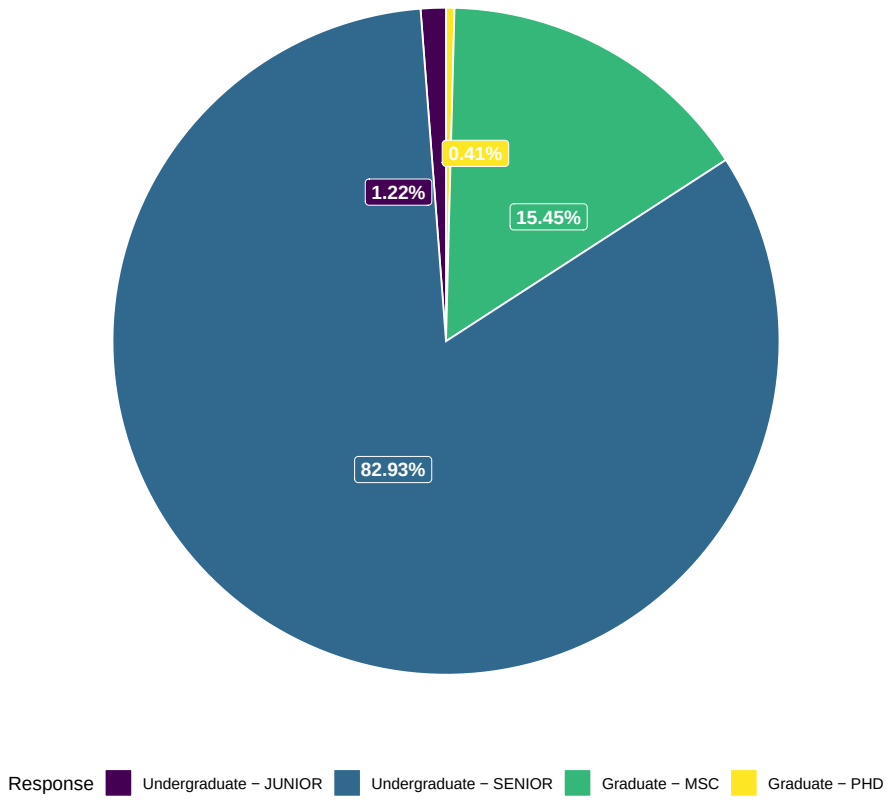

Figure 6.4: Educational Background of Students Participating in the Authentic Learning-based Exercise

6.2.1 Answer to RQ 6.1

We answer RQ1 by presenting the findings regarding the perception of students based on their diverse backgrounds, including their educational level, expertise in software quality assurance, expertise in software security, and expertise in static analysis tools. We report our findings related to students' perception of learning security misconfigurations in Figure 6.5, Figure 6.6, Figure 6.7 and Figure 6.8, respectively. We also report our findings related to students' perception in learning about automated configuration management tools and how the tools work in Figure 6.9, 6.10, 6.11 and 6.12

Table 6.1: Educational Background of Participating Students

Semester	Undergraduate Junior	Undergraduate Senior	Graduate Masters	Graduate PhD	Total
Fall 2022	0	55	6	0	61
Spring 2023	1	54	14	1	70
Fall 2022	0	44	16	0	60
Spring 2024	2	51	2	0	55
Total	3	204	38	1	246

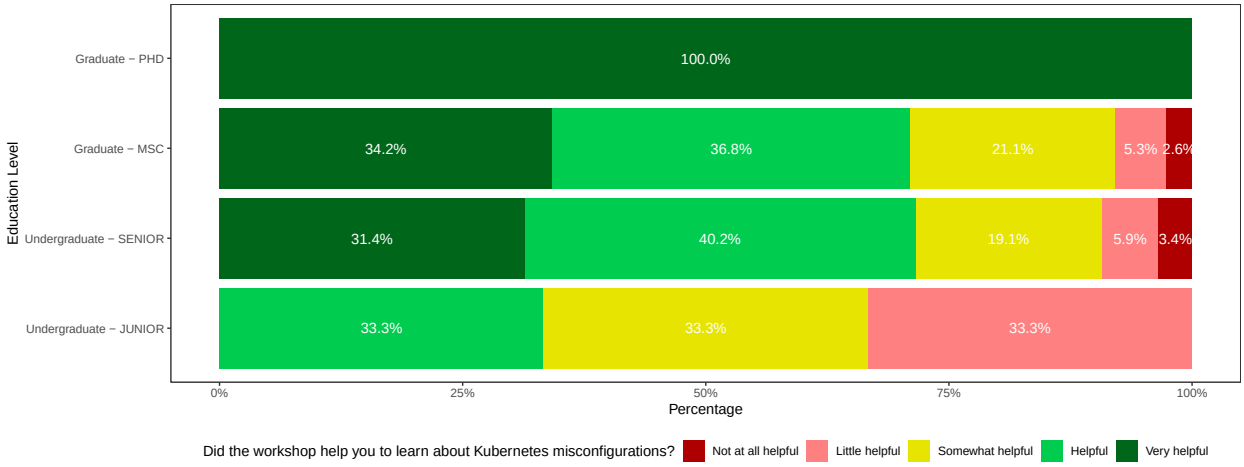

Figure 6.5: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Educational Background

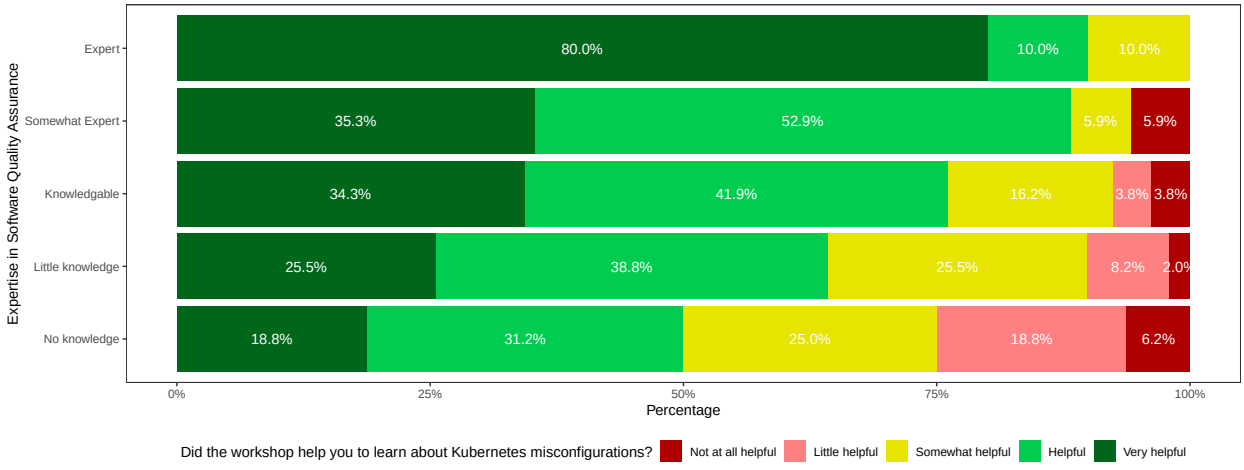

Figure 6.6: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Software Quality Assurance

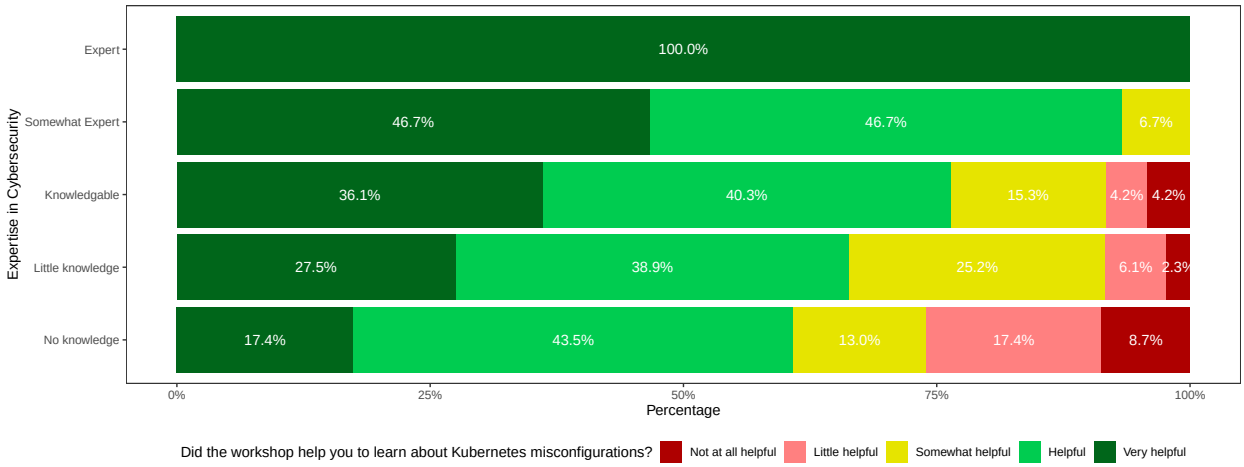

Figure 6.7: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Cybersecurity

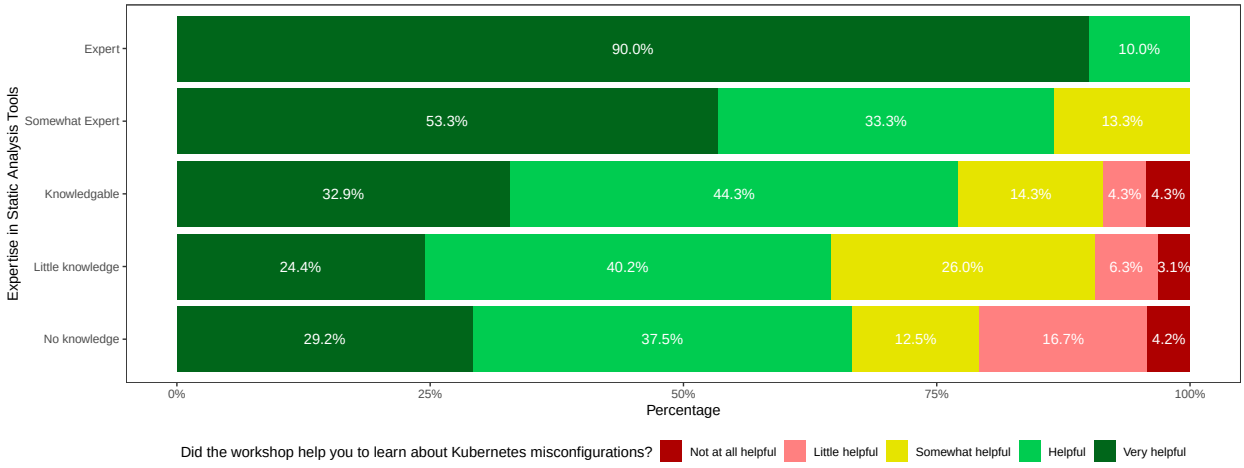

Figure 6.8: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Static Analysis Tools

Upon analysis, we observed that a significant majority of graduate master's students (92.1%), undergraduate senior students (90.7%),and graduate PhD students (100%) find our designed authentic learning-based exercise to be helpful in learning about Kubernetes misconfigurations. We also observe 92.7% of the undergraduate senior students, 92.1% graduate master's students, and 100% graduate PhD students report that they learn about automated configuration management tools and how they work. However, it is noteworthy that only 66.66% undergraduate students reported finding the exercise to be helpful in learning

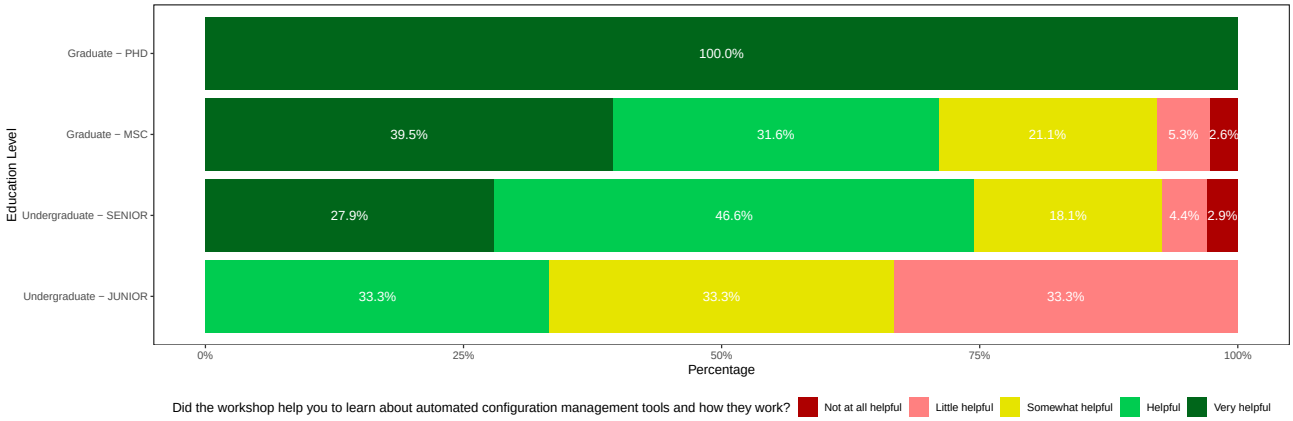

Figure 6.9: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Automated Configuration Management Tools and How the Tools Work Based on Their Educational Background

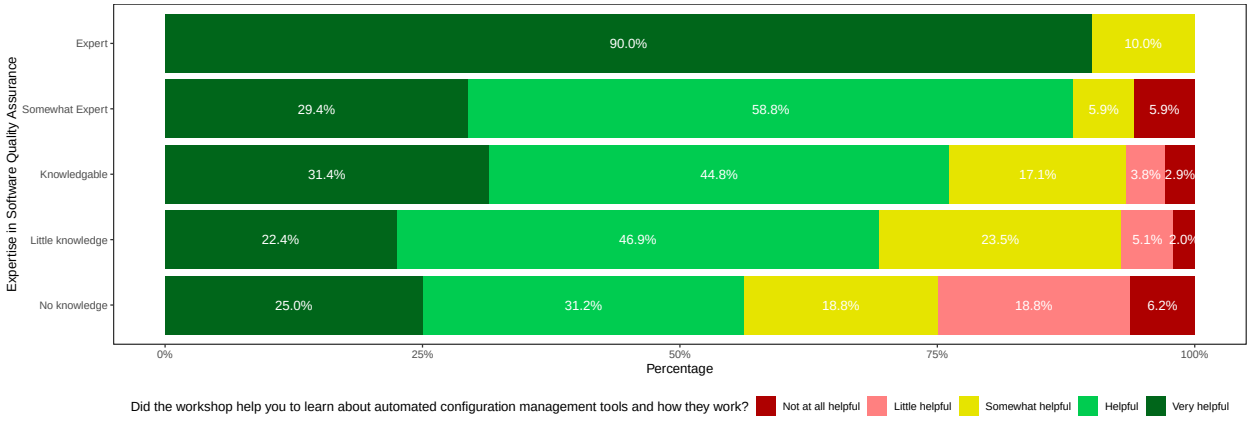

Figure 6.10: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Automated Configuration Management Tools and How the Tools Work Based on Their Expertise in Software Quality Assurance

Kubernetes security misconfigurations and automated configuration management tools, respectively. In Figure 6.5, and 6.9, we provide an overview of the student's perception of our authentic learning-based exercise based on their education level.

From Figure 6.6, 6.7, 6.8, 6.10, 6.11, and 6.12, we observe that the students who has prior expertise in software quality analysis, software security and static analysis report the exercises are helpful for them to learn about Kubernetes security misconfigurations and automated configuration management tools. Also, We find that the students who evaluate themselves as "Expert" and "Somewhat Expert" have learned better than other students

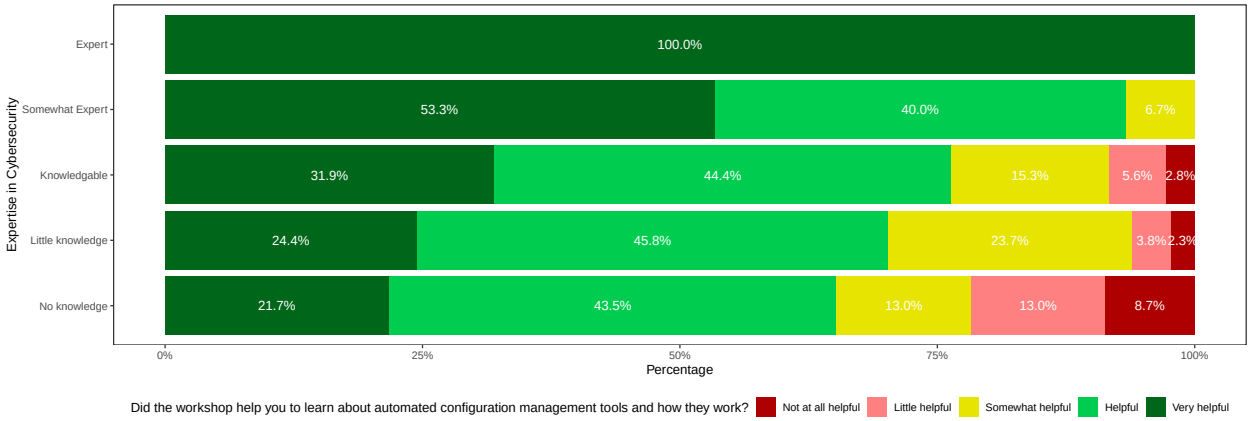

Figure 6.11: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Automated Configuration Management Tools and How the Tools Work Based on Their Expertise in Cybersecurity

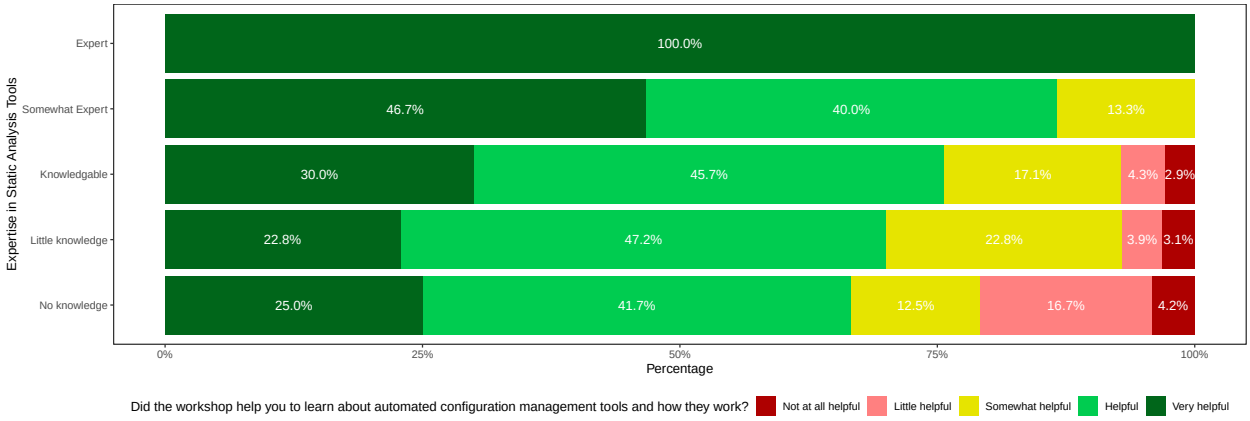

Figure 6.12: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Automated Configuration Management Tools and How the Tools Work Based on Their Expertise in Static Analysis Tools

in the class. One potential reason can be the lack of adequate technical background to follow through with the hands-on exercise and perform post-lab exercise. For instance, one student reports that *"I could not get docker to install correctly. I don't know why, but with many of the recent workshops, even if I follow the instructions one-to-one, my laptop just doesn't agree with the software. It is pretty frustrating to not even be able to start these workshops."* Another student reports that *"This workshop was very informative, and using docker was a plus because it is useful in industry."*. Our results suggest that our designed workshop exercise is more suitable for students with prior relevant technical background.

In the subsequent iteration of our exercise deployment, we integrate prerequisite technical background concept dissemination and more detailed installation instruction in the workshop to make our authentic learning-based exercise more usable for students with little to no background in security, static analysis, and software quality assurance.

These findings demonstrate the importance of considering students' prior software engineering background and educational levels when designing and implementing authentic learning-based exercises. The results highlight the positive impact of our authentic learning-based exercise on students with higher educational levels and expertise in software quality assurance, software security, and static analysis tools.

6.2.2 Answer to RQ 6.2

We report students' perceptions on the usefulness of the authentic learning exercise steps. We find that students perceive the usefulness of our 'Pre-Stage' step, 'In-Class Experience' step and 'Post-Stage' step of our authentic learning exercises are 88.7%, 93.1% , and 91.5% , respectively. Based on the students response on the usefulness of the authentic learning-based exercise steps, we observe that 29.3%, and 48.8% of the students find all the three steps "Extremely useful" and "Useful" respectively. We notice that only 2.4% of the students report that all 3 steps of our designed authentic learning-based exercise are "Not useful at all".

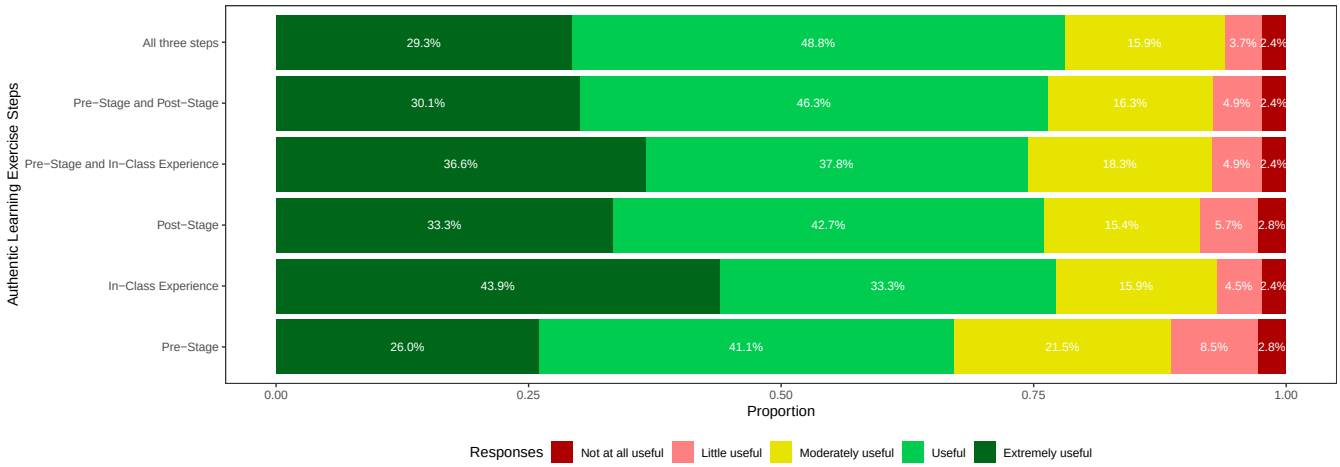

Figure 6.13: Reported Perception of Students on the usefulness of Authentic Learning-based Exercise

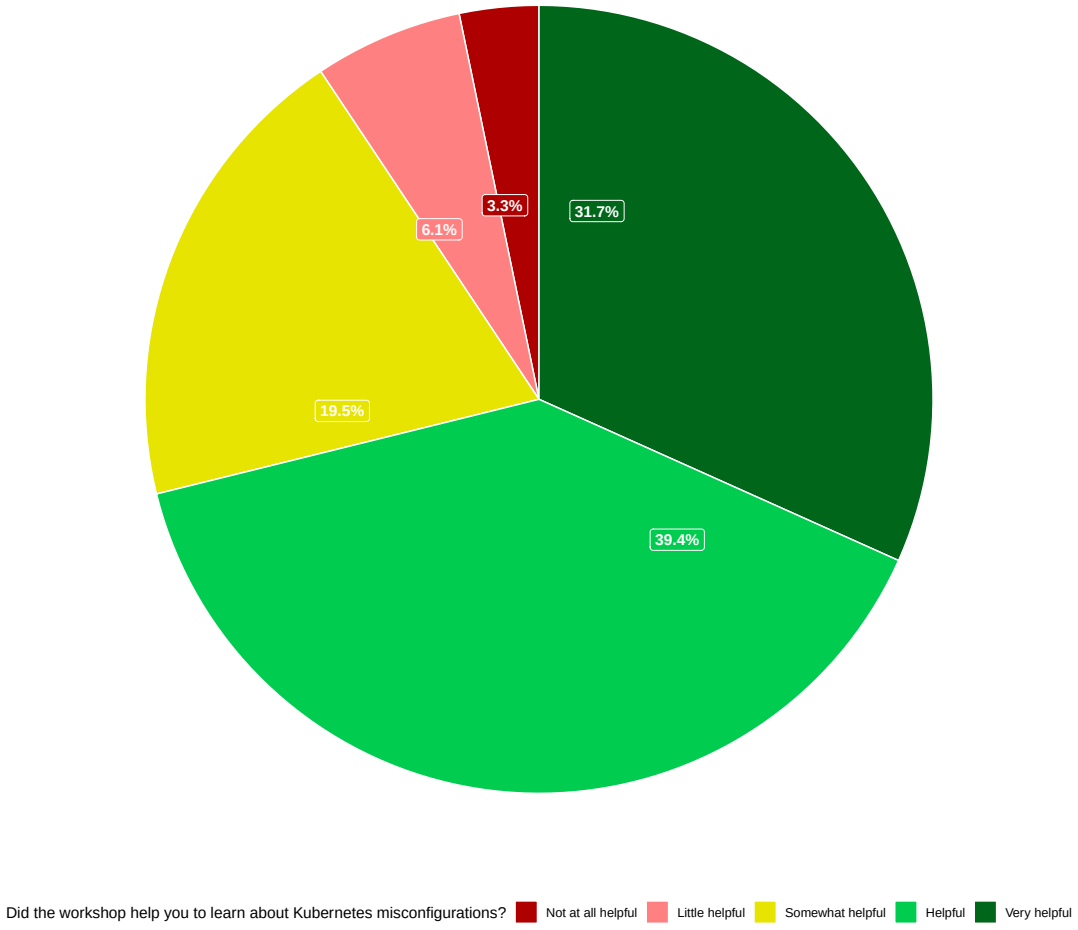

Figure 6.14: Overall Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Security Misconfiguration

We demonstrate the overall perception among all the students in the class on how this authentic learning-based exercise helps them understand Kubernetes security misconfiguration in Figure 6.14. We find that 31.7%, 39.4% , 19.5% of the students report that they find the authentic learning-based exercise "Very helpful", "Helpful" and "Somewhat helpful" respectively. We find 6.1% and 3.3% students report the exercise is "Little helpful" and "Not at all helpful".

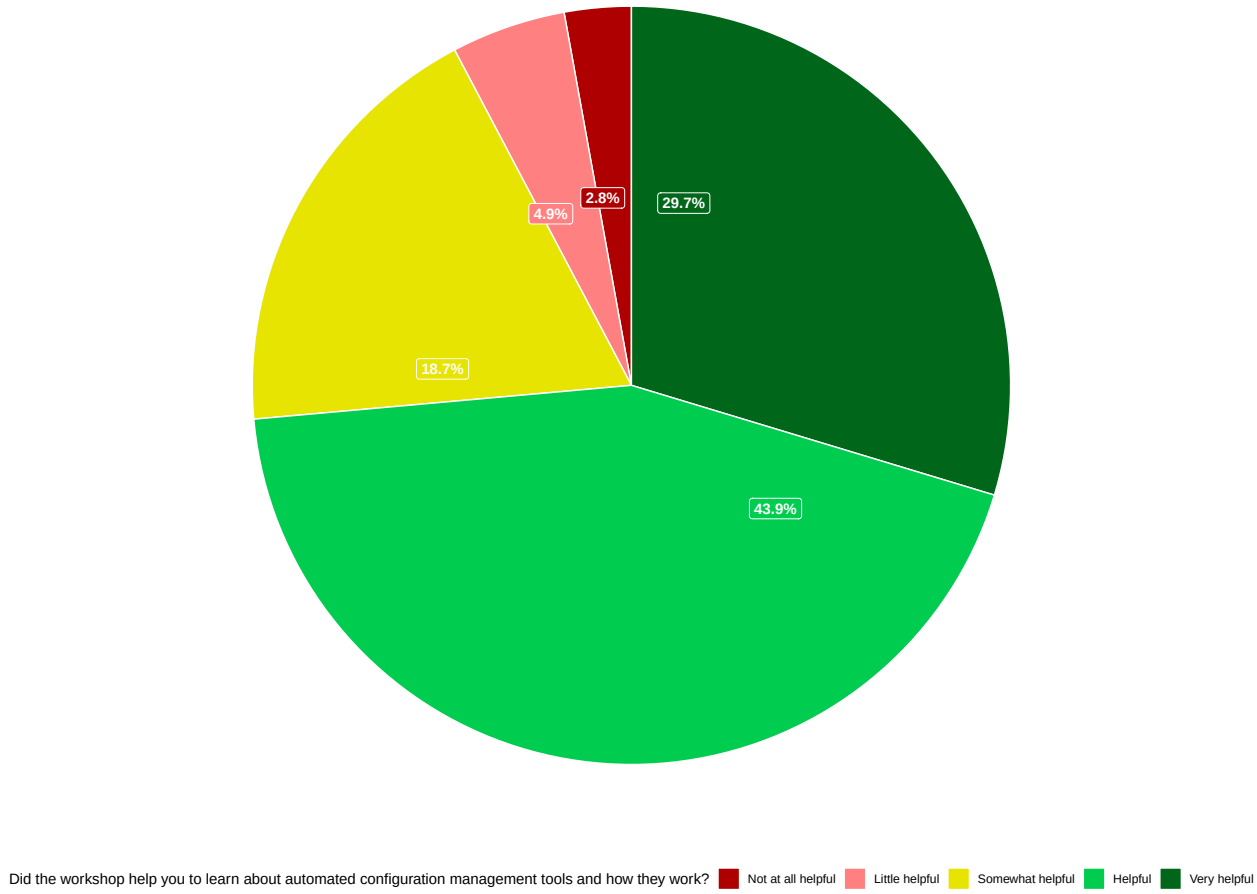

Figure 6.15: Overall Perception of Students on the Authentic Learning-based Exercise to Learn Automated Configuration Management Tools and How the Tools Work

We also discuss the overall perception among all the students in the class on how this authentic learning-based exercise helps them learn automated configuration management tools and how they work in Figure 6.15. We find that 29.7%, 43.9% , 18.7% of the students

report that they find the authentic learning-based exercise "Very helpful", "Helpful" and "Somewhat helpful" respectively. We find 4.9% and 2.8% students report the exercise is "Little helpful" and "Not at all helpful".

6.2.3 Answer to RQ 6.3

We answer RQ3 by reporting students' perceptions of instructor attributes in Figure 6.16. The students find instructor attributes to be beneficial for authentic learning-based exercises. From Figure 6.16, we observe that 84.2% of students find the instructor's industry background 'Extremely Useful' or 'Useful.' We also observe that on average, ¡1% students report that the instructor's background is 'Not useful at all'. Students find the instructor's academic background, industry experience, prior research background, enthusiasm, and in-person engagement to be useful for concept dissemination.

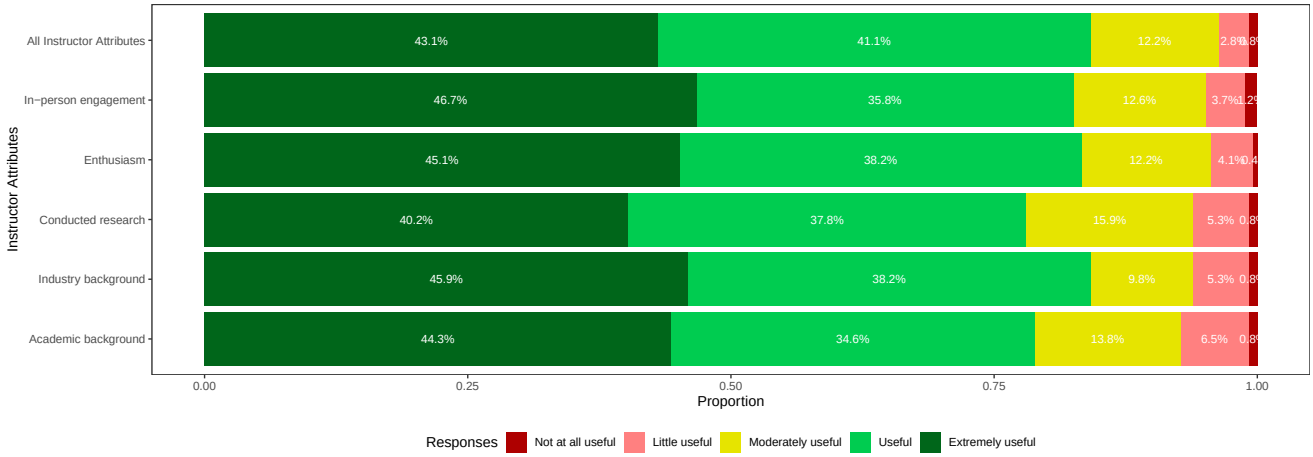

Figure 6.16: Reported Perception of Students of the Instructor Background for Authentic Learning-based Exercise

Chapter 7

Discussion

7.1 Implication for Practitioners

7.1.1 Application of Kubernetes Security Best Practices

Kubernetes provides utilities for users to manage containers at scale. However, our description of the 11 practices in Section 3.3 shows that effective and secure usage of Kubernetes requires the implementation of security practices applicable for multiple components within the Kubernetes installations: containers, pods, 'etcd' database etc. Applying the 11 practices mentioned above in Section 3.3 also needs a deep understanding of Kubernetes components and configurations. Our discussion in Section 3.3 can be helpful in two ways: *first*, understand the components where security practices are applicable. *Second*, practitioners who already have Kubernetes in place can use our identified practices as a benchmark and compare their usage of practices.

7.1.2 Application of Security Static Analysis

SLIKUBE+ extends SLIKUBE with 13 additional security misconfigurations. We recommend practitioners use our security static analysis tool SLIKUBE+ to perform regular scanning to avoid the propagation of misconfigurations.

7.1.3 Better Understanding of Pod-related Configuration Parameters

We verify the pod security requirements using the NuXMV model checker and identify security attacks from the counterexamples. The practitioner can use our research to have

a better understanding of the consequences of pod configuration parameters and how the parameters can pose a threat to the overall security posture of Kubernetes cluster.

7.2 Implication for Researchers

7.2.1 Baseline for Future Research

Our discussion in Section 2.2 shows that Kubernetes security to be an under-explored research area. Our derived list of security practices can provide the groundwork for Kubernetes security research. 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 opera- tors, control planes, and network planes.

7.2.2 Enhancing Security Analysis Tools

Researchers constructed security analysis tools that detect 11 pod-related configuration parameters [88]. Researchers conclude that each of the configuration parameter is important as they can cause security attack [88]. Moreover, existing security analysis tools such as KubeLinter [58], Checkov [14], Snyk [103] report one configuration parameter at a time. In our research we have demonstrated that a single configuration parameter can not be used to conduct a security attack and we have demonstrated in Table 5.16 that we need at least a combination 13 pod-related configuration parameters to conduct an security attack in Kubernetes. Researchers can incorporate attack-related information by emphasizing the fact that a single configuration parameter does not cause an attack. Without any context, a practitioner may ignore the detected misconfiguration as false alert and not take any action to fix. That is why encourage researchers and toolsmiths to improve security analysis tools for Kubernetes. In Table 5.13 and Table 5.14, we not only report the percentage of configuration parameters present in each manifest or repository rather we show the relevance

of pod configuration parameters in a manifest or in a repository to cause a security attack in Kubernetes.

7.2.3 Automated Framework for Identifying Pod-Related Configuration Parameters

We describe the existing challenges in constructing finite state machines for Kubernetes and discuss the possible ways for future researchers to address the challenges.

Advancing Context-Aware FSM Models

Our approach involves creating a finite state machine for a pod in Kubernetes by harnessing knowledge extracted from various Internet artifacts and the official Kubernetes documentation. As a result, the state of the finite state machine in our model is influenced by specific pod security requirements and assumptions of the model designer. In the past, researchers have extracted finite state machines from component interactions by instrumenting conformance tests to detect logical vulnerabilities in cellular network protocols [48].

Presently, Kubernetes requires approximately 380 conformance tests for all its distributions from various vendors, making them essential components of the system [61]. Researchers can now employ source code level instrumentation through annotations in Kubernetes conformance tests to generate information-rich logs. These logs can help identify the state of a Kubernetes object, corresponding actions, and the values of function parameters, thus enabling the detection of state transitions.

Utilizing instrumentation in conformance tests will help researchers in generate information-rich logs for all the conformance tests. From the logs, researchers can identify the interaction between the components in Kubernetes and build semantically meaningful models. Such models can play a crucial role in identifying misconfiguration-related vulnerabilities and logical vulnerabilities present in Kubernetes.

Extending Verification for Enhancing Completeness

In our research, we have developed a finite state model based on existing knowledge extracted from the official Kubernetes documentation and various Internet artifacts. Our model abstracts a running Kubernetes cluster in the context of its individual components. It represents the abstraction of the pod life cycle, its phases, the containers within each pod, and their states. Prior research has demonstrated that achieving soundness and completeness for parameterized verification problems is generally undecidable [9] [45].

In constructing the pod finite state model, we primarily focused on soundness rather than completeness. We have ensured that our FSM model is sound and does not generate false positives. Hence, whenever our model checker identifies a pod security requirement violation, it is indeed valid. However, our approach is not complete, as it cannot detect all possible violations. We have only extracted the necessary information from the documentation and internet artifacts to construct state machines. To improve the completeness of our approach, researchers can incorporate Kubernetes conformance testing to cover more interaction among the Kubernetes components. The coverage of tests will enhance the completeness of our approach, as it can detect more possible violations.

Utilizing Isolation Boundaries for Kubernetes Entities

Kubernetes is a complex software system with various isolation boundaries. The isolation boundary can be defined as the separation between the entities in a system environment that protects each entity from threats from other entities in the system. Isolation boundary can be specified as machine-level, process or component-level, and trust boundary level. For instance, a pod running with a misconfiguration `hostIPC: true` may put the other pods at a security risk if they run in the same worker node rather than a different worker node. If an attacker get an access to the misconfigured pod with remote code execution then the attack path to compromise the pods in the same worker node will be different than the pods running in the other working nodes in the cluster. In our research, we did not explicitly

define the isolation boundary to establish the threat model against a motivated attacker and extracting the attack context from the counterexamples.

To address this, future researchers can design the model to specify machine-level isolations, such as those for the scheduler, controller, and API server residing in control plane nodes, as well as Kubelet, Kube-proxy, and user-specified pods running in worker nodes. Additionally, researchers can include the trust boundary of components within their model design consideration. For example, a pod and the containers inside it are within the same trust boundary. By defining isolation boundaries, researchers can better identify the attack context from the counterexamples.

7.3 Implication for Educators

Our result in Section 6.2.1 suggests that students with prior background in security, software quality assurance and static analysis tools find the exercise helpful for them to learn about the Kubernetes security misconfigurations. From Figure 6.14 and 6.15, we observe that 90.6% and 92.3% of the students report that they learn Kubernetes security misconfiguration and how the automated configuration management tools work. Also, from Figure 6.13, we observe that 94% of the students perceive the steps of authentic learning-based exercise as 'Extremely Useful', 'Useful' and 'Somewhat Useful'. Based on the student's responses, the authentic learning-based exercise is very effective for learning about Kubernetes and misconfigurations. From Figure 6.16, we observe that the industry experience of the instructor is pivotal for concept dissemination as 84.2% students report it as 'Extremely Useful' or 'Useful.' However, we also observe students find other attributes such as 'Enthusiasm' and 'In-person Engagement' beneficial. Even if the instructor does not have adequate experience in industry or research background, they can conduct effective, authentic learning-based exercises by showing enthusiasm and engaging with the students in class.

7.4 Threats to Validity

In this Section, We describe the limitations of our research work.

7.4.1 Conclusion Validity

The identified security best practices described in Section 3.3 can be susceptible to biases of the rater who identified the practices by applying open coding. We mitigate this limitation by allocating another rater who applied closed coding. The 13 additional misconfiguration categories of SLIKUBE+ described in Section 5.5 may return false positives if it is evaluated on the proprietary dataset. Our construction of a finite state machine to abstract Kubernetes cluster is dependent on pod phases, container states, pod status and their relationships described in the Internet artifacts and Kubernetes documentation. Hence the transition relations may result in an unrealistic attack path with false positive counter-examples. We mitigated this limitation by validating each of the counterexamples with counterexample guided abstraction refinement(CEGAR) approach to derive realizable attack steps to conduct security attack. Our evaluation result in Section **??** of the survey consists of 127 members and the background of the students may create a bias. We mitigate this bias by deploying the authentic learning module into 3 courses in 2 universities in fall 2022 and spring 2023 semester.

7.4.2 Construct Validity

Our identified categories in Section 3.3 are susceptible to experimenter bias in which author's professional experience can influence the category results. To collect the pod properties, we systematically curating Internet artifacts to identify pod properties related to pod security requirements in Section 5.1.1. The list of pod properties in Table 5.1 can be susceptible to author's bias.

7.4.3 External Validity

Our identified security best practices in Section 3.3 might not be generalizable as we might have excluded practices unique to the proprietary domains, and not discussed publicly in Internet artifacts. The evaluation of SLIKUBE+ is limited to our dataset described in Table 5.10. The validation of attacks in the `kubeadm` cluster may not be generalizable and replicable in all other distribution of Kubernetes as the Kubernetes version and relevant components change very frequently. Results in Section 6.2.2, may not be generalizable as we did not survey the students who are enrolled in other courses where Kubernetes security is taught.

7.4.4 Internal Validity

We acknowledge that the our Internet artifact search process describe in Section 3.2 and Section 5.1 are not comprehensive. We also acknowledge that the limited number of manifests we consider for constructing SLIKUBE+ rules can impact the credibility of our tool.

Chapter 8

Conclusion

Kubernetes has become the go-to tool for implementing the practice of automated container orchestration. While Kubernetes has yielded benefits for IT organizations, security misconfigurations can make Kubernetes-based software deployments susceptible to security attacks. To help practitioners secure their Kubernetes cluster, systematization of knowledge related to practitioner-reported security best practices can help practitioners in securing Kubernetes installations. We conduct a qualitative analysis of 104 Internet artifacts, such as blog posts, to identify 11 security best practices for Kubernetes. To help practitioners understand the consequences of pod related security misconfiguration, we conduct a systematic investigation of pod-related configuration parameters that can cause security attacks in kubernetes pods. 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 combina- tions of 21 configuration parameters. We construct an authentic learning module for Kubernetes security misconfigurations and a survey to collect feedback. We deploy the authentic learning module into among 246 students across 4 semesters. We evaluate the feedback from the students and observe that 90.6% and 92.3% of the students report that they learn Kubernetes security misconfiguration and how the automated configuration management tools work. We also observe that 94% of the students perceive the steps of authentic learning-based exercise as 'Extremely Useful', 'Useful' and 'Somewhat Useful'. Students also report that industry experience of the instructor is pivotal for concept dissemination as 84.2% students report it as 'Extremely Useful' or 'Useful.'

We discuss the limitations of the dissertation in Section 7.4. Furthermore, we provide the implication for practitioners, opportunities for future researchers to build upon our work and implication for educators in Section 7.1, Section 7.2 and Section 7.3, respectively. Our work will help practitioners in adopting security best practices. Moreover, practitioners will understand how the misconfigurations in Kubernetes manifests can make the Kubernetes environment susceptible to security attacks. We expect our research will help the researchers further advance the science of secure Kubernetes manifest development.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies. We don't need another hero?: The impact of "heroes" on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 245–253, New York, NY, USA, 2018. ACM.

[3] akondrahman. akondrahman/sli-kube, 2022.

[4] M. S. Akter, H. Shahriar, D. Lo, N. Sakib, K. Qian, M. Whitman, and F. Wu. Authentic learning approach for artificial intelligence systems security and privacy. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1010–1012. IEEE, 2023.

[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

[6] S. K. Andrew Fletcher and H. Huijser. Authentic learning using mobile applications and contemporary geospatial information requirements related to environmental science. *Journal of Geography in Higher Education*, 46(2):185–203, 2022.

[7] Anonymous. Dataset for paper.

[8] V. Antinyan, M. Staron, and A. Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 22(6):3057–3087, 2017.

[9] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[10] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[11] S. Bobkov, A. Teslyuk, S. Zolotarev, M. Rose, K. Ikonnikova, V. Velikhov, I. Vartanyants, and V. Ilyin. Software platform for european xfel: Towards online experimental data analysis. *Lobachevskii Journal of Mathematics*, 39(9):1170–1178, 2018.

[12] D. B. Bose, A. Rahman, and M. S. I. Shamim. 'under-reported' security defects in kubernetes manifests. In *EnCyCriS 2021*. IEEE, 2021.

[13] 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)*, pages 9–12. IEEE, 2021.

[14] bridgecrew. checkov. `https://www.checkov.io/4.Integrations/Kubernetes.html`, 2022. [Online; accessed 12-May-2022].

[15] Canonical. Kubernetes and cloud native operations report 2021, 2021.

[16] 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*, pages 334–342. Springer, 2014.

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

[18] CNCF. With kubernetes, the u.s. department of defense is enabling devsecops on f-16s and battleships, 2020.

[19] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[21] B. F. Crabtree and W. L. Miller. *Doing qualitative research*. sage publications, 1999.

[22] Tesla cloud resources are hacked to run cryptocurrency-mining malware, February 2018.

[23] datree. datree. `https://hub.datree.io/built-in-rules#containers`, 2022. [Online; accessed 14-May-2022].

[24] dghubble. dghubble/go-twitter. `https://github.com/dghubble/go-twitter`, 2022. [Online; accessed 12-Jan-2022].

[25] Docker. Daemon socket option. `https://docs.docker.com/engine/reference/commandline/dockerd/`, 2022. [Online; accessed 19-Jan-2022].

[26] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[27] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23:452–489, 2018.

[28] R. Feldt, T. Zimmermann, G. R. Bergersen, D. Falessi, A. Jedlitschka, N. Juristo, J. Münch, M. Oivo, P. Runeson, M. Shepperd, et al. Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering*, 23:3801–3820, 2018.

[29] E. Friess. Scrum language use in a software engineering firm: An exploratory study. *IEEE Transactions on Professional Communication*, 62(2):130–147, 2019.

[30] V. Garousi, M. Felderer, and T. Hacaloğlu. Software test maturity assessment and test process improvement: A multivocal literature review. *Information and Software Technology*, 85:16 – 42, 2017.

[31] 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*, 106:101–121, 2019.

[32] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.

[33] J. W. Gentry. What is experiential learning. *Guide to business gaming and experiential learning*, 9:20, 1990.

[34] R. L. Glass. *Software Creativity 2.0*. developer.* Books, 2006.

[35] A. V. Goldberg and M. Kharitonov. *On implementing scaling push-relabel algorithms for the minimum-cost flow problem*. Department of Computer Science, Stanford University, 1992.

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

[37] J. Herrington. Introduction to authentic learning. In *Activity theory, authentic learning and emerging technologies*, pages 61–67. Routledge, 2015.

[38] J. Herrington, T. C. Reeves, and R. Oliver. *Authentic learning environments*. Springer, 2014.

[39] S. Hopewell, M. Clarke, and S. Mallett. Grey literature and systematic reviews. *Publication bias in meta-analysis: Prevention, assessment and adjustments*, pages 49–72, 2005.

[40] https://cloudnativenow.com. The Brutal Learning Curve of a New Kubernetes Cluster . `https://cloudnativenow.com/features/the-brutal-learning-curve-of-a-new-kubernetes-cluster/`, 2023. [Online; accessed 20-June-2023].

[41] https://microsoft.github.io/. Microsoft Threat Matrix. `https://microsoft.github.io/Threat-Matrix-for-Kubernetes/tactics/InitialAccess/`, 2023. [Online; accessed 28-April-2023].

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

[43] https://www.cncf.io. CNCF ANNUAL SURVEY 2021 . https://www.cncf.io/wp-content/uploads/2022/02/CNCF-Annual-Survey-2021.pdf, 2022. [Online; accessed 20-June-2023].

[44] https://www.forbes.com. Addressing The Kubernetes Skills Gap . https://www.forbes.com/sites/forbestechcouncil/2023/05/10/addressing-the-kubernetes-skills-gap/?sh=6f5bc84e23f4, 2023. [Online; accessed 20-June-2023].

[45] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[46] IBM. A tour of the kubernetes source code. https://developer.ibm.com/articles/a-tour-of-the-kubernetes-source-code/, 2024. [Online; accessed 29-May-2024].

[47] L. Ingvarson, M. Meiers, and A. Beavis. Factors affecting the impact of professional development programs on teachers' knowledge, practice, student outcomes & efficacy. 2005.

[48] I. Karim, S. R. Hussain, and E. Bertino. Prochecker: An automated security and privacy analysis framework for 4g lte protocol implementations. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 773–785. IEEE, 2021.

[49] M. Karjalainen and A.-L. Ojala. Authentic learning environments for in-service training in cybersecurity: a qualitative study. *International Journal of Continuing Engineering Education and Life Long Learning*, 33(1):128–147, 2023.

[50] M. Keller, K. Neumann, and H. E. Fischer. Teacher enthusiasm and student learning. In *International guide to student achievement*, pages 247–249. Routledge, 2013.

[51] J. Kindervag et al. Build security into your network's dna: The zero trust network architecture. *Forrester Research Inc*, pages 1–26, 2010.

[52] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[53] A. Y. Kolb and D. A. Kolb. Experiential learning theory as a guide for experiential educators in higher education. *Experiential Learning & Teaching in Higher Education*, 1(1):7–44, 2017.

[54] D. Kortepeter. U.S. lawmakers eye AWS role in Capital One data breach, 2019.

[55] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[56] K. Krippendorff and J. L. Fleiss. Reliability of binary attribute data, 1978.

[57] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies. What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 306–315, New York, NY, USA, 2018. ACM.

[58] kubelinter. kubelinter. `https://docs.kubelinter.io/#/generated/checks`, 2022. [Online; accessed 13-May-2022].

[59] Kubernetes. Production-grade container orchestration.

[60] Kubernetes User Case Studies, May 2020.

[61] Kubernetes. Kubernetes Conformance Tests . `https://github.com/kubernetes/kubernetes/blob/master/test/conformance/testdata/conformance.yaml`, 2024. [Online; accessed 26-March-2024].

[62] Kubernetes and cloud native. Cloud native usage report 2022.

[63] M. Kuhrmann, D. M. Fernández, and M. Daneva. On the pragmatic design of literature studies in software engineering: an experience-based guideline. *Empirical software engineering*, 22(6):2852–2891, 2017.

[64] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[65] Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[66] M. Lombard, J. Snyder-Duch, and C. C. Bracken. Practical resources for assessing and reporting intercoder reliability in content analysis research projects. 2010.

[67] M. M. Lombardi and D. G. Oblinger. Approaches that work: How authentic learning is transforming higher education. *EDUCAUSE Learning Initiative (ELI) Paper*, 5(2007), 2007.

[68] M. M. Lombardi and D. G. Oblinger. Authentic learning for the 21st century: An overview. *Educause learning initiative*, 1(2007):1–12, 2007.

[69] A. M. Love, J. A. Findley, L. A. Ruble, and J. H. McGrew. Teacher self-efficacy for teaching students with autism spectrum disorder: Associations with stress, teacher engagement, and student iep outcomes following compass consultation. *Focus on Autism and Other Developmental Disabilities*, 35(1):47–54, 2020.

[70] F. W. Maina. Authentic learning: Perspectives from contemporary educators. 2004.

[71] A. Martin and M. Hausenblas. *Hacking Kubernetes: Threat-Driven Analysis and Defense*. O'Reilly Media, 2021.

[72] M. McCarthy. Experiential learning theory: From theory to practice. *Journal of Business & Economics Research*, 14(3), 2016.

[73] 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.

[74] Mirantis. What are the primary reasons your organization is using Kubernetes?, 2021.

[75] Y. Morizumi, T. Hayashi, and Y. Ishida. A network visualization of stable matching in the stable marriage problem. *Artificial Life and Robotics*, 16(1):40–43, 2011.

[76] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, pages 1–35, 2017.

[77] NIST. misconfiguration, 2021.

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

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

[80] A. V. Oppenheim. *Discrete-time signal processing*. Pearson Education India, 1999.

[81] A. Ornellas, K. Falkner, and E. Edman Stålbrandt. Enhancing graduates' employability skills through authentic learning approaches. *Higher education, skills and work-based learning*, 9(1):107–120, 2019.

[82] G. J. Palardy and R. W. Rumberger. Teacher effectiveness in first grade: The importance of background qualifications, attitudes, and instructional practices for student learning. *Educational evaluation and policy analysis*, 30(2):111–140, 2008.

[83] K. Qian, D. Lo, R. Parizi, F. Wu, E. Agu, and B.-T. Chu. Authentic learning secure software development (ssd) in computing education. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2018.

[84] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran. Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, SWAN 2018, pages 8–14, New York, NY, USA, 2018. ACM.

[85] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019.

[86] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), Jan. 2021.

[87] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1):1–31, 2021.

[88] 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.*, dec 2022. Just Accepted.

[89] A. Rahman, S. I. Shamim, H. Shahriar, and F. Wu. Can we use authentic learning to educate students about secure infrastructure as code development? In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*, pages 631–631, 2022.

[90] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Proceedings of the 2015 Agile Conference*, AGILE '15, page 1–10, USA, 2015. IEEE Computer Society.

[91] P. Raulamo-Jurvanen, S. Hosio, and M. V. Mäntylä. Practitioner evaluations on software testing tools. In *Proceedings of the Evaluation and Assessment on Software Engineering*, EASE '19, page 57–66, New York, NY, USA, 2019. Association for Computing Machinery.

[92] RedHat. Kubernetes adoption, security, and market trends report, 2021.

[93] RedHat. State of Kubernetes Security Report 2021, 2021.

[94] RedHat. State of Kubernetes Security Report 2022, 2021.

[95] RedHat. State of Kubernetes Security Report 2024, 2024.

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

[97] D. G. Rees Lewis, E. M. Gerber, S. E. Carlson, and M. W. Easterday. Opportunities for educational innovations in authentic project-based learning: understanding instructor perceived challenges to design for adoption. *Educational technology research and development*, 67:953–982, 2019.

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

[99] J. Saldana. *The Coding Manual for Qualitative Researchers*. SAGE, 2015.

[100] 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].

[101] 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)*, pages 58–64, Los Alamitos, CA, USA, sep 2020. IEEE Computer Society.

[102] Y. Shi et al. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 81–86. IEEE, 2001.

[103] Snyk. snyk. `https://snyk.io/security-rules/kubernetes/`, 2022. [Online; accessed 15-May-2022].

[104] Stackrox. Kubernetes and container security and adoption trends, 2021.

[105] stefanprodan. stefanprodan/podinfo. `https://github.com/stefanprodan/podinfo`, 2022. [Online; accessed 12-Jan-2022].

[106] T4. Container Platform Market Share, Market Size and Industry Growth Drivers, 2018 - 2023, 2020.

[107] T. Taylor. 5 Kubernetes security incidents and what we can learn from them, 2020.

[108] I. Turner-Trauring. "let's use kubernetes!" now you have 8 problems. `https://pythonspeed.com/articles/dont-need-kubernetes/`, 2024. [Online; accessed 29-May-2024].

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

[110] A. A. Ur Rahman and L. Williams. Software security in devops: Synthesizing practitioners' perceptions and practices. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, CSED '16, pages 70–76, New York, NY, USA, 2016. ACM.

[111] VMWare. What is a Kubernetes Deployment? `https://www.vmware.com/topics/glossary/content/kubernetes-deployment`, 2021. [Online; accessed 01-Nov-2021].

[112] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.

Appendix A

Appendix

Table A1: List of 105 Publications for Literature Review

Index	**Publication**
P1	Medel, Víctor, Omer Rana, José Ángel Bañares, and Unai Arronategui. "Modelling performance & resource management in kubernetes." In Proceedings of the 9th International Conference on Utility and Cloud Computing, pp. 257-262. 2016.
P2	Takahashi, Kimitoshi, Kento Aida, Tomoya Tanjo, Jingtao Sun, and Kazushige Saga. "A Portable Load Balancer with ECMP Redundancy for Container Clusters." IEICE TRANSACTIONS on Information and Systems 102, no. 5 (2019): 974-987.
P3	Hariri, Sahand, and Matias Carrasco Kind. "Batch and online anomaly detection for scientific applications in a Kubernetes environment." In Proceedings of the 9th Workshop on Scientific Cloud Computing, pp. 1-7. 2018.
P4	Sarajlic, Semir, Julien Chastang, Suresh Marru, Jeremy Fischer, and Mike Lowe. "Scaling JupyterHub using Kubernetes on Jetstream cloud: Platform as a service for research and educational initiatives in the atmospheric sciences." In Proceedings of the Practice and Experience on Advanced Research Computing, pp. 1-4. 2018.
P5	Li, Qiankun, Gang Yin, Tao Wang, and Yue Yu. "Building a Cloud-Ready Program: A highly scalable Implementation based on Kubernetes." In Proceedings of the 2nd International Conference on Advances in Image Processing, pp. 159-164. 2018.
	Continued on next page

Table A1 – continued from previous page

Index	Publication
P6	Xu, Cong, Karthick Rajamani, and Wesley Felter. "Nbwguard: Realizing network qos for kubernetes." In Proceedings of the 19th International Middleware Conference Industry, pp. 32-38. 2018.
P7	Liu, Haifeng, Shugang Chen, Yongcheng Bao, Wanli Yang, Yuan Chen, Wei Ding, and Huasong Shan. "A High Performance, Scalable DNS Service for Very Large Scale Container Cloud Platforms." In Proceedings of the 19th International Middleware Conference Industry, pp. 39-45. 2018.
P8	Wei-guo, Zhang, Ma Xi-lin, and Zhang Jin-zhong. "Research on Kubernetes' Resource Scheduling Scheme." In Proceedings of the 8th International Conference on Communication and Network Security, pp. 144-148. 2018.
P9	Zhuang, Jinfeng, and Yu Liu. "PinText: A Multitask Text Embedding System in Pinterest." In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2653-2661. 2019.
P10	Tu, Tengfei, Xiaoyu Liu, Linhai Song, and Yiying Zhang. "Understanding real-world concurrency bugs in Go." In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 865-878. 2019.
P11	Govind, Yash, Pradap Konda, Paul Suganthan GC, Philip Martinkus, Palaniappan Nagarajan, Han Li, Aravind Soundararajan et al. "Entity matching meets data science: A progress report from the magellan project." In Proceedings of the 2019 International Conference on Management of Data, pp. 389-403. 2019.
P12	Patel, Jemish, Goutam Tadi, Oz Basarir, Lawrence Hamel, David Sharp, Fei Yang, and Xin Zhang. "Pivotal Greenplum© for Kubernetes: Demonstration of Managing Greenplum Database on Kubernetes." In Proceedings of the 2019 International Conference on Management of Data, pp. 1969-1972. 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P13	Carcassi, Gabriele, Joe Breen, Lincoln Bryant, Robert W. Gardner, Shawn Mckee, and Christopher Weaver. "SLATE: Monitoring Distributed Kubernetes Clusters." In Practice and Experience in Advanced Research Computing, pp. 19-25. 2020.
P14	Huang, Yuzhou, Kaiyu cai, Ran Zong, and Yugang Mao. "Design and implementation of an edge computing platform architecture using docker and kubernetes for machine learning." In Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications, pp. 29-32. 2019.
P15	Kouchaksaraei, Hadi Razzaghi, and Holger Karl. "Service function chaining across openstack and kubernetes domains." In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pp. 240-243. 2019.
P16	Thurgood, Brandon, and Ruth G. Lennon. "Cloud computing with Kubernetes cluster elastic scaling." In Proceedings of the 3rd International Conference on Future Networks and Distributed Systems, pp. 1-7. 2019.
P17	Ambati, Pradeep, and David Irwin. "Optimizing the cost of executing mixed interactive and batch workloads on transient vms." Proceedings of the ACM on Measurement and Analysis of Computing Systems 3, no. 2 (2019): 1-24.
P18	Le, Tan N., Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. "AlloX: compute allocation in hybrid clusters." In Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1-16. 2020.
P19	Liu, Yang, Huanle Xu, and Wing Cheong Lau. "Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications." In Proceedings of the ACM Symposium on Cloud Computing, pp. 479-479. 2019.
P20	Xu, Charles, and Dmitry Ilyevskiy. "Isopod: An Expressive DSL for Kubernetes Configuration." In Proceedings of the ACM Symposium on Cloud Computing, pp. 483-483. 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P21	Kaminski, Matthijs, Eddy Truyen, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. "A framework for black-box SLO tuning of multi-tenant applications in Kubernetes." In Proceedings of the 5th International Workshop on Container Technologies and Container Clouds, pp. 7-12. 2019.
P22	Verreydt, Stef, Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, and Wouter Joosen. "Leveraging Kubernetes for adaptive and cost-efficient resource management." In Proceedings of the 5th International Workshop on Container Technologies and Container Clouds, pp. 37-42. 2019.
P23	Yeh, Ting-An, Hung-Hsin Chen, and Jerry Chou. "KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud." In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, pp. 173-184. 2020.
P24	Zhong, Zhiheng, and Rajkumar Buyya. "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources." ACM Transactions on Internet Technology (TOIT) 20, no. 2 (2020): 1-24.
P25	Lee, Chun-Hsiang, Zhaofeng Li, Xu Lu, Tiyun Chen, Saisai Yang, and Chao Wu. "Multi-Tenant Machine Learning Platform Based on Kubernetes." In Proceedings of the 2020 6th International Conference on Computing and Artificial Intelligence, pp. 5-12. 2020.
P26	Alimudin, Akhmad, and Yoshiteru Ishida. "Service-Based Container Deployment on Kubernetes Using Stable Marriage Problem." In Proceedings of the 2020 The 6th International Conference on Frontiers of Educational Technologies, pp. 164-167. 2020.
P27	Li, Dong, Yi Wei, and Bing Zeng. "A Dynamic I/O Sensing Scheduling Scheme in Kubernetes." In Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications, pp. 14-19. 2020.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P28	Fan, Dayong, and Dongzhi He. "A Scheduler for Serverless Framework base on Kubernetes." In Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence, pp. 229-232. 2020.
P29	Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade." Queue 14, no. 1 (2016): 70-93.
P30	Singh, Satnam. "Cluster-level Logging of Containers with Containers: Logging Challenges of Container-Based Cloud Deployments." Queue 14, no. 3 (2016): 83-106.
P31	Bernstein, David. "Containers and cloud: From lxc to docker to kubernetes." IEEE Cloud Computing 1, no. 3 (2014): 81-84.
P32	Medel, Víctor, Omer Rana, José Ángel Bañares, and Unai Arronategui. "Adaptive application scheduling under interference in kubernetes." In 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), pp. 426-427. IEEE, 2016.
P33	Bila, Nilton, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. "Leveraging the serverless architecture for securing linux containers." In 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 401-404. IEEE, 2017.
P34	Dupont, Corentin, Raffaele Giaffreda, and Luca Capra. "Edge computing in IoT context: Horizontal and vertical Linux container migration." In 2017 Global Internet of Things Summit (GIoTS), pp. 1-4. IEEE, 2017.
P35	Tsai, Pei-Hsuan, Hua-Jun Hong, An-Chieh Cheng, and Cheng-Hsin Hsu. "Distributed analytics in fog computing platforms using tensorflow and kubernetes." In 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), pp. 145-150. IEEE, 2017.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P36	Sima, Vasile, Alexandru Stanciu, and Florin Hartescu. "New software applications for system identification." In 2017 21st International Conference on System Theory, Control and Computing (ICSTCC), pp. 106-111. IEEE, 2017.
P37	Coullon, Hélène, Christian Perez, and Dimitri Pertin. "Production deployment tools for IaaSes: an overall model and survey." In 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 183-190. IEEE, 2017.
P38	Javed, Asad, Keijo Heljanko, Andrea Buda, and Kary Främling. "Cefiot: A fault-tolerant iot architecture for edge and cloud." In 2018 IEEE 4th world forum on internet of things (WF-IoT), pp. 813-818. IEEE, 2018.
P39	Tosh, Deepak, Sachin Shetty, Peter Foytik, Charles Kamhoua, and Laurent Njilla. "CloudPoS: A proof-of-stake consensus design for blockchain integrated cloud." In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 302-309. IEEE, 2018.
P40	Herger, Lorraine M., Mercy Bodarky, and Carlos Fonseca. "Breaking down the barriers for moving an enterprise to cloud." In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 572-576. IEEE, 2018.
P41	Podolskiy, Vladimir, Anshul Jindal, and Michael Gerndt. "Iaas reactive autoscaling performance challenges." In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 954-957. IEEE, 2018.
P42	Vayghan, Leila Abdollahi, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. "Deploying microservice based applications with Kubernetes: experiments and lessons learned." In 2018 IEEE 11th international conference on cloud computing (CLOUD), pp. 970-973. IEEE, 2018.
P43	Modak, Arsh, S. D. Chaudhary, P. S. Paygude, and S. R. Ldate. "Techniques to secure data on cloud: Docker swarm or kubernetes?." In 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), pp. 7-12. IEEE, 2018.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P44	Netto, Hylson Vescovi, Aldelir Fernando Luiz, Miguel Correia, Luciana de Oliveira Rech, and Caio Pereira Oliveira. "Koordinator: A service approach for replicating Docker containers in Kubernetes." In 2018 IEEE Symposium on Computers and Communications (ISCC), pp. 00058-00063. IEEE, 2018.
P45	Xiong, Ying, Yulin Sun, Li Xing, and Ying Huang. "Extend cloud to edge with kubeedge." In 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 373-377. IEEE, 2018.
P46	Aly, Mohab, Foutse Khomh, and Soumaya Yacout. "Kubernetes or openShift? Which technology best suits eclipse hono IoT deployments." In 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), pp. 113-120. IEEE, 2018..
P47	Brito, Andrey, Christof Fetzer, Stefan Köpsell, Marcelo Pasin, Pascal Felber, Keiko Fonseca, Marcelo Rosa et al. "Cloud challenge: Secure end-to-end processing of smart metering data." In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 36-42. IEEE, 2018.
P48	Shah, Jay, and Dushyant Dubaria. "Building modern clouds: using docker, kubernetes & Google cloud platform." In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0184-0189. IEEE, 2019.
P49	Rehman, Kasim, Orthodoxos Kipouridis, Stamatis Karnouskos, Oliver Frendo, Helge Dickel, Jonas Lipps, and Nemrude Verzano. "A cloud-based development environment using hla and kubernetes for the co-simulation of a corporate electric vehicle fleet." In 2019 IEEE/SICE International Symposium on System Integration (SII), pp. 47-54. IEEE, 2019.
P50	Townend, Paul, Stephen Clement, Dan Burdett, Renyu Yang, Joe Shaw, Brad Slater, and Jie Xu. "Improving data center efficiency through holistic scheduling in kubernetes." In 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 156-15610. IEEE, 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P51	Bao, Yixin, Yanghua Peng, and Chuan Wu. "Deep learning-based job placement in distributed machine learning clusters." In IEEE INFOCOM 2019-IEEE conference on computer communications, pp. 505-513. IEEE, 2019.
P52	Podolskiy, Vladimir, Michael Mayo, Abigail Koay, Michael Gerndt, and Panos Patros. "Maintaining SLOs of cloud-native applications via self-adaptive resource sharing." In 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 72-81. IEEE, 2019.
P53	Chiba, Tatsuhiro, Rina Nakazawa, Hiroshi Horii, Sahil Suneja, and Seetharami Seelam. "Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes." In 2019 IEEE International Conference on Cloud Engineering (IC2E), pp. 168-178. IEEE, 2019.
P54	Santos, Jose, Tim Wauters, Bruno Volckaert, and Filip De Turck. "Towards network-aware resource provisioning in Kubernetes for fog computing applications." In 2019 IEEE Conference on Network Softwarization (NetSoft), pp. 351-359. IEEE, 2019.
P55	Rattihalli, Gourav, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 33-40. IEEE, 2019.
P56	Gawel, Maciej, and Krzysztof Zielinski. "Analysis and evaluation of kubernetes based nfv management and orchestration." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 511-513. IEEE, 2019.
P57	Kaur, Kuljeet, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. "Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem." IEEE Internet of Things Journal 7, no. 5 (2019): 4228-4237.
P58	Kelley, Jaimie, and Nathaniel Morris. "Rapid In-situ Profiling of Colocated Workloads." In IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 528-534. IEEE, 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P59	Vayghan, Leila Abdollahi, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. "Microservice based architecture: Towards high-availability for stateful applications with Kubernetes." In 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), pp. 176-185. IEEE, 2019.
P60	Astyrakakis, Nikolaos, Yannis Nikoloudakis, Ioannis Kefaloukos, Charalabos Skianis, Evangelos Pallis, and Evangelos K. Markakis. "Cloud-Native Application Validation & Stress Testing through a Framework for Auto-Cluster Deployment." In 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), pp. 1-5. IEEE, 2019.
P61	Marathe, Nikhil, Ankita Gandhi, and Jaimeel M. Shah. "Docker swarm and kubernetes in cloud computing environment." In 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), pp. 179-184. IEEE, 2019.
P62	Casquero, Oskar, Aintzane Armentia, Isabel Sarachaga, Federico Pérez, Darío Orive, and Marga Marcos. "Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications." In 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1213-1217. IEEE, 2019.
P63	Chen, Hung-Li, and Fuchun Joseph Lin. "Scalable IoT/M2M platforms based on kubernetes-enabled NFV MANO architecture." In 2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 1106-1111. IEEE, 2019.
P64	Link, Coleman, Jesse Sarran, Garegin Grigoryan, Minseok Kwon, M. Mustafa Rafique, and Warren R. Carithers. "Container Orchestration by Kubernetes for RDMA Networking." In 2019 IEEE 27th International Conference on Network Protocols (ICNP), pp. 1-2. IEEE, 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P65	Bringhenti, Daniele, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. "Towards a fully automated and optimized network security functions orchestration." In 2019 4th International Conference on Computing, Communications and Security (ICCCS), pp. 1-7. IEEE, 2019.
P66	Hussain, Fatima, Weiyue Li, Brett Noye, Salah Sharieh, and Alexander Ferworn. "Intelligent Service Mesh Framework for API Security and Management." In 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 0735-0742. IEEE, 2019.
P67	Pan, Yao, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, and Richard Sinnott. "A performance comparison of cloud-based container orchestration tools." In 2019 IEEE International Conference on Big Knowledge (ICBK), pp. 191-198. IEEE, 2019.
P68	Beltre, Angel M., Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. "Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms." In 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 11-20. IEEE, 2019.
P69	Ferreira, Arnaldo Pereira, and Richard Sinnott. "A performance evaluation of containers running on managed kubernetes services." In 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 199-208. IEEE Computer Society, 2019.
P70	Wu, Qiang, Jiadi Yu, Li Lu, Shiyou Qian, and Guangtao Xue. "Dynamically adjusting scale of a kubernetes cluster under QoS guarantee." In 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS), pp. 193-200. IEEE, 2019.
P71	Tesliuk, Anton, Sergey Bobkov, Viacheslav Ilyin, Alexander Novikov, Alexey Poyda, and Vasily Velikhov. "Kubernetes container orchestration as a framework for flexible and effective scientific data analysis." In 2019 Ivannikov Ispras Open Conference (ISPRAS), pp. 67-71. IEEE, 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P72	De Iasio, Antonio, and Eugenio Zimeo. "Avoiding Faults due to Dangling Dependencies by Synchronization in Microservices Applications." In 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 169-176. IEEE, 2019.
P73	Fu, Yuqi, Shaolun Zhang, Jose Terrero, Ying Mao, Guangya Liu, Sheng Li, and Dingwen Tao. "Progress-based container scheduling for short-lived applications in a kubernetes cluster." In 2019 IEEE International Conference on Big Data (Big Data), pp. 278-287. IEEE, 2019.
P74	Rajavaram, Harika, Vineet Rajula, and B. Thangaraju. "Automation of Microservices Application Deployment Made Easy By Rundeck and Kubernetes." In 2019 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), pp. 1-3. IEEE, 2019.
P75	Dewi, Lily Puspa, Agustinus Noertjahyana, Henry Novianus Palit, and Kezia Yedutun. "Server Scalability Using Kubernetes." In 2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON), pp. 1-4. IEEE, 2019.
P76	Schneider, Stefan, Manuel Peuster, Kai Hannemann, Daniel Behnke, Marcel Muller, Patrick-Benjamin Bök, and Holger Karl. ""Producing Cloud-Native": Smart Manufacturing Use Cases on Kubernetes." In 2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pp. 1-2. IEEE, 2019.
P77	Beltre, Angel, Pankaj Saha, and Madhusudhan Govindaraju. "Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters." In 2019 IEEE Cloud Summit, pp. 14-20. IEEE, 2019.
P78	Wang, Mingming, Dongmei Zhang, and Bin Wu. "A Cluster Autoscaler Based on Multiple Node Types in Kubernetes." In 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), vol. 1, pp. 575-579. IEEE, 2020.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P79	Surya, Rahmad Yesa, and Achmad Imam Kistijantoro. "Dynamic Resource Allocation for Distributed TensorFlow Training in Kubernetes Cluster." In 2019 International Conference on Data and Software Engineering (ICoDSE), pp. 1-6. IEEE, 2019.
P80	Huang, Jiaming, Chuming Xiao, and Weigang Wu. "RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning." In 2020 IEEE International Conference on Cloud Engineering (IC2E), pp. 116-123. IEEE, 2020.
P81	Balla, David, Csaba Simon, and Markosz Maliosz. "Adaptive scaling of Kubernetes pods." In NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, pp. 1-5. IEEE, 2020.
P82	Liu, Qingyang, E. Haihong, and Meina Song. "The Design of Multi-Metric Load Balancer for Kubernetes." In 2020 International Conference on Inventive Computation Technologies (ICICT), pp. 1114-1117. IEEE, 2020.
P83	Donca, Ionut-Catalin, Cosmina Corches, Ovidiu Stan, and Liviu Miclea. "Autoscaled RabbitMQ Kubernetes Cluster on single-board computers." In 2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1-6. IEEE, 2020.
P84	Donca, Ionut-Catalin, Cosmina Corches, Ovidiu Stan, and Liviu Miclea. "Autoscaled RabbitMQ Kubernetes Cluster on single-board computers." In 2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1-6. IEEE, 2020.
P85	Botez, Robert, Calin-Marian Iurian, Iustin-Alexandru Ivanciu, and Virgil Dobrota. "Deploying a Dockerized Application With Kubernetes on Google Cloud Platform." In 2020 13th International Conference on Communications (COMM), pp. 471-476. IEEE, 2020.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P86	Eidenbenz, Raphael, Yvonne-Anne Pignolet, and Alain Ryser. "Latency-Aware Industrial Fog Application Orchestration with Kubernetes." In 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), pp. 164-171. IEEE, 2020.
P87	Qi, Shixiong, Sameer G. Kulkarni, and K. K. Ramakrishnan. "Understanding container network interface plugins: design considerations and performance." In 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN, pp. 1-6. IEEE, 2020.
P88	Nguyen, Nguyen, and Taehong Kim. "Toward Highly Scalable Load Balancing in Kubernetes Clusters." IEEE Communications Magazine 58, no. 7 (2020): 78-83.
P89	Muddinagiri, Ruchika, Shubham Ambavane, and Simran Bayas. "Self-Hosted Kubernetes: Deploying Docker Containers Locally With Minikube." In 2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET), pp. 239-243. IEEE, 2019.
P90	Rossi, Fabiana, Valeria Cardellini, and Francesco Lo Presti. "Hierarchical scaling of microservices in Kubernetes." In 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 28-37. IEEE, 2020.
P91	Guerrero, Carlos, Isaac Lera, and Carlos Juiz. "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture." Journal of Grid Computing 16, no. 1 (2018): 113-135.
P92	Ahmadvand, Mohsen, Alexander Pretschner, Keith Ball, and Daniel Eyring. "Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework." In Federation of International Conferences on Software Technologies: Applications and Foundations, pp. 573-588. Springer, Cham, 2018.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P93	Tien, Chin-Wei, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches." Engineering Reports 1, no. 5 (2019): e12080.
P94	Bogo, Matteo, Jacopo Soldani, Davide Neri, and Antonio Brogi. "Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes." Software: Practice and Experience 50, no. 9 (2020): 1793-1821.
P95	Medel, Víctor, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F. Rana. "Characterising resource management performance in Kubernetes." Computers & Electrical Engineering 68 (2018): 286-297.
P96	Diouf, Gor Mack, Halima Elbiaze, and Wael Jaafar. "On Byzantine fault tolerance in multi-master Kubernetes clusters." Future Generation Computer Systems 109 (2020): 407-419.
P97	Netto, Hylson V., Lau Cheuk Lung, Miguel Correia, Aldelir Fernando Luiz, and Luciana Moreira Sá de Souza. "State machine replication in containers managed by Kubernetes." Journal of Systems Architecture 73 (2017): 53-59.
P98	Araya, Mauricio, Maximiliano Osorio, Matías Díaz, Carlos Ponce, Martín Villanueva, Camilo Valenzuela, and Mauricio Solar. "JOVIAL: Notebook-based astronomical data analysis in the cloud." Astronomy and computing 25 (2018): 110-117.
P99	Christodoulopoulos, Christos, and Euripides GM Petrakis. "Commodore: fail safe container scheduling in kubernetes." In International Conference on Advanced Information Networking and Applications, pp. 988-999. Springer, Cham, 2019.

Continued on next page

Table A1 – continued from previous page

Index	Publication
P100	Surantha, Nico, and Felix Ivan. "Secure kubernetes networking design based on zero trust model: A case study of financial service enterprise in indonesia." In International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pp. 348-361. Springer, Cham, 2019.
P101	Mercl, Lubos, and Jakub Pavlik. "Public Cloud Kubernetes Storage Performance Analysis." In International Conference on Computational Collective Intelligence, pp. 649-660. Springer, Cham, 2019.
P102	Goethals, Tom, Filip De Turck, and Bruno Volckaert. "Fledge: Kubernetes compatible container orchestration on low-resource edge devices." In International Conference on Internet of Vehicles, pp. 174-189. Springer, Cham, 2019.
P103	Kratzke, Nane. "About the complexity to transfer cloud applications at runtime and how container platforms can contribute?." In International Conference on Cloud Computing and Services Science, pp. 19-45. Springer, Cham, 2017.
P104	Zheng, Wei-Sheng, and Li-Hsing Yen. "Auto-scaling in Kubernetes-based fog computing platform." In International Computer Symposium, pp. 338-345. Springer, Singapore, 2018.
P105	Hu, Yang, Huan Zhou, Cees de Laat, and Zhiming Zhao. "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints." Future Generation Computer Systems 102 (2020): 562-573.