VITOR ORIEL DE CASTRO NUNES BORGES

SMELLY KUBE: A TOOL TO DETECT SECURITY SMELLS IN KUBERNETES MANIFESTS

LAVRAS – MG

2025

VITOR ORIEL DE CASTRO NUNES BORGES

SMELLY KUBE: A TOOL TO DETECT SECURITY SMELLS IN KUBERNETES MANIFESTS

Dissertation presented to the Federal University of Lavras, as part of the requirements of the Post Graduate Program in Computer Science, area of concentration in Computer Networks and Embedded Systems, for obtaining the Master's degree.

Prof. DSc. Luiz Henrique Andrade Correia
Advisor

LAVRAS – MG
2025

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos da Biblioteca Universitária da UFLA

Borges, Vitor Oriel de Castro Nunes

Smelly Kube : A tool to detect security smells in Kubernetes manifests / Vitor Oriel de Castro Nunes Borges - 2025.

86 p. il.

Dissertação (Mestrado acadêmico)–Universidade Federal de Lavras, 2024.

Bibliografia.

1. Kubernetes. 2. Security smells. 3. Kubernetes manifests. I. Correia, Luiz Henrique Andrade. II. Tìtulo.

VITOR ORIEL DE CASTRO NUNES BORGES

SMELLY KUBE: A TOOL TO DETECT SECURITY SMELLS IN KUBERNETES MANIFESTS
SMELL KUBE: UMA FERRAMENTA PARA DETECTAR MALCHEIROS DE SEGURANÇA EM MANIFESTOS KUBERNETES

Dissertation presented to the Federal University of Lavras, as part of the requirements of the Post Graduate Program in Computer Science, area of concentration in Computer Networks and Embedded Systems, for obtaining the Master's degree.

APPROVED in December 16, 2024.

Prof. DSc. Luiz Henrique Andrade Correia	UFLA
Prof. DSc. Rafael Serapilha Durelli	UFLA
Prof. DSc. Neumar Costa Malheiros	UFLA
Prof. DSc. Hudson Silva Borges	UFMS

Prof. DSc. Luiz Henrique Andrade Correia
Advisor

LAVRAS – MG
2025

I dedicate this work to my grandfather Oriel Nunes Borges and my grandmother Maria Helena de Castro Nunes Borges.

ACKNOWLEDGEMENTS

I thank Professor Rafael Serapilha Durelli for helping with the guidance and validation of the work, Professor Ivo Augusto Andrade Rocha Calado for assisting with the collection of the Artifact Hub dataset, and Professor Hudson Silva Borges for providing the GitHub dataset.

ABSTRACT

The adoption of microservices-based architectures has become common in modern web applications. Kubernetes stands out as a leading platform for managing and orchestrating these services due to its scalability and ability to handle complex environments. However, security risks often arise when Kubernetes manifests are not carefully designed. To address this issue, the tool Smelly Kube (SK) was developed, following a client/server architecture: *i*) Client: a Visual Studio Code plugin that sends Kubernetes manifests to a Golang-based server via HTTP requests, providing developers with a straightforward interface for performing security checks directly in their development environment, and *ii*) Server: a Golang application that processes these manifests to detect and analyze security smells. This architecture ensures that developers can easily integrate security checks into their workflows using other tools and platforms by reusing the same server API. To validate the tool's effectiveness, an experiment was conducted, applying SK to 2,107 Kubernetes applications after sanitizing 5,055 Cloud Native packages sourced from Artifact Hub. Another dataset of 183,225 Kubernetes manifests was provided from GitHub. The results demonstrated that SK successfully detected a wide range of security smells, providing valuable insights for developers and helping to improve the overall security of Kubernetes-based microservices.

Keywords: security; security smells; misconfiguration; Kubernetes; Kubernetes manifests.

RESUMO

A adoção de arquiteturas baseadas em microserviços tornou-se comum em aplicações web modernas. O Kubernetes se destaca como uma plataforma líder para gerenciar e orquestrar esses serviços devido à sua escalabilidade e capacidade de lidar com ambientes complexos. No entanto, riscos de segurança frequentemente surgem quando os manifestos do Kubernetes não são cuidadosamente projetados. Para resolver essa questão, a ferramenta Smelly Kube (SK) foi desenvolvida, seguindo uma arquitetura cliente/servidor: *i*) Cliente: um plugin do Visual Studio Code que envia manifestos do Kubernetes para um servidor baseado em Golang via requisições HTTP, oferecendo aos desenvolvedores uma interface simples para realizar verificações de segurança diretamente em seu ambiente de desenvolvimento, e *ii*) Servidor: uma aplicação Golang que processa esses manifestos para detectar e analisar malcheiros de segurança. Essa arquitetura garante que os desenvolvedores possam integrar facilmente as verificações de segurança em seus fluxos de trabalho, reutilizando a mesma API do servidor com outros ferramentas e plataformas. Para validar a eficácia da ferramenta, foi realizado um experimento, aplicando o SK em 2.107 aplicações Kubernetes após a sanitização de 5.055 pacotes Cloud Native provenientes do Artifact Hub. Outro conjunto de dados, contendo 183.225 manifestos Kubernetes, foi fornecido do GitHub. Os resultados demonstraram que o SK detectou com sucesso uma ampla gama de malcheiros segurança, fornecendo insights valiosos para os desenvolvedores e ajudando a melhorar a segurança geral dos microserviços baseados em Kubernetes.

Palavras-chave: segurança; malcheiros de segurança; falhas de configuração; Kubernetes; manifestos Kubernetes.

SOCIAL, TECHNOLOGICAL, ECONOMIC AND CULTURAL IMPACTS

The objective of this work was to strengthen the security of Kubernetes-based environments through the identification of security smells in configuration files, by proposing and evaluating the Smelly Kube (SK) tool, developed using a client-server architecture. The client, a plugin for Visual Studio Code, allows developers to perform security checks directly within their development environment; the server, implemented in Golang, processes manifests received via HTTP requests and performs the detection of security smells. This architecture facilitates the integration of the tool into various workflows and platforms, encouraging the continuous adoption of security best practices in container-based systems. To validate the proposal, SK was applied to two large datasets: the first, consisting of 2,107 Kubernetes applications extracted and sanitized from 5,055 packages from Artifact Hub; and the second, composed of 183,225 Kubernetes manifests collected from GitHub. The results revealed a high incidence of security smells, indicating the recurrence of insecure configurations and the importance of automated tools to support secure development. The technological impact is direct, as it provides an automated, extensible, open-source, and easy-to-use solution that contributes to strengthening the security of modern cloud infrastructures. The extensionist nature of the work is evidenced by the public release of the tool and its interaction with technical communities, promoting knowledge dissemination and practical use of the developed solution, as well as the publication of collected metrics and the detailed description of the experimental methodology, making the project reproducible. The project involved one professor and one master's student, and its target audience includes students and professionals in the fields of information security, software engineering, and infrastructure. The impacts are concentrated in the thematic area of technology and production, as defined by the National Extension Policy, and are aligned with the Sustainable Development Goal (SDG) 9 (Industry, Innovation and Infrastructure), by encouraging secure practices and the development of resilient digital solutions.

IMPACTOS SOCIAIS, TECNOLÓGICOS, ECONÔMICOS E CULTURAIS

O trabalho teve como objetivo fortalecer a segurança de ambientes baseados em Kubernetes por meio da identificação de security smells em arquivos de configuração, propondo e avaliando a ferramenta Smelly Kube (SK), desenvolvida com arquitetura cliente-servidor. O cliente, um plugin para o Visual Studio Code, permite que desenvolvedores realizem verificações de segurança diretamente em seu ambiente de desenvolvimento; o servidor, implementado em Golang, processa os manifestos recebidos via requisições HTTP e realiza a detecção de malcheiros de segurança. Essa arquitetura promove a integração da ferramenta a diferentes fluxos de trabalho e plataformas, incentivando o uso contínuo de boas práticas de segurança em sistemas baseados em contêineres. Para validar a proposta, o SK foi aplicado a dois grandes conjuntos de dados: o primeiro, com 2.107 aplicações Kubernetes extraídas e sanitizadas a partir de 5.055 pacotes do Artifact Hub; o segundo, composto por 183.225 manifestos Kubernetes coletados do GitHub. Os resultados mostraram uma alta incidência de security smells, revelando a recorrência de configurações inseguras e a importância de ferramentas automatizadas para apoiar o desenvolvimento seguro. O impacto tecnológico é direto, ao disponibilizar uma solução automatizada, extensível, de código aberto e fácil utilização, que contribui para o fortalecimento da segurança de infraestruturas modernas em nuvem. O caráter extensionista é evidenciado pela publicação pública da ferramenta e pela interação com comunidades técnicas, promovendo a disseminação do conhecimento e o uso prático do que foi desenvolvido, bem como as métricas coletadas no trabalho e a descrição da metodologia abordada na condução dos experimentos, tornando o projeto reprodutível. O trabalho contou com a participação de 1 docente, 1 mestrando, e tem como público-alvo estudantes e profissionais das áreas de segurança da informação, engenharia de software e infraestrutura. Os impactos concentram-se na área temática de tecnologia e produção, conforme a Política Nacional de Extensão, e estão alinhados aos Objetivos de Desenvolvimento Sustentável (ODS) 9 (Indústria, inovação e infraestrutura), ao incentivar práticas seguras e o desenvolvimento de soluções digitais resilientes.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

While the technological development of microservices can significantly improve the speed and agility of software service delivery, it also raises the operational complexity associated with modern applications (LI et al., 2019). Despite several facilitating technologies (e.g., orchestration), there are many challenges to be addressed in the development and deployment of a microservices-based application. Network security, reliability, and latency are critical factors since every transaction implemented using this type of system will involve the transmission of messages across a network. Furthermore, multiple microservices are exposed to a large attack surface (CHANDRAMOULI, 2019).

Containers have revolutionized the software industry by providing a consistent and isolated way to package and run applications. This approach enhances portability, allowing applications to run uniformly across different environments, whether in development, testing, or production. Additionally, the adoption of containers promotes scalability, efficient resource usage, and the automation of deployment processes, leading to faster development cycles and the ability to manage complex applications with greater agility (LI; KANSO; GHERBI, 2015; FELTER et al., 2015).

In this context, Kubernetes has emerged as the leading container orchestration platform. Its rise can be directly attributed to its efficiency and agility in managing scalable applications. Kubernetes provides a solid foundation for creating and deploying applications in containerized environments by providing several important building blocks such as creating workloads, providing connectivity, auto-scaling, and persistence management among others. As a consequence, Kubernetes has simplified the way of deploying and managing complex solutions (OZTOPRAK; TUNCEL; BUTUN, 2023).

Kubernetes is widely recognized as one of the most popular open-source container orchestration tools, adopted by organizations like Adidas, Nokia, Spotify, and even the U.S. Department of Defense (DoD) (KUBERNETES, 2020; FOUNDATION, 2020b). According to the Cloud Native Computing Foundation (CNCF) 2023 Annual Survey, there is a notable increase in Kubernetes adoption among cloud consumers, with 66% using it in production during 2023 compared to 58% during 2022, despite a reduction in evaluation rates (FOUNDATION, 2023). Furthermore, only 15% of organizations consuming cloud services reported no plans to use Ku-

bernetes, indicating its growing importance as a foundational technology in the evolving cloud landscape.

According to Stackrox (2020) survey findings from RedHat, 44% of organizations postpone their deployments due to security concerns. Additionally, the results reveal that 94% of these organizations experienced at least one security incident in the past year, with 69% of these incidents being attributed to security misconfiguration in container environments. Related to this, the Cloud Native Computing Foundation (CNCF) survey indicates that 32% of practitioners among the 1,324 survey participants view security as their main challenge in deployments (FOUNDATION, 2020a).

The State of Kubernetes Security Report (2024) from Red Hat (HAT, 2024), based on a survey of 600 site reliability engineers and security professionals across diverse organizations, outlines key security challenges and strategies. The study reveals that 67% of organizations experience deployment delays due to Kubernetes security issues, with 46% reporting revenue or customer loss from these incidents. Security remains a top concern for 42% of respondents, reflecting widespread apprehension over vulnerabilities and security misconfigurations. DevSecOps practices are maturing, with 42% of organizations reporting advanced initiatives, while 48% are still in early adoption stages, fostering collaboration on security policies and workflows. Additionally, 33% of respondents feel their current security solutions hinder development speed, underscoring the need for more streamlined and effective security approaches in Kubernetes environments.

The IBM Cost of a Data Breach (2024) report research — independently conducted by the Ponemon Institute, and sponsored, reviewed, and published by IBM — studied 604 organizations affected by data breaches between March 2023 and February 2024. Researchers analyzed organizations across 17 sectors in 16 countries and regions, covering breaches that ranged from 2,100 to 113,000 compromised records (IBM, 2024). Ponemon Institute researchers interviewed 3,556 security leaders and executives with firsthand knowledge of the data breach incidents within their organizations to gain direct insights. From the cloud perspective, this study reveals that cloud data breaches increasingly involve multiple environments (public, private, on-premises), making data harder to secure. Breaches on public cloud environments remain the costliest at an average of $5.17 million, to 13.1% last year.

Security misconfiguration, identified as the fifth category in the OWASP Top 10[1] (A05: 2021), represents a critical risk stemming from improper configurations that expose systems to vulnerabilities (OWASP, 2021b). This issue arises when security settings are inadequately defined, insecure defaults remain active, or mismanaged assets are not patched or updated. From this study, OWASP says that 90% of applications were tested for some form of misconfiguration, with an average incidence rate of 4%, and over 208,000 occurrences of a Common Weakness Enumeration (CWE) in this risk category.

A code smell is a recurring coding pattern suggesting possible maintenance issues. While a code smell doesn't necessarily lead to negative outcomes, it warrants consideration since it can indicate underlying problems (FOWLER, 2018). Security smells are recurring coding patterns that indicate potential security weaknesses. Unlike vulnerabilities, which always facilitate a security breach, security smells may not directly lead to a breach but suggest areas that require further attention and improvement (RAHMAN; PARNIN; WILLIAMS, 2019; GHAFARI; GADIENT; NIERSTRASZ, 2017; DEMIR et al., 2019; PONCE et al., 2022; KAMBHAMPATI; MOHAMMED; FARD, 2024; ZHANG; MURPHY; RAHMAN, 2025).

Kubernetes manifests, known as the configuration files for deploying and managing applications in Kubernetes clusters, present significant security challenges related to security smells, especially when involving privileged containers with unnecessary capabilities, insecure user mapping, and improper resource configurations (LIN et al., 2018; OWASP, 2021a; KAMPA, 2024). Security misconfigurations in these manifests represent a top concern that should be prioritized (OWASP, 2022a).

A privileged container, as highlighted by Microsoft (2023), has the same capabilities as the host machine, bypassing the typical limitations of standard containers. An attacker who gains access to a privileged container or can create one through a compromised pod's service account can perform nearly any action on the host itself, leading to a potential full system compromise. Additionally, including unnecessary capabilities in container configurations can expand the attack surface by allowing operations that containers do not require for their intended functionality. Misconfigured resource limits and requests compound these issues by enabling potential denial-of-service (DoS) attacks when containers monopolize node resources.

[1] The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

All the mentioned evidence underscores the importance of inspecting and mitigating security misconfiguration in Kubernetes manifests, as indicated by security smells, raising the following question: given the Kubernetes manifests, how to identify security smells in these infrastructure files? Therefore, the proposed solution of this work is to develop and introduce Smelly Kube, a Static Analysis tool designed to identify security smells in Kubernetes manifest files. Targeted at infrastructure professionals, this tool will pinpoint security smells within each file and provide actionable remediation suggestions. Unlike many existing solutions, Smelly Kube is a free, open-source tool with a REST API as its backend server.

This work also conducts experiments to validate the Smelly Kube's effectiveness, including its application in real-world scenarios and various microservices configurations, from popular repository sources such as Artifact Hub (with 5,055 samples) and GitHub (with 183,225 samples). The security smells identified by Smelly Kube, the experiment results, and the datasets will fit a gap in the research landscape, providing important insights and future directions for studies about security in Kubernetes manifests (RAHMAN et al., 2023).

1.1 Research Questions

Given the prevalence of security incidents linked to misconfigurations, as highlighted in studies such as Red Hat's State of Kubernetes Security Report (2024), it is posited that a substantial portion of these vulnerabilities can be attributed to recurring patterns or "smells" in Kubernetes manifests. By identifying and categorizing these smells, it becomes possible to enhance the visibility of potential security risks during the development and deployment phases, ultimately enabling more proactive mitigation strategies. The Smelly Kube aims to establish that the systematic identification of security smells, provides a practical study for addressing security smells in Kubernetes manifests, answering the following research questions:

a) **RQ**$_1$: how frequently do security smells occur in Kubernetes manifests?
b) **RQ**$_2$: what is the proportion of security smells by Kubernetes workloads?
c) **RQ**$_3$: what is the proportion of security smells by security smell categories?
d) **RQ**$_4$: what is the density of security smells?
e) **RQ**$_5$: what is the most prevalent security smell per workload?
f) **RQ**$_6$: what are the tool's limitations?

1.2 Objectives

The main objective of this work is to design and develop a proposed solution to identify security smells in Kubernetes manifests. The specific objectives are divided into the following topics:

a) provide industry studies about the importance of securing Kubernetes applications;
b) define the scope of security smells from relevant industry sources;
c) develop Smelly Kube, a tool for identifying security smells in Kubernetes manifests;
d) develop a Visual Studio Code plugin that integrates with Smelly Kube;
e) present a practical usage of the proposed solution, using an example scenario;
f) extract and provide Kubernetes manifests datasets, used in the experiments;
g) identify and analyze security smells in Kubernetes manifests using Smelly Kube.

1.3 Work Organization

This work is organized as follows. Chapter 2 presents the main concepts used for the development of this research, such as concepts about computer fundamentals, information security, and technologies. In chapter 3 can be seen the related work, including other market tools and academic works. The proposed solution, Smelly Kube, is described in Chapter 4, detailing the application architecture and the proposed security smells. Chapter 5 presents a detailed methodology for conducting this research, including the metrics to be collected, the employed test scenarios, and the overall test flow. In Chapter 6 are presented the experiments and results from this study, and answers most of the research questions. Finally, Chapter 7 concludes the work with a results discussion and future work proposals.

2 BIBLIOGRAPHYC REVISION

This chapter presents a description of the main concepts and techniques addressed in this work. The subjects were divided into three main sections: Computer Fundamentals, Security, and Technologies.

2.1 Security

In the evolving scenario of containerized infrastructure, ensuring secure Kubernetes configurations is essential as misconfigurations can introduce vulnerabilities. This section explores foundational security principles — such as the CIA triad, least privilege, and the Secure Software Development Life Cycle (SSDLC) — as they apply to identifying security smells in Kubernetes manifests. Security smells are recurring patterns that suggest potential vulnerabilities, requiring further inspection to prevent misconfigurations from escalating into breaches. By addressing these security smells proactively, organizations can enhance Kubernetes resilience and maintain a robust security posture.

2.1.1 CIA Triad

The CIA triad is a widely accepted security model that stands for Confidentiality, Integrity, and Availability (FENRICH, 2008). These principles form the foundation of security policies, ensuring that information is protected from unauthorized access and modification, and is accessible to authorized users when needed. According to the ISO/IEC 27001 standard, the formal definitions of these principles are (ISO, 2022):

a) **Confidentiality**: only the right people can access the information held by the organization. A risk example could be criminals getting hold of a client's login details and selling them on the Darknet;
b) **Integrity**: data that the organization uses to pursue its business or keeps safe for others is reliably stored and not erased or damaged. A risk example could be a staff member accidentally deleting a row in a file during processing;
c) **Availability**: the organization and its clients can access the information whenever it is necessary so that business purposes and customer expectations are satisfied. A risk ex-

ample could be an enterprise database that goes offline because of server problems and insufficient backup.

2.1.2 Principle of Least Privilege

The Least Privilege principle dictates that a security architecture should be designed such that each entity is granted only the minimum system resources and permissions necessary to perform its functions (FORCE, 2017). Restricting access to only what is essential, reduces the attack surface and minimizes the potential impact of compromised accounts or vulnerabilities. This principle prevents unauthorized access to sensitive data, limits the spread of malware, and reduces the likelihood of privilege escalation attacks, where attackers gain higher-level access to critical systems. It also helps prevent accidental or malicious misuse of excessive permissions, ensuring that any breach or failure is contained within the smallest possible scope.

2.1.3 Security Smells

A code smell is a recurring coding pattern suggesting possible maintenance issues. While a code smell doesn't necessarily lead to negative outcomes, it warrants consideration since it can indicate underlying problems (FOWLER, 2018). Code smells can manifest in various forms within a codebase. For instance, they could be long methods, duplicated code, large classes, inappropriate naming, and excessive use of global variables.

Security smells are recurring coding patterns that indicate potential security weaknesses. Unlike vulnerabilities, which always facilitate a security breach, security smells may not directly lead to a breach but suggest areas that require further attention and improvement (RAHMAN; PARNIN; WILLIAMS, 2019). A vulnerability is a weakness that an entity can exploit to modify or access unintended data, disrupt proper execution, or perform unauthorized actions. Identifying a coding pattern as a vulnerability requires considering the application context, which is not a factor for security smells (RAHMAN; WILLIAMS, 2021). Instead, security smells serve as indicators for security-focused inspections that may or may not reveal vulnerabilities. They warrant attention and examination, and if necessary, remediation.

Several studies have explored security smells in different contexts and programming environments, highlighting the significance of identifying and addressing such patterns. They

include identifying security smells in Android applications (GHAFARI; GADIENT; NIERSTRASZ, 2017), smart contracts (DEMIR et al., 2019), Infrastructure-as-Code (IaC) (RAHMAN; WILLIAMS, 2021), in microservices architecture (PONCE et al., 2022), JavaScript code (KAMBHAMPATI; MOHAMMED; FARD, 2024), and Julia programs (ZHANG; MURPHY; RAHMAN, 2025). These studies collectively emphasize the widespread presence of security smells across various domains and the critical need for focused inspection and remediation.

2.1.4 Secure Software Development Life Cycle (SSDLC)

Before discussing the SSDLC, it is important to understand the SDLC. The Software Development Life Cycle (SDLC) is the process of creating or altering systems, and the models and methodologies people use to develop these systems (KUTE; THORAT, 2014). Its primary purpose is to provide a systematic approach that ensures the software's quality and reliability, aligning closely with organizational requirements and user expectations. Common phases of the SDLC include requirements gathering, design, development, testing, deployment, and maintenance.

The Secure Software Development Life Cycle (SSDLC) is an extension of the traditional SDLC, with a dedicated focus on integrating security practices at each stage of the software development process. The SSDLC framework embeds security measures, such as threat modeling, security requirements, code reviews, and vulnerability testing, within the development workflow to proactively identify and mitigate potential security risks (KHAN; ZULKERNINE, 2009). In traditional development, security is often added late, making it costly and complex. Introducing security early in the SDLC reduces costs, shortens timelines, and protects business reputation (MOHINO et al., 2019).

2.2 Clusters, Containers, and DevOps

This section covers essential concepts and technologies for modern software delivery, including Docker, Kubernetes, distributed systems, clusters, Linux containers, and DevOps. Docker and Kubernetes enable efficient containerization and integration across environments, while distributed systems involve autonomous computing elements collaborating as a unified

system. Clusters manage workloads across multiple nodes, ensuring scalability and reliability. Linux containers provide lightweight isolation, offering efficiency over traditional virtualization. DevOps practices, particularly Continuous Integration (CI) and Continuous Delivery (CD), automate workflows and enable rapid, reliable software delivery, fostering collaboration between development and operations teams to meet the demands of modern applications.

2.2.1 Distributed Systems

Distributed systems have come into existence in today's industrial society in some very natural ways. For example, can be observed the emergence of a large number of distributed databases-systems, which have evolved due to the decentralized nature of data sources and the local need for frequent and immediate access to data generated on-site (e.g., the employee database at a branch office of a nationwide organization), while also supporting a global need to view the entire database (KLEINROCK, 1985).

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. This definition refers to two characteristic features of distributed systems. The first one is that a distributed system is a collection of computing elements each being able to behave independently. A computing element, which generally refers to a node, can be either a hardware device or a software process. A second element is that clients, be they people or applications, believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate (STEEN; TANENBAUM, 2016).

2.2.2 Clusters

Clusters are groups of servers that are managed together and participate in workload management. A cluster can contain nodes or individual application servers. A node is usually a physical computer system with a distinct host IP address running one or more application servers. Clusters can be grouped under the configuration of a cell, which logically associates many servers and clusters with different configurations and applications with one another depending on the discretion of the administrator and what makes sense in their organizational environments (IBM, 2023).

2.2.3 Linux Containers

Containers, superficially at least, share similarities with conventional virtualization technologies such as VMWare, QEMU, Hyper-V, and Xen. This likeness stems from their capability to facilitate the concurrent operation of multiple isolated applications on a single host computer, which runs on top of the virtualized machine without the knowledge of any intermediary layer (BARHAM et al., 2003).

Conventional virtualization entails executing a complete guest operating system (OS) within an isolated environment, wherein the guest OS kernel communicates with the host through a comprehensive set of virtualized devices (CPU, disk, memory, etc.). In contrast, Linux containers leverage the host's underlying kernel while segregating the processes within it from all other processes on the host operating system. Rather than virtualizing the hardware, containers virtualize the operating system, presenting a distinct approach to isolation and resource allocation (MERKEL et al., 2014; BERNSTEIN, 2014).

While traditional virtualization and containers yield a similar outcome — an isolated, distinct, and portable computing environment — differ significantly in user experience. Conventional virtual machines load entire guest operating systems into memory, consuming substantial storage space and system resources even before executing any tasks, often taking minutes to start. In contrast, containers share the host kernel, utilizing dependencies efficiently. They use minimal resources when idle, allowing thousands to run on a single host. Container startup times mirror those of native applications, as the host merely initiates isolated processes (HALE et al., 2017).

2.3 DevOps

DevOps bridges the gap between development and operations by using automation in development, deployment, and infrastructure monitoring. It represents an organizational shift from isolated, siloed teams to cross-functional collaboration, enabling continuous delivery of operational features. This approach accelerates the delivery of value, minimizes miscommunication, and enhances problem resolution. At its core, DevOps fosters a culture of collaboration between development, quality assurance, and operations, driving efficiency and innovation (EBERT et al., 2016).

In the context of modern application delivery, Continuous Integration (CI) and Continuous Delivery (CD) are fundamental components of DevOps (HAT, 2020a). CI ensures that code changes are automatically tested and integrated into a shared repository, reducing the risk of errors and enabling faster feedback. CD takes this a step further by automating the deployment process, allowing code to be delivered to production quickly and reliably. Together, CI/CD pipelines enable rapid iteration and frequent releases, which are essential in today's fast-paced software development environment. This approach not only accelerates the delivery of new features but also enhances the ability to quickly address bugs, improve system stability, and respond to user feedback, all of which are critical in maintaining competitive advantage and meeting user expectations in modern applications.

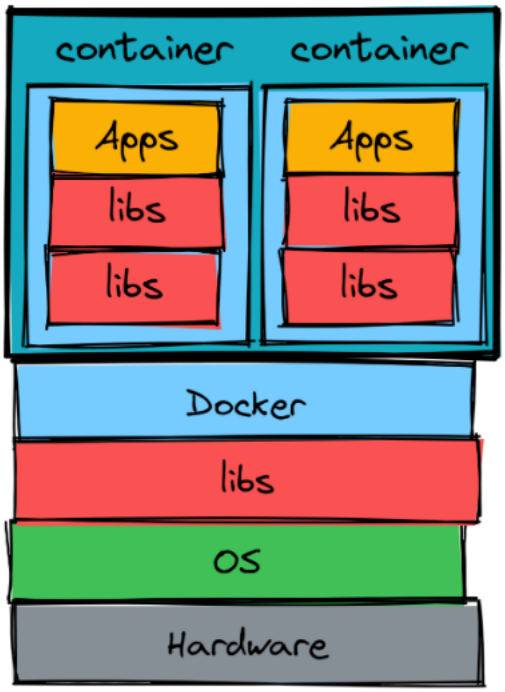
2.3.1 Docker

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Containers isolate software from its environment and ensure that it works uniformly despite differences, for example, between development and staging (DOCKER, 2022). With Docker, it is possible to treat containers like highly lightweight modular virtual machines (HAT, 2018). A key difference between Docker images and other virtual machines is that the Docker images share the Linux kernel with the host machine (BOETTIGER, 2015).

Docker is an open platform for developing, shipping and running applications. Docker enables the separation of applications from the infrastructure so it can deliver software quickly. With Docker, is possible to manage infrastructure in the same way as an application. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, it can significantly reduce the delay between writing code and running it in production (DOCUMENTATION, 2022a).

Figure 2.1 shows how Docker is structured in a system. Docker can be executed on any operating system that is compatible with the Docker Engine. The Docker Engine serves as a container management tool, enabling the creation of containers through the utilization of Docker images. These containers encompass the necessary application libraries and configurations.

Figure 2.1 – Docker structure.



Source: made by author (2022).

2.3.1.1 Dockerfile

A Dockerfile is a text-based script used to define and configure the environment and instructions for building a Docker container image. It serves as a blueprint for creating these containers by specifying the base image, setting up the required software packages, copying files, configuring settings, and more (DOCUMENTATION, 2022b).

In essence, a Dockerfile contains a series of instructions that Docker's build process follows to create a container image. This image can then be used to launch new containers, each of which runs in an isolated environment with its own filesystem, processes, and network settings. An example of a Dockerfile for a Python application is shown in Figure 2.2. It starts by using the base Python image in version 3.9, installs the project dependencies, adds the source files from the host machine to the container, and prepares the entry point to run the application.

2.3.2 Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating containerized applications' deployment, scaling, and management (KUBERNETES, 2022j). It is possible to

Figure 2.2 – Dockerfile example for a Python application.

Dockerfile

```
1   FROM python:3.9
2
3   ADD requirements.txt .
4   RUN pip3 install -r requirements.txt
5
6   WORKDIR app
7   ADD src/ app/
8
9   ENTRYPOINT ["python3 main.py"]
```

Source: made by author (2022).

cluster together groups of hosts running Linux containers, and Kubernetes helps to easily and efficiently manage those clusters. Kubernetes clusters can span hosts across on-premise, public, private, or hybrid clouds. For this reason, Kubernetes is an ideal platform for hosting cloud-native applications that require rapid scaling, like real-time data streaming through Apache Kafka (HAT, 2020b).

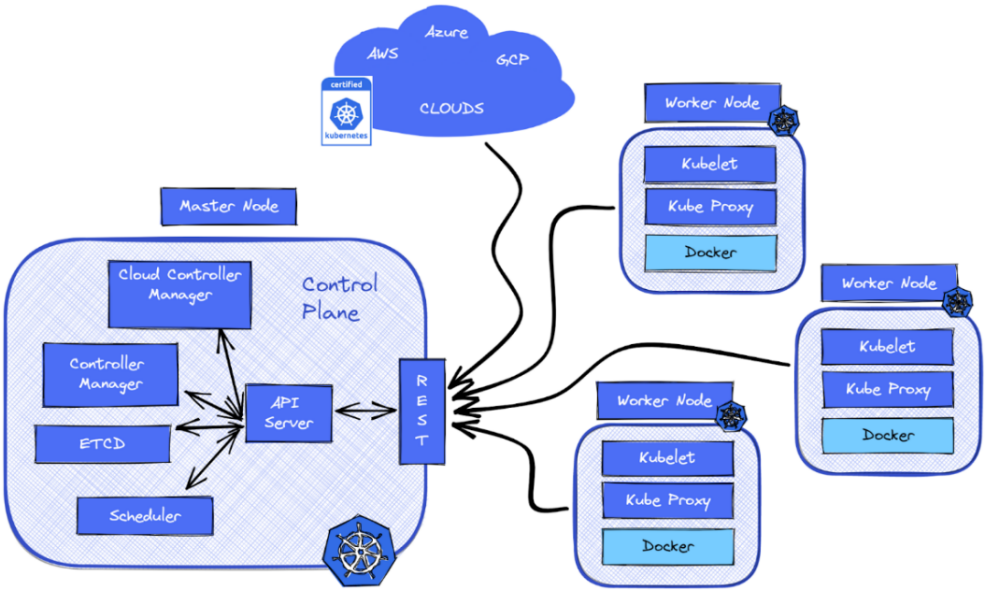
In Kubernetes, applications are running inside containers. A Kubernetes Pod is a group of one or more containers that share the same resources and can communicate with each other through localhost or various interprocess communication methods (BALLA; SIMON; MALIOSZ, 2020). Figure 2.3 shows the Kubernetes architecture and its components that will be described in the following sections.

2.3.2.1 Control Plane

The control plane refers to the set of components that collectively manage and control a Kubernetes cluster. It is responsible for orchestrating and coordinating all activities within the cluster, including scheduling containers, monitoring their health, maintaining the desired state, and managing the overall cluster infrastructure.

The control plane components work together to ensure the proper functioning of the cluster and provide the necessary services for managing containerized applications (KUBER-

Figure 2.3 – Kubernetes architecture.



Source: made by author (2022).

NETES, 2022c). The main components of the Kubernetes control plane are described as follows:

a) **API Server**: acts as the central management interface of the Kubernetes cluster, exposing the Kubernetes API for interaction with users and other components. It handles requests, performs authentication and authorization, and enforces policies. The main implementation, `kube-apiserver`, can scale horizontally by running multiple instances for load balancing (KUBERNETES, 2022c);

b) **etcd Database**: a distributed key-value store that acts as the reliable source of truth for all cluster configuration data (KUBERNETES, 2022c; ETCD, 2013). It stores information about nodes, Pods, services, and other Kubernetes objects, ensuring consistency and high availability through a distributed consensus algorithm;

c) **Scheduler**: responsible for assigning Pods to nodes based on resource requirements, node availability, and specific constraints to optimize cluster resource utilization and maintain high availability. It considers factors such as resource needs, hardware/software/policy constraints, affinity/anti-affinity rules, data locality, and deadlines (KUBERNETES, 2022c);

d) **Cloud Controller Manager**: manages interactions with the cloud provider's APIs and services, abstracting cloud-specific details from the Kubernetes core (KUBERNETES,

2022b). This component includes controllers for handling cloud-specific tasks such as node lifecycle, routing, volume management, and Identity and Access Management (IAM);

e) **Kube Controller Manager**: runs a variety of controllers to maintain the cluster's desired state. Each controller is a control loop dedicated to a specific task, such as node management, Pod scheduling, service endpoints, and replication. Unlike the cloud controller manager, it focuses on general Kubernetes resource management and cluster state maintenance (KUBERNETES, 2022h).

2.3.2.2 Worker Node

A Worker node is responsible for running the applications and workloads. Each Worker node hosts and manages Pods, the smallest deployable units in Kubernetes that contain one or more containers. Worker nodes contain essential services such as the kubelet, which communicates with the Kubernetes control plane to manage the state of the node and the Pods it runs; the kube-proxy, which manages networking for forwarding traffic to appropriate containers; and a container runtime, such as Docker, to run and manage container lifecycles (KUBERNETES, 2022a). These nodes can be physical servers or virtual machines and work together to form a scalable, distributed environment that ensures the application remains available and performant.

2.3.2.3 Workloads

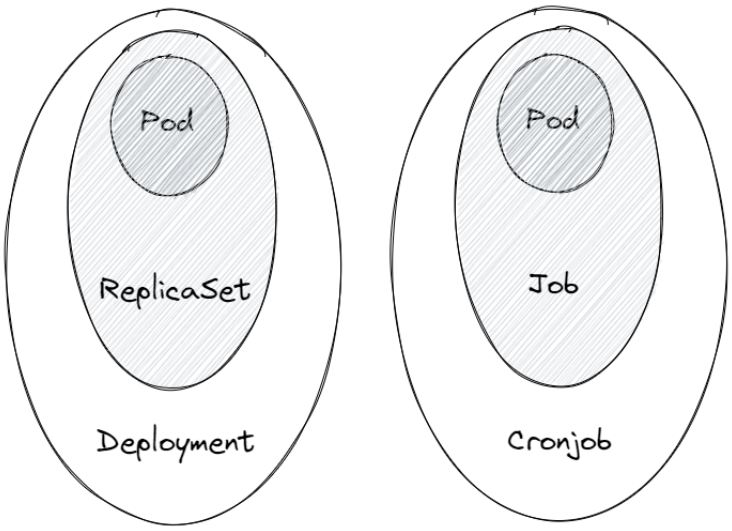
A workload in Kubernetes refers to any application or service running within the cluster. Whether it comprises a single component or multiple interconnected components, Kubernetes deploys and manages workloads as a set of Pods. A Pod, in Kubernetes, is the smallest deployable unit, encapsulating one or more containers that share the same network namespace and storage.

Pods in Kubernetes follow a well-defined lifecycle. For example, if a Pod is active within the cluster and a severe issue impacts its node, all Pods on that node will fail. Kubernetes considers such failures as non-recoverable for the affected Pods and triggers the creation of new Pods on healthy nodes for recovery, even if the original node becomes operational again.

To streamline management, Kubernetes eliminates the need to handle individual Pods directly. Instead, workload resources can be leveraged to oversee groups of Pods. These re-

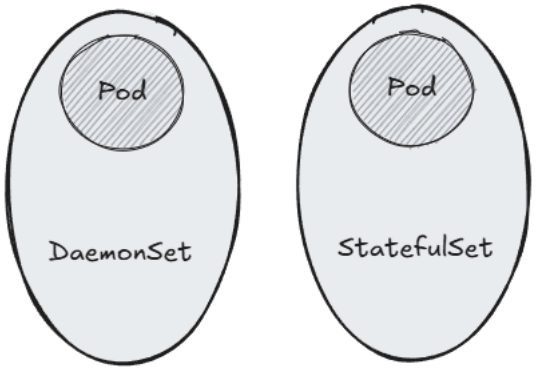
sources configure controllers that maintain the desired state by ensuring the correct number of Pods of a specified type is running. Common Kubernetes workloads include Pods, ReplicaSets, Deployments, StatefulSets, DaemonSets, Jobs, and CronJobs (KUBERNETES, 2022m). Figure 2.4 shows the hierarchical structure of ReplicaSet, Deployment, Job, and CronJob. It can be seen that the Deployment inherits from ReplicaSet, and the same occurs in CronJob which inherits from Job. For StatefulSet and DaemonSet, Figure 2.5 illustrates that they only inherit from Pods.

Figure 2.4 – Kubernetes workloads hierarchy 1.



Source: made by author (2024).

Figure 2.5 – Kubernetes workloads hierarchy 2.



Source: made by author (2024).

a) **ReplicaSet**: it ensures that a specified number of identical Pods run continuously for scaling and load balancing purposes. It maintains the desired number of replicas by creating

or terminating Pods as needed, using labels and selectors to identify and manage Pods (KUBERNETES, 2022k);

b) **Deployment**: as an extension of a ReplicaSet, a Deployment has the same features of running continuously and maintaining the desired number of replicas. In addition, it supports versioning, rolling updates, and automatic rollback, ensuring high availability and seamless updates (KUBERNETES, 2022f);

c) **StatefulSet**: manages stateful applications by maintaining a consistent identity for each Pod and associating them with persistent storage volumes. This ensures that Pods retain their identifiers and persistent data across rescheduling, which is essential for applications needing stable storage (KUBERNETES, 2022l);

d) **DaemonSet**: ensures that a copy of a specific Pod runs on all or a subset of Nodes in the cluster, providing node-local services such as monitoring, logging, or storage daemons. Pods are automatically added or removed as nodes are added or removed from the cluster (KUBERNETES, 2022e);

e) **Job**: a Job represents a task or batch process that runs to completion, creating and managing Pods to perform the task until successful termination. Jobs are suitable for finite-duration tasks like data processing or batch operations and ensure task completion by recreating failed Pods (KUBERNETES, 2022g);

f) **CronJob**: a CronJob schedules and automates the execution of Jobs at specific time intervals using a cron[1] syntax. It manages recurring tasks such as backups or data synchronization by creating Jobs according to a defined schedule, ensuring timely and repetitive task execution (KUBERNETES, 2022d).

2.3.3 Kubernetes Manifests

A Kubernetes manifest, typically written in YAML[2] or JSON[3], is a declarative configuration file or set of files that define the desired state of resources within a cluster. Kubernetes generally uses configuration files called manifests to specify what resources should be created,

[1] Cron command-line utility is a job scheduler on Unix-like operating systems, used by users who create and maintain scheduled subroutines

[2] YAML is a human-readable data serialization language that is often used for writing configuration files.

[3] JSON stands for JavaScript Object Notation. JSON is a lightweight format for storing and transporting data.

updated, or deleted. Suppose there are differences between the manifest and the etcd when applying it. In that case, Kubernetes reconciles them, such as creating new resources, updating existing ones, or removing unnecessary ones (KUBERNETES, 2022i). A Kubernetes manifest typically includes the following information:

a) **API Version**: Specifies the Kubernetes API version;
b) **Kind**: Defines which workload will be used such as Pod, Deployment, Service, etc;
c) **Metadata**: Contains metadata about the resource, including its name, namespace, labels, and annotations;
d) **Spec**: Defines the desired state of the resource. For example, in the case of a Deployment, it might specify the container images to use, the number of replicas, resource requirements, etc.

An example of a Kubernetes manifest about a Deployment application can be found in Figure 2.6. It defines three replicas on its `spec` key, which ensures that three identical Pods will run. The `selector` is used to track which resource will be replicated, and in that case, it will match a label `app: demo-nginx-app` defined in the Pod specification. Finally, the container defines its name, the image, and the container port that other applications will access.

Figure 2.6 – Kubernetes manifest of a nginx application.

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: demo-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-nginx-app
  template: # equivalent to a Pod workload definition
    metadata:
      labels:
        app: demo-nginx-app
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Source: made by author (2024).

This chapter has established the foundational concepts and technologies crucial to this research, covering computer fundamentals, security principles, and technologies like Docker and Kubernetes. With this groundwork laid, the next chapter will review existing tools and research on Kubernetes security, highlighting current solutions, their strengths, and gaps. This sets the stage for identifying how this work will contribute to the field.

3 RELATED WORK

This chapter reviews key tools and research relevant to Kubernetes security, focusing on solutions for identifying and mitigating security smells in Kubernetes manifests. First, it highlights academic research that advances security in microservices and Infrastructure-as-Code, especially through static analysis. These insights lay the groundwork for identifying areas where this work aims to contribute. Then, it examines popular market tools like Snyk and Kube-score, which provide robust vulnerability scanning but face customization limits.

3.1 Academic Works

There are related works about securing microservice applications, which are also crucial (SOLDANI; TAMBURRI; Van Den Heuvel, 2018). For instance, Chondamrongkul, Sun and Warren (2020) proposed a technique to automatically identify security threats in microservice applications based on a collection of formally defined security characteristics. The authors Schneider and Scandariato (2023) and Zdun et al. (2023) contribute by supporting security checks in microservice applications through the extraction of security-aware dataflow diagrams and the provision of KPI metrics to measure the effectiveness of security measures, respectively.

In the context of Infrastructure-as-Code (IaC), the work Rahman, Parnin and Williams (2019) employs a static analysis approach to detect security smells IaC scripts, which is related to the subject of this work, including analysis for unnecessary privileges for containers up to detecting hardcoded secrets. By qualitatively analyzing 1,726 IaC scripts, the authors identify seven security smells in 15,232 scripts from 293 open-source repositories. Their findings reveal 21,201 occurrences of security smells, including 1,326 hard-coded passwords. The Zhang, Murphy and Rahman (2025) paper conducts an empirical study on security smells in Julia programs, analyzing 4,592 programs across 126 open-source projects. The study identifies seven security weakness categories and develops the Julia Static Analysis Tool (JSAT) to automatically detect vulnerabilities. Findings highlight the prevalence of security weaknesses, with 24.9% of Julia source code files affected, and recommend further development of security inspection tools. In Ponce et al. (2022) the authors conduct a multivocal review of 58 studies to identify ten security smells in microservice-based applications and their associated refactorings. The study organizes these smells into a taxonomy, linking each one to the security properties it violates.

Findings provide practical guidelines for practitioners and offer a foundation for future research on securing microservices.

The paper Rahman et al. (2023) is directly related to identifying security misconfigurations in Kubernetes manifests. The automation described by the authors extends beyond the final Kubernetes manifest, known as the Kind manifest, to include Helm template analysis. The authors conducted an empirical analysis of 2,039 Kubernetes manifests from 92 open-source repositories, revealing 11 categories of misconfigurations, including absent resource limits and security contexts, as well as the activation of hostIPC.

To better understand the prevalence of security defects in Kubernetes manifests, the authors of Bose, Rahman and Shamim (2021) analyzed 5,193 commits across 38 OSS (open-source software) repositories, finding that only 0.79% of these commits address security concerns. This low figure suggests that many security vulnerabilities in Kubernetes manifests remain undetected. In Habbal (2020) the author addresses how Kubernetes security configurations can enhance the availability of microservices for critical systems, an area essential due to the rising security challenges in cloud-based solutions. Also, German and Ponomareva (2023) advocates for real-time monitoring tools as an added security layer and compares several solutions to enhance Kubernetes protection.

When comparing the proposed solution to the work by Rahman et al. (2023), Smelly Kube demonstrates an advantage by detecting additional security smells not covered in their research, such as enforcing a read-only root filesystem and running containers as a non-root user. Furthermore, their study lacks a refined categorization of certain smells; for instance, the absence of resource limits could be more precisely divided into 'absent resource requests' and 'absent resource limits'. This nuanced categorization would aid in assessing the percentage of projects that fail to configure these settings individually, offering deeper insights into security practices.

Also, the authors of Rahman et al. (2023), Rahman, Parnin and Williams (2019), Bose, Rahman and Shamim (2021) use, in general, popular code repositories for extracting their datasets, like GitHub and GitLab. This work about Smelly Kube not only assets GitHub repositories but also Artifact Hub Helm projects, which were not covered by these studies.

3.2 Market Tools

In the rapidly evolving of Kubernetes security, numerous tools have emerged to assess and secure Kubernetes manifests and the cluster itself, including popular solutions like Snyk[1], Kube-score[2], Kube-bench[3], and FluidAttacks[4], see Figure 3.1 for characteristics comparison. These tools collectively offer a robust set of features to enhance the security and reliability of Kubernetes environments, catering to different aspects of security assessment and compliance. Among these, Snyk supports both Command-Line Interface (CLI) and Visual Studio Code (VS Code) integrations, providing a flexible setup for developers and security teams.

Figure 3.1 – Comparison of Kubernetes Security Tools.

Tool	OSS	CLI	GUI	API	Scans Manifests	Scans Clusters
Snyk		✓	✓	✓	✓	✓
Kube-score	✓	✓			✓	
Kube-bench	✓	✓				✓
FluidAttacks		✓	✓	✓	✓	✓
Smelly Kube	✓			✓	✓	

Source: made by author (2024).

Before analyzing each tool, let's explain the table columns, which represent the compared features. Open Source Software (OSS) refers to software with source code that anyone can inspect, modify, and enhance. A Command-Line Interface (CLI) allows users to interact with software through text-based commands, while a Graphical User Interface (GUI) provides a visual way to operate applications. An Application Programming Interface (API) enables software components to communicate programmatically, supporting automation and integration. Furthermore, 'Scans Manifests' refer to the tool's ability to analyze Kubernetes configuration files, while 'Scans Clusters' indicate whether the tool can assess the security of a running Kubernetes environment.

Snyk, a developer-centric security platform, specializes in identifying and remediating vulnerabilities across multiple layers: source code, dependencies, container images, and infrastructure-as-code (IaC), including Kubernetes manifests. Kube-score evaluates the quality

[1] https://snyk.io/
[2] https://kube-score.com/
[3] https://github.com/aquasecurity/kube-bench
[4] https://fluidattacks.com/

and security of Kubernetes configurations by providing a detailed score based on best practices and recommendations, helping users improve their manifests.

Kube-bench, developed by Aqua Security, runs checks based on the Center for Internet Security (CIS) Kubernetes Benchmark, offering a comprehensive assessment of a cluster's security posture. FluidAttacks focuses on continuous vulnerability assessment and penetration testing, combining automated scanning with manual testing to identify and remediate security issues in Kubernetes clusters and applications.

Both Snyk and FluidAttacks share significant similarities with Smelly Kube, not just in having APIs and the ability to scan manifests but also in their capability to detect the same security smells as Smelly Kube. The key difference is that Smelly Kube is an OSS project, meaning anyone can continue this work in the future, open to the community to improve it, and deploy it wherever they want. Also, Kube-score can detect some of the security smells proposed by Smelly Kube, such as resource requests and limits, but does not handle capabilities and privilege escalation settings for example.

This study was driven by the need to address a research gap identified in the academic context, particularly the limited coverage of certain security smells in existing literature and the overall scarcity of studies focused on security within Kubernetes manifests. From the perspective of market tools, Smelly Kube was designed as an open-source solution capable of seamless integration with any client via HTTP requests through its API. The experiments across different dataset sources, with a granular categorization of the security smells, bring important insights into the security maturity of these Kubernetes manifest samples.

To summarize, this chapter has explored Kubernetes security tools and related academic research, highlighting both strengths and gaps in current approaches to securing Kubernetes manifests and comparing them with Smelly Kube. These insights set the stage for the next chapter, which will delve into the specifics of Smelly Kube, including its architecture, the security smells it addresses, the application flow, and the algorithms behind its functionality. This deeper examination will demonstrate how Smelly Kube aims to elevate the standards for Kubernetes security assessment.

4 SMELLY KUBE

This chapter comprehensively explains the Smelly Kube architecture and application flow, rules and their sources, and a case study.

4.1 Definition

This work aims to develop and introduce Smelly Kube, a Static Analysis tool designed to identify security smells in Kubernetes manifests. Targeted at infrastructure professionals, this tool will pinpoint security smells within each file and provide actionable remediation suggestions. It covers the following Kubernetes workloads: Pod, Job, CronJob, ReplicaSet, Deployment, StatefulSet, and DaemonSet. Before the security smells definition, it is important to understand the main Kubernetes manifest configurations as follows:

a) **Resource Request (RR)**: Ensuring that resource requests are properly defined is essential for maintaining the stability and performance of the Kubernetes cluster. It helps prevent resource contention and ensures critical applications receive the necessary resources to function effectively;
b) **Resource Limits (RL)**: Setting resource limits is vital for preventing any single application from consuming excessive resources, which could lead to the starvation of other applications. Properly defined limits help maintain a balanced and fair distribution of resources across the cluster;
c) **Security Context Run as User (SCRU)**: Defining security contexts for users is important for enforcing the principle of least privilege. It ensures that applications run with only the necessary permissions, reducing the risk of privilege escalation and potential security breaches;
d) **Security Context Capabilities (SCC)**: Managing security context capabilities helps control the actions that containers are permitted to perform. This prevents containers from gaining unnecessary privileges that attackers could exploit to compromise the system;
e) **Security Context Allow Privileged Escalation (SCAPE)**: Preventing privileged escalation is crucial for maintaining the integrity and security of the Kubernetes environment. By disallowing privileged escalation, you can mitigate the risk of containers gaining el-

evated privileges that could be used to attack the underlying infrastructure. It controls whether a process can gain more privileges than its parent process;

f) **Security Context Read-Only Root File System (SCROFS)**: Enforcing a read-only root filesystem is an important security measure that prevents unauthorized filesystem modifications. This helps protect the integrity of the application and reduces the attack surface available to potential attackers;

g) **Security Context Privileged (SCP)**: Running containers in privileged mode can pose significant security risks, as it grants the container access to the host's resources and capabilities. Identifying and avoiding privileged containers helps safeguard the host system and other containers from attacks;

h) **Security Context Run as Non-Root (SCRNR)**: Enforcing that containers run as non-root users is a key security measure. Running containers with non-root privileges reduces the risk of privilege escalation attacks, ensuring that even if an attacker compromises a container, their ability to harm the host system or other services is limited.

Figure 4.1 shows how each of the configurations shown before relates to CIA Triad and CWE[1], indicating their security property and associated weakness. Smelly Kube was designed to detect the absence of essential security configurations and the presence of non-compliant values within them.

The security smells catalog can be found in Figure 4.2, and Section 4.2.2 elaborates on the underlying algorithms that drive the identification of these smells. The sources used to research the security smells are OWASP (2022b), Series (2020), Series (2019), CIS (2024), which were from relevant industry institutions such as OWASP and CIS.

[1] CWE (Common Weakness Enumeration) is a standardized list of software weaknesses and vulnerabilities used to identify and categorize security flaws in software systems

Figure 4.1 – Kubernetes Security Configurations and Corresponding CWEs and CIA Triad.

Code	CIA Triad	CWE
RR	Availability	CWE-754: Improper Check for Unusual or Exceptional Conditions
RL	Availability	CWE-400: Uncontrolled Resource Consumption
SCRU	Confidentiality, Integrity	CWE-269: Improper Privilege Management
SCC	Confidentiality, Integrity	CWE-266: Incorrect Privilege Assignment
SCAPE	Confidentiality, Integrity	CWE-269: Improper Privilege Management
SCROFS	Integrity	CWE-732: Incorrect Permission Assignment for Critical Resource
SCP	Confidentiality, Integrity	CWE-250: Execution with Unnecessary Privileges
SCRNR	Confidentiality, Integrity	CWE-269: Improper Privilege Management

Source: made by author (2024).

4.1.1 Application Architecture

Smelly Kube works with a client/server architecture that communicates via the HTTP protocol, as shown in Figure 4.3. This approach enables integrations with various types of clients, such as CI/CD pipelines and code editors. The server is responsible for receiving the manifest files, analyzing them looking for security smells, and sending a response to the client. It expects the file name to be sent, alongside the manifest content, as part of a JSON object in the request body. The client is responsible for requesting the server and parsing the response to show the security smells found to the user. More details about the response structure will be presented in the following section 4.2.1.

Figure 4.2 – Security smells catalog.

Code	Definition
RR_UNSET	Absence of Resource Requests
RL_UNSET	Absence of Resource Limits
SCRU_UNSET	Absence of Security Context Run as User
SCC_UNSET	Absence of Security Context Capabilities
SCAPE_UNSET	Absence of Security Context Allow Privilege Escalation
SCROFS_UNSET	Absence of Read-Only Root Filesystem
SCRNR_UNSET	Absence of Security Context Run as non Root
SCRU_VALUE	The container UID is set to zero (root)
SCC_VALUE	The capabilities are not configured to be all removed
SCAPE_VALUE	The container allows privilege escalation
SCROFS_VALUE	The root filesystem is writable
SCP_VALUE	The container is running in privileged mode
SCRNR_VALUE	Runtime validation of the root user is disabled

Source: made by author (2024).

4.2 Server

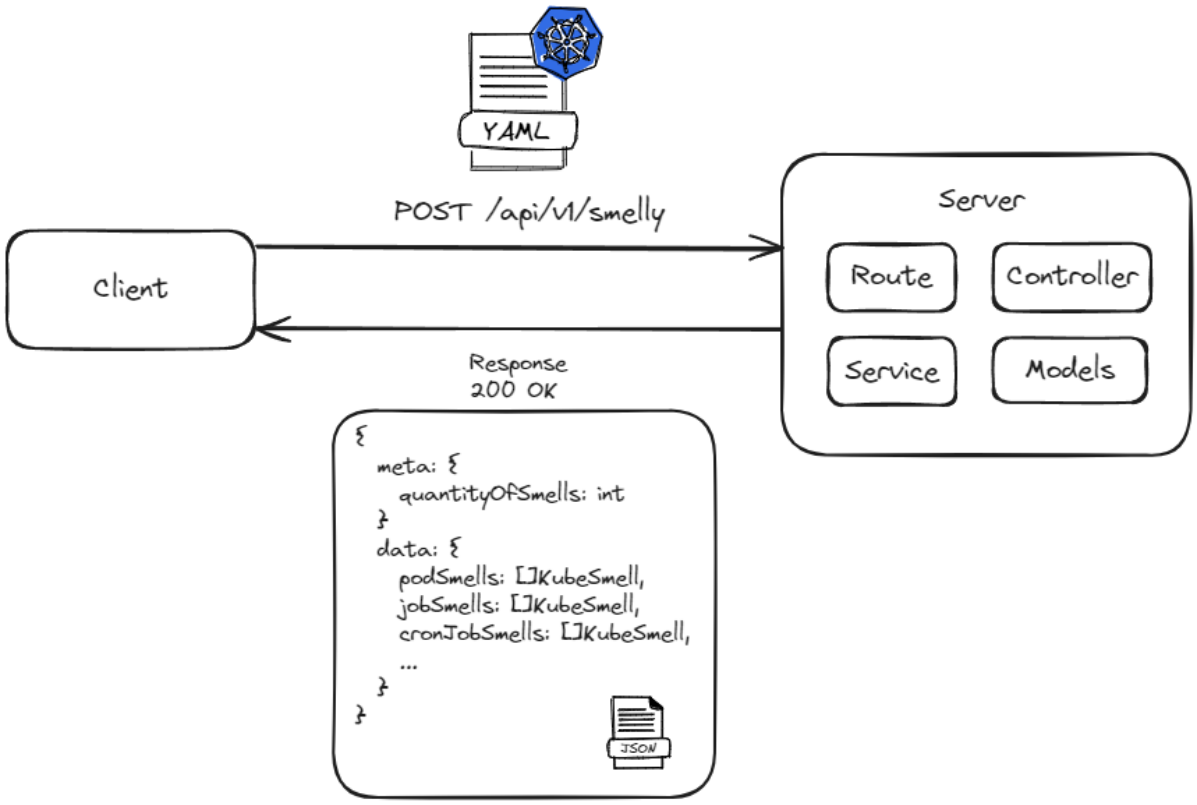
Smelly Kube was written in Golang. The libraries used are YAML file parsing, Kubernetes workloads structs, and REST API server with Go Fiber[2] lib to handle the HTTP requests. It listens on a port configured by the user with an environment variable and accepts POST requests through the ***/api/v1/smelly*** path.

4.2.1 Application Flow

Figure 4.4 presents this application flow. When the Smelly Kube receives an invalid YAML file, or it is not a valid Kubernetes workload, it sends an error response to the client. Otherwise, it decodes the received YAML string into a workload struct and puts this decoded object into a slice, for each workload found in the string after splitting it by the document separator.

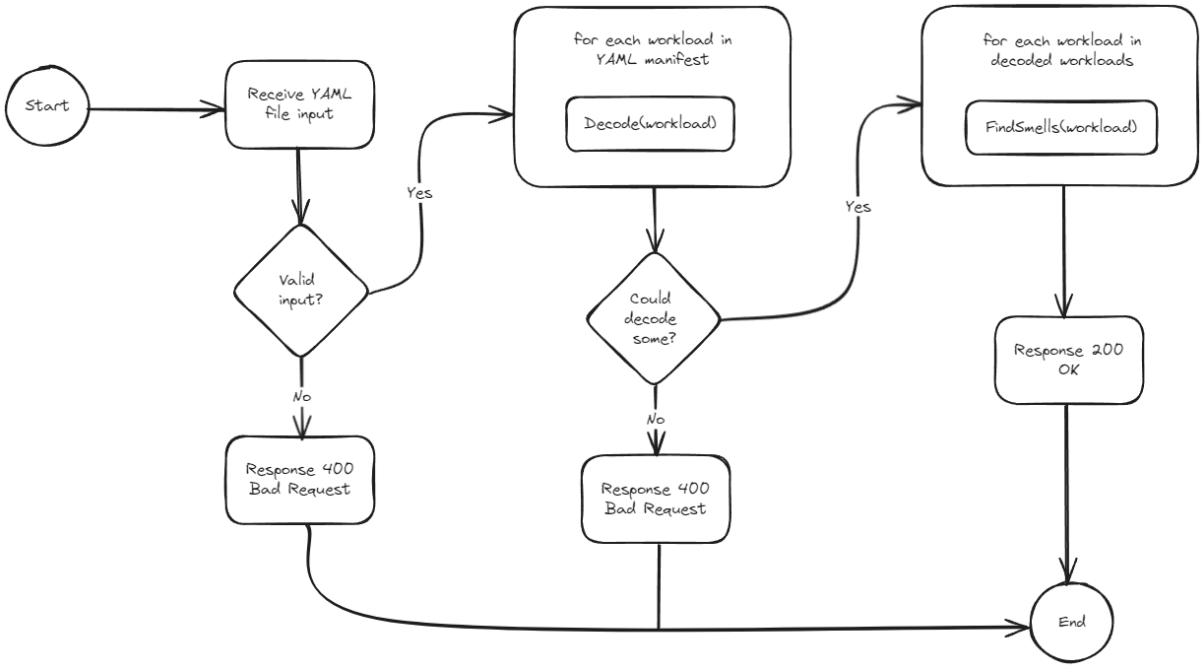
[2] `https://gofiber.io/`. Accessed on June 20, 2024

Figure 4.3 – Application architecture.



Source: made by author (2024).

Figure 4.4 – Application flow.



Source: made by author (2024).

The application utilizes different packages from Kubernetes API library[3], which is synced with the official Kubernetes repository, to decode the workloads into struct objects. For decoding ReplicaSet, Deployment, DaemonSet, and StatefulSet the application uses the `k8s.io/api/apps/v1`[4] package; for Job and CronJob it uses `k8s.io/api/batch/v1`[5] package; and finally, for Pod, it uses the package `k8s.io/api/core/v1`[6].

After decoding the workload, each decoded workload is passed to its respective analyzer. These analyzers identify security smells in the context of the Pod **spec**. Each analyzer will execute the algorithms described in Section 4.2.2, appending the smells when they are found into a slice.

The `KubernetesSmell` structure holds information about the affected workload, the security smell category, its description, and the remediation. Finally, the server sends the response with status 200 (OK), and the response body object is built with meta and data keys. The meta key (metadata) provides information about the number of security smells found from the manifest and the quantity of decoded workloads (e.g. the entry had two Deployments and one CronJob successfully decoded). The data key provides the report, containing each workload's security smells list.

4.2.2 Smelly Algorithms

The `SmellySecurityContextRunAsUser` identifies potential security risks in Kubernetes workloads by examining the `RunAsUser` attribute at both the Pod and container levels, see Algorithm 1 for details. It begins by checking if the `RunAsUser` is set at the Pod level in lines 2, and whether it has an insecure value, such as `0`, which allows the container to run as the root user, as seen in line 3. The algorithm then iterates through each container starting at line 4. If a container lacks a `SecurityContext` or `RunAsUser` setting, as checked in line 5, it inherits the Pod-level configuration. If neither the Pod nor the container has `RunAsUser` set, the algorithm flags it as a security smell (`SCRU_UNSET`), as shown in lines 6 to 8. However, if the Pod-level value is insecure, set to `0` in line 3, the procedure flags as a security smell (`SCRU_VALUE`), as checked in line 9. If a container explicitly has `RunAsUser` set to `0`, the algorithm flags this as

[3] `https://pkg.go.dev/k8s.io/api`. Accessed on June 20, 2024
[4] `https://pkg.go.dev/k8s.io/api/apps/v1`. Accessed on June 20, 2024
[5] `https://pkg.go.dev/k8s.io/api/batch/v1`. Accessed on June 20, 2024
[6] `https://pkg.go.dev/k8s.io/api/core/v1`. Accessed on June 20, 2024

a smell (SCRU_VALUE), as seen in lines 13 and 14. This approach identifies security risks when containers may inadvertently run with root privileges, promoting safer Kubernetes deployments.

Algorithm 1 - Check Security Context for RunAsUser.

Algorithm 1 Check Security Context for RunAsUser

```
1:  procedure SmellySecurityContextRunAsUser(spec)
2:      isSetAtPodLevel ← (spec.SecurityContext ≠ nil) and (spec.SecurityContext.RunAsUser ≠ nil)
3:      isValueSmellAtPodLevel ← isSetAtPodLevel and (spec.SecurityContext.RunAsUser = 0)
4:      for each container in spec.Containers do
5:          if container.SecurityContext = nil or container.SecurityContext.RunAsUser = nil then
6:              if not isSetAtPodLevel then
7:                  AppendKubernetesSmell(SCRU_UNSET)
8:              else if isValueSmellAtPodLevel then
9:                  AppendKubernetesSmell(SCRU_VALUE)
10:             end if
11:             continue
12:         end if
13:         if container.SecurityContext.RunAsUser = 0 then
14:             AppendKubernetesSmell(SCRU_VALUE)
15:         end if
16:     end for
17: end procedure
```

Source: made by author (2024).

The SmellySecurityContextRunAsNonRoot algorithm identifies the potential security smells related to the RunAsNonRoot attribute in Kubernetes Pods and containers, as shown in Algorithm 2. The algorithm begins by checking if the RunAsNonRoot attribute is set at the Pod level and whether it enforces a non-root user (i.e., set to true) in lines 2 and 3. If the Pod-level RunAsNonRoot is explicitly set to false, this is flagged as a potential security smell (line 3). The algorithm then iterates through each container in the Pod, starting at line 4. If a container does not have a SecurityContext or if the RunAsNonRoot attribute is not set in line 5, it checks the Pod-level configuration. If the Pod-level configuration is not set, as seen in line 5, the algorithm flags this as a security smell (SCRNR_UNSET) in line 6. If the Pod-level RunAsNonRoot is set to false, the algorithm flags this as a security smell (SCRNR_VALUE) in line 9. Finally, if a container has its RunAsNonRoot attribute explicitly set to false (line 13), the algorithm flags this as a security smell (SCRNR_VALUE) in line 14. This ensures that containers are properly checked for non-root configurations, preventing containers from running with root privileges unless explicitly configured otherwise.

Algorithm 2 - Check Security Context for RunAsNonRoot.

Algorithm 2 Check Security Context for RunAsNonRoot

```
1:  procedure SmellySecurityContextRunAsNonRoot(spec)
2:      isSetAtPodLevel ← (spec.SecurityContext ≠ nil) and (spec.SecurityContext.RunAsNonRoot ≠ nil)
3:      isValueSmellAtPodLevel ← isSetAtPodLevel and (spec.SecurityContext.RunAsNonRoot = false)
4:      for each container in spec.Containers do
5:          if container.SecurityContext = nil or container.SecurityContext.RunAsNonRoot = nil then
6:              if not isSetAtPodLevel then
7:                  AppendKubernetesSmell(SCRNR_UNSET)
8:              else if isValueSmellAtPodLevel then
9:                  AppendKubernetesSmell(SCRNR_VALUE)
10:             end if
11:             continue
12:         end if
13:         if container.SecurityContext.RunAsNonRoot = false then
14:             AppendKubernetesSmell(SCRNR_VALUE)
15:         end if
16:     end for
17: end procedure
```

Source: made by author (2024).

The `SmellySecurityContextCapabilities` detects potential security smells related to Linux capabilities in Kubernetes containers by analyzing the `Capabilities` field in the container's `SecurityContext`, as shown in Algorithm 3. The algorithm iterates through each container in the Pod, starting at line 2. It first checks if the `SecurityContext` or `Capabilities` field is missing, as seen in lines 3. If either is absent, it flags a security smell (`SCC_UNSET`) in line 4, indicating that no capabilities have been explicitly defined. It then examines whether the container drops all capabilities by checking the `Drop` field in `Capabilities` in line 7. If the `Drop` list is either empty or does not include `"ALL"`, it flags another security smell (`SCC_VALUE`) in line 8. This step is crucial because retaining unnecessary capabilities can increase the container's attack surface. By enforcing that all capabilities are dropped unless explicitly needed, the algorithm helps reduce the risk of privilege escalation within containers, ensuring more secure Kubernetes deployments.

Algorithm 3 - Check Security Context for Capabilities.

Algorithm 3 Check Security Context for Capabilities

```
1: procedure SMELLYSECURITYCONTEXTCAPABILITIES(spec)
2:     for each container in spec.Containers do
3:         if container.SecurityContext = nil or container.SecurityContext.Capabilities = nil
   then
4:             APPENDKUBERNETESSMELL(SCC_UNSET)
5:             continue
6:         end if
7:         if len(container.SecurityContext.Capabilities.Drop) = 0 or con-
   tainer.SecurityContext.Capabilities.Drop[0] ≠ "ALL" then
8:             APPENDKUBERNETESSMELL(SCC_VALUE)
9:         end if
10:    end for
11: end procedure
```

Source: made by author (2024).

The `SmellySecurityContextAllowPrivilegeEscalation` identifies potential security smells associated with privilege escalation settings in Kubernetes containers by examining the `AllowPrivilegeEscalation` attribute within the container's `SecurityContext`, as shown in Algorithm 4. The algorithm iterates through each container in the Pod, starting at line 2. It first checks if the `SecurityContext` or `AllowPrivilegeEscalation` field is missing, as seen in line 3. If either is absent, it flags a security smell (`SCAPE_UNSET`) in line 4, indicating that privilege escalation settings have not been explicitly defined. It then verifies whether privilege escalation is allowed by checking if `AllowPrivilegeEscalation` is set to `true` in line 7. If this setting is enabled, the algorithm flags a security smell (`SCAPE_VALUE`) in line 8. Allowing privilege escalation can increase the risk of container breakouts and other security vulnerabilities, so ensuring privilege escalation is disabled by default is a critical security best practice in Kubernetes deployments.

The `SmellySecurityContextReadOnlyRootFilesystem` detects the potential security smells related to the root filesystem settings in Kubernetes containers by examining the `ReadOnlyRootFilesystem` attribute within the container's `SecurityContext`, as shown in Algorithm 5. The algorithm iterates through each container in the Pod, starting at line 2. It first checks if the `SecurityContext` or `ReadOnlyRootFilesystem` field is missing, as seen in line 3.

Algorithm 4 - Check Security Context for AllowPrivilegeEscalation.

Algorithm 4 Check Security Context for AllowPrivilegeEscalation

```
1:  procedure SMELLYSECURITYCONTEXTALLOWPRIVILEGEESCALATION(spec)
2:      for each container in spec.Containers do
3:          if container.SecurityContext = nil or container.SecurityContext.AllowPrivilegeEscalation
    = nil then
4:              APPENDKUBERNETESSMELL(SCAPE_UNSET)
5:              continue
6:          end if
7:          if container.SecurityContext.AllowPrivilegeEscalation = true then
8:              APPENDKUBERNETESSMELL(SCAPE_VALUE)
9:          end if
10:     end for
11: end procedure
```

Source: made by author (2024).

If either is absent, it flags a security smell (`SCROFS_UNSET`) in line 4, indicating that the root filesystem configuration has not been explicitly defined. It then checks whether the root filesystem is set to be read-only by verifying if `ReadOnlyRootFilesystem` is set to `false` in line 7. If this setting allows write access, the algorithm flags a security smell (`SCROFS_VALUE`) in line 8. Ensuring that the root filesystem is mounted as read-only helps mitigate the risk of unauthorized modifications and enhances the overall security posture of Kubernetes deployments.

Algorithm 5 - Check Security Context for ReadOnlyRootFilesystem.

Algorithm 5 Check Security Context for ReadOnlyRootFilesystem

```
1:  procedure SMELLYSECURITYCONTEXTREADONLYROOTFILESYSTEM(spec)
2:      for each container in spec.Containers do
3:          if container.SecurityContext = nil or container.SecurityContext.ReadOnlyRootFilesystem
    = nil then
4:              APPENDKUBERNETESSMELL(SCROFS_UNSET)
5:              continue
6:          end if
7:          if container.SecurityContext.ReadOnlyRootFilesystem = false then
8:              APPENDKUBERNETESSMELL(SCROFS_VALUE)
9:          end if
10:     end for
11: end procedure
```

Source: made by author (2024).

The `SmellySecurityContextPrivileged` identifies potential security smells associated with privileged container settings in Kubernetes by examining the `Privileged` attribute within each container's `SecurityContext`, as shown in Algorithm 6. The algorithm iterates through each container in the Pod, starting at line 2, and checks if the `SecurityContext` and `Privileged` fields are present, as seen in line 3. If the `Privileged` attribute is explicitly set to `true`, indicating that the container has elevated privileges, the algorithm flags a security smell (`SCP_VALUE`) in line 4. Privileged containers have broad access to host resources, which can increase the risk of security vulnerabilities. Therefore, ensuring containers are not unnecessarily granted privileged access is essential for maintaining secure Kubernetes deployments.

Algorithm 6 - Check Security Context for Privileged.

Algorithm 6 Check Security Context for Privileged

```
1: procedure SMELLYSECURITYCONTEXTPRIVILEGED(spec)
2:     for each container in spec.Containers do
3:         if container.SecurityContext ≠ nil and container.SecurityContext.Privileged ≠ nil
   and container.SecurityContext.Privileged = true then
4:             APPENDKUBERNETESSMELL(SCP_VALUE)
5:         end if
6:     end for
7: end procedure
```

Source: made by author (2024).

The `SmellyResourceAndLimit` assesses the configuration of resource requests and limits in Kubernetes containers to identify potential issues related to resource management, as shown in Algorithm 7. The algorithm iterates through each container in the Pod, starting at line 2, and checks for the presence of resource requests and limits within the `Resources` field. If the `Requests` attribute is missing, as seen in line 3, it flags a security smell (`RR_UNSET`) in line 4, indicating that no minimum resource requirements have been defined. Similarly, if the `Limits` attribute is absent, as seen in line 6, it flags a security smell (`RL_UNSET`) in line 7, signaling that no maximum resource constraints are set. Properly defining both resource requests and limits is crucial for ensuring stable and predictable performance in Kubernetes environments, as it helps prevent resource contention and potential system overloads.

Algorithm 7 - Check Resource Requests and Limits.

Algorithm 7 Check Resource Requests and Limits

```
1:  procedure SMELLYRESOURCEANDLIMIT(spec)
2:      for each container in spec.Containers do
3:          if container.Resources.Requests = nil then
4:              APPENDKUBERNETESSMELL(RR_UNSET)
5:          end if
6:          if container.Resources.Limits = nil then
7:              APPENDKUBERNETESSMELL(RL_UNSET)
8:          end if
9:      end for
10: end procedure
```

Source: made by author (2024).

4.3 Client

As the server is responsible for conducting security analysis, like a REST API, the client can be made by any text editor that allows the user to develop extensions or functionalities in it, using HTTP request handling. The client used in this work is a VS Code extension, also called a plugin, and it provides an easy way to register custom commands and events necessary for the Smelly Kube.

This VS Code extension was written in TypeScript. The foundational code structure for its development was provided by Yeoman[7] and Yo Code[8], which generate the base code for developing the extension. The libraries used are VS Code, Node Fetch for handling HTTP requests, and `dotenv` for managing environment variables.

The first step to using the extension is to build and run it through VS Code. Therefore, this process necessitates the configuration of the API_URL environment variable to specify the server endpoint. When the extension is running, the user only needs to open the desired Kubernetes manifest file and execute the command Inspect File. This command will make a POST request to the server, sending the file content on the request body. The client's behavior can be seen in the next section, handling the server response on the Visual Studio Code side.

[7] `https://yeoman.io/`. Accessed on June 21, 2024

[8] `https://www.npmjs.com/package/generator-code`. Accessed on June 21, 2024

4.4 Smelly Kube usage example

In this section is presented a Smelly Kube usage example. A Kubernetes manifest will be used as described in Listing 4.1, to show the VS Code plugin behavior. The manifest contains a few security smells based on the catalog described earlier in this chapter.

Listing 4.1 – Kubernetes Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: http-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: http-server
  template:
    metadata:
      labels:
        app: http-server
    spec:
      serviceAccountName: http-server-sa
      containers:
      - name: http-server
        image: python:3.9-slim
        ports:
        - containerPort: 8000
        command: [ "python", "http_server_builtin.py" ]
```

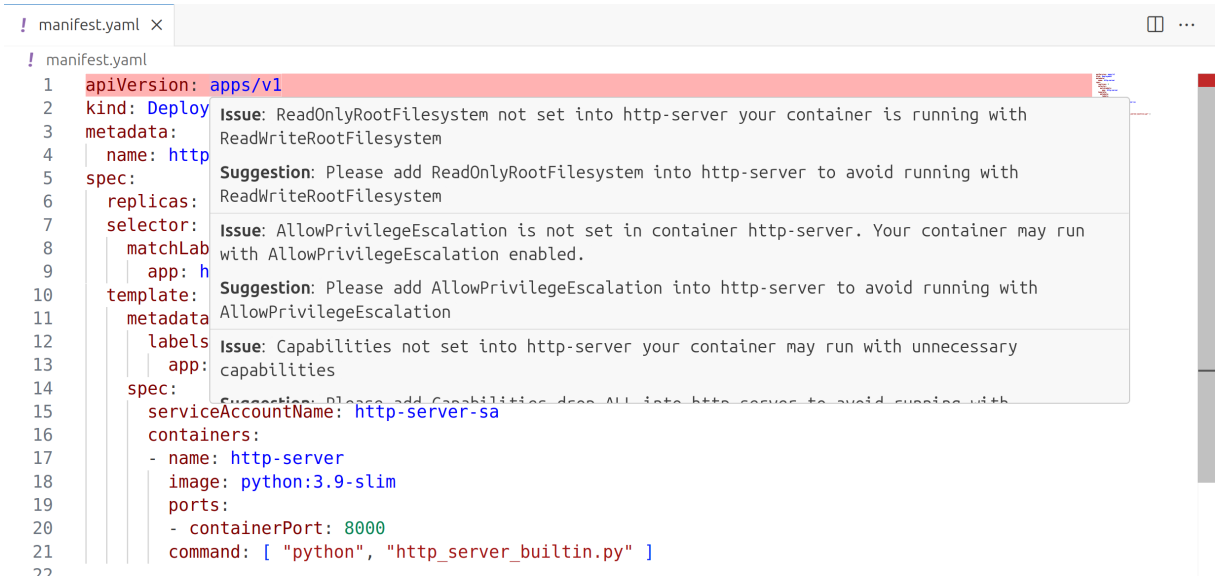
Source: made by author (2024).

This Kubernetes manifest, named `http-server`, defines a Deployment workload with one replica and matching the label `app: http-server`. It also sets a ServiceAccount `http server-sa` in the Pod definition, containing one container named by `http-server`. This container uses the `python:3.9-slim` image, exposes port 8000, and sets the entry point for initializing the Python application server.

Figure 4.5 shows the client behavior when parsing the response from the server. If the response returns an error code, the extension will show a pop-up telling the error string. Otherwise, it will show a pop-up with the number of security smells found on the manifest, and color the first line of the affected workload in red. Also, if the user hovers the mouse cursor on these colored lines, it OPENS a text box containing the list of smells found on that specific workload.

Finally, this chapter introduced Smelly Kube with its architecture and application flow, security rules and their sources, algorithms that implement these rules, and a case study. The next chapter presents the methodology of this work, including the collected metrics to evaluate the results and the tool's effectiveness, the test scenarios explaining why GitHub and Artifact Hub were chosen, and the test flow.

Figure 4.5 – VS Code plugin threatening the security smells found.



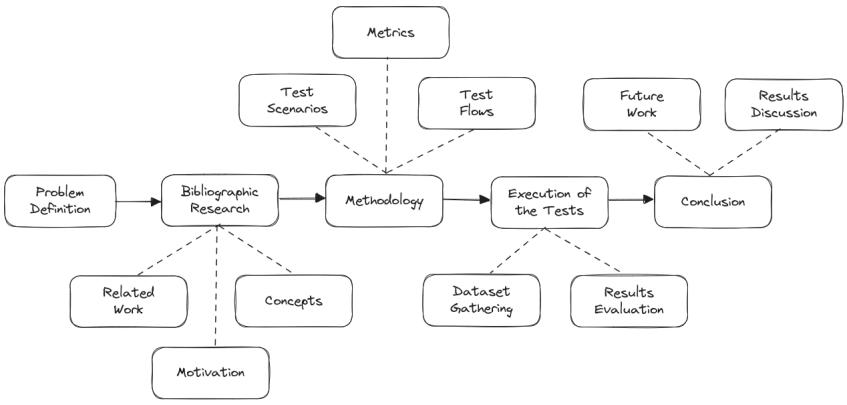
Source: made by author (2024).

5 METHODOLOGY

The research steps are described in Figure 5.1. This process encompasses an exploratory bibliographic analysis, drawing from various sources including scholarly articles, periodicals, and academic publications. Additionally, the bibliography will incorporate technical materials from vendors and official websites, ensuring a comprehensive understanding of the subject matter.

According to Jung (2004), this work methodology fills the experimental development research by conducting experiments in real-world scenarios to answer the dissertation hypothesis. In addition, the exploratory research could match the intentions of this study, as experimental results can open new research questions and discussions and be a preliminary study for future work.

Figure 5.1 – Work overview.



Source: made by author (2022).

In order to identify and analyze security smells in Kubernetes manifests using Smelly Kube, this work methodology was divided into the definition of the evaluated metrics, the experimental scenarios, the data collection process, and the execution of the experimental tests.

5.1 Metrics

This section outlines the key metrics used to evaluate the effectiveness of this study. These metrics are designed to provide a comprehensive view of the proportion of security smells in Kubernetes workloads, and the distribution of these smells across different categories. Tracking metrics such as the time taken for testing, the number of valid projects, and the occurrence of security smells aims to offer clear insights into the security posture of Kubernetes applica-

tions in the examined projects. These metrics form the basis for analyzing and comparing the security robustness of various workloads and projects.

Measuring the time taken, CPU usage, and memory usage metrics requires a thorough understanding of the hardware specifications used during the tests and how these metrics were collected. For instance, the test runs on a machine with 16 GB of RAM, with a Ryzen 7 7700 3.8 GHz processor. The Operating System used, and the application requirements and versions can be found in Appendix A. By conducting the work experiments on deploying Smelly Kube in a Docker container, collecting these metrics with `docker stats`[1] was an easy task, as it tracks the resources consumption only for the container PID[2].

a) **Time taken**: tracks the total time spent conducting the tests;
b) **CPU usage**: collects both the peak and mean CPU usage during the tests to understand the resource consumption over time;
c) **Memory usage**: records the peak and mean memory usage throughout the testing to evaluate memory efficiency and detect potential memory bottlenecks;
d) **Total projects and valid projects**: tells the number of projects provided by the dataset, and the projects with at least one valid Kubernetes manifests;
e) **Workloads with at least one security smell found**: identifies the Kubernetes workloads (e.g., Pods, Deployments) that have at least one security smell detected, in comparison with the overall decoded workloads;
f) **Security smells found by workload**: categorizes the total number of security smells identified in different types of workloads. It provides a percentage indicating how many security smells are present in each workload relative to the total amount of security smells;
g) **Security smells found by category**: breaks down the identified security smells into specific categories (e.g., resource requests, user permissions). It includes a percentage that shows the proportion of each category against the total number of security smells found;
h) **Security smells density**: measures the occurrence of security smells per 1,000 lines of code in the analyzed Kubernetes manifests. This metric helps quantify the prevalence of these smells relative to code size;

[1] `https://docs.docker.com/reference/cli/docker/container/stats/`. Accessed on August 10, 2024
[2] PID is a unique number assigned to each process running in an operating system

i) **Most prevalent security smell per workload**: identifies the most frequently occurring security smell in each workload type (e.g., Pods, Deployments). This metric highlights which security smell is most common in different workload configurations.

5.2 Experiment Scenarios

Two primary platforms, Artifact Hub and GitHub, were chosen for the experiment scenarios to evaluate Kubernetes security within OSS and real-world contexts. These platforms collectively offer a rich, diverse set of Kubernetes manifests, which include reusable cloud-native artifacts and configurations commonly used in production environments. By focusing on Artifact Hub and GitHub, the scenarios leverage large repositories of Kubernetes resources, allowing for a comprehensive analysis of security practices in infrastructure definitions.

Both platforms facilitate access to Helm charts and other Kubernetes-related packages, enabling a practical examination of how well security standards are maintained across a wide range of Kubernetes manifests. This setup provides the basis for assessing security flaws, potential misconfigurations, and adherence to best practices within Kubernetes infrastructure definitions.

5.2.1 Artifact Hub

Artifact Hub[3] is a centralized repository designed to help developers and operators discover, install, and manage cloud-native packages such as Helm charts[4], Kubernetes operators, Falco rules, and OPA policies. It is a platform where users can search for reusable cloud-native artifacts, such as Helm charts, install them, and publish their artifacts (CHEN et al., 2023). Developed by the Cloud Native Computing Foundation (CNCF), Artifact Hub is considered the largest repository of Helm charts (ZEROUALI; OPDEBEECK; ROOVER, 2023).

Helm Chart[5] is a package of pre-configured Kubernetes resources used to deploy and manage applications in Kubernetes clusters (GOKHALE et al., 2021). It acts like a template that bundles all the necessary components—such as deployments, services, and configurations—into a single, reusable package. Helm charts allow users to define the structure of

[3] `https://artifacthub.io/`. Accessed on August 2024
[4] `https://helm.sh/`. Accessed on August 2024
[5] `https://helm.sh/`. Accessed on August 2024

their application and provide a way to deploy it easily and consistently across different environments.

Many types of Cloud Native packages are stored on Artifact Hub, such as Argo templates, OLM Operators, OPA Policies, and more. To extract only Helm packages used as the dataset for this study, it was needed to call a specific Artifact Hub API endpoint `/api/v1/helm-exporter`, iterate over each package, and merge the Helm template with the Helm values. For more details, see Section A.3.

5.2.2 GitHub

Software forges are collaborative, web-based platforms designed to facilitate distributed development, making them particularly valuable for Open Source Software (OSS) projects. GitHub[6] represents the latest generation of these forges, combining traditional features, such as free hosting and version control, with social functionalities that enhance collaboration and community engagement (SQUIRE, 2014).

The platform has seen a surge in popularity. Since its launch in 2008, the number of hosted projects and users has grown exponentially, increasing from 135,000 projects in 2009 to over 35 million by 2015 (COSENTINO; IZQUIERDO; CABOT, 2017). These OSS forges, such as GitHub and GitLab, have already been used to conduct research in Kubernetes security (BOSE; RAHMAN; SHAMIM, 2021; RAHMAN et al., 2023).

To make the GitHub dataset, a JavaScript program was made to retrieve the repositories by calling `/search/repositories` and `/search/code` endpoints. The dataset will contain a directory named by the repository name, and that directory contains the Kubernetes manifests. Since GitHub does not contain only Kubernetes manifests, some filters needed to be applied during the extraction. The filters include (*i*) search query for common Kubernetes manifest extensions like `extension:yaml` and `extension:yml`, and (*ii*) apply the regex[7] expressions described below for each file content in the repository:

a) `/apiVersion:\s*(\S+)\s*\n/`

b) `/kind:\s*(\S+)\s*\n/`

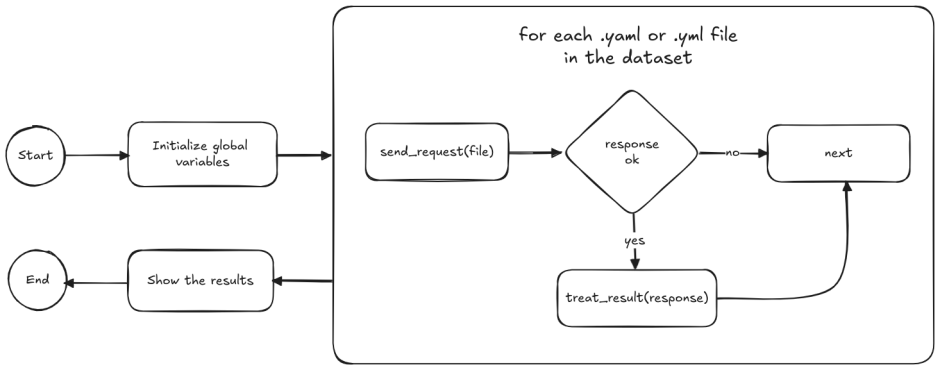
[6] `https://github.com/`. Accessed on August 2024

[7] Regex, or regular expression, is a sequence of characters that defines a search pattern, used for matching and manipulating strings in text processing.

5.3 Metrics Evaluation

The execution of the tests with Smelly Kube and metrics evaluation will be run in the datasets described in the previous section. Figure 5.2 shows the flowchart used to conduct this process. Even though each dataset iteration is different, as the GitHub dataset requires a recursive traversal of the directory tree, the process applied to each file is the same.

Figure 5.2 – Metrics evaluation flowchart.



Source: made by author (2024).

First, the global variables are initialized with their default values, which will accumulate the results processed by the program. Past this initialization `process_yaml_files` is called, processing each file in the dataset. If the filename ends with a manifest extension, in this case, `.yaml` or `.yml`, the file is sent to Smelly Kube API when `send_file` is called. If the HTTP response code is OK, the response is passed through `treat_result` function to accumulate the test results from the response body. Next, the number of lines in the Kubernetes manifest is determined by iterating over each line. The whitespace and non-visible characters are removed from it. Then, a line is counted only if it meets the following criteria: (*i*) it is not empty, and (*ii*) it does not begin with a comment character. Finally, the program shows the final results obtained by processing the dataset.

The `treat_result` function receives the Smelly Kube API response body as an argument, containing report metadata and data for a given manifest entry (see Section 4.2.1 about the response body structure). First, each decoded workload is looped through a `for each` structure to accumulate the number of decoded workloads. Later, there is a `for each` to iterate over each workload in the report data. In this loop, security vulnerabilities are accumulated, and if the total exceeds zero, the count of affected workloads is incremented. Also, each workload smell is iterated through a loop, incrementing the security smells found for a smell category, and

incrementing the security smells found for a smell category associated with the given workload. For more details about this algorithm, see Appendix A.4.

This chapter detailed the research process, including the definition of real-world experimental testing scenarios to answer some of the research questions. It described the use of Artifact Hub and GitHub as platforms for data collection, explained the metrics for evaluating Kubernetes security, and outlined the testing flow using Smelly Kube to analyze YAML manifests. The next chapter will present the experimental results, detailing the execution of tests and analyzing the findings obtained through these methodologies.

6 EXPERIMENTS AND RESULTS

This chapter describes the experiments and results. The study aims to investigate whether the repositories available in the largest repository of Cloud Native packages (Artifact Hub) and GitHub contain the security smells defined in this work.

The experiment was conducted by compiling a diverse dataset of Helm chart repositories from Artifact Hub's API, resulting in an initial collection of 5,055 Kubernetes manifests. To broaden the scope and depth of the analysis, an additional dataset was acquired from GitHub, contributing 183,225 manifests. Together, these sources form a comprehensive dataset that spans a wide range of Kubernetes configurations, providing a substantial basis for identifying and analyzing security smells in real-world projects of varied sizes and complexities. This dual-sourcing method ensures a representative sample of contemporary Kubernetes deployments, yielding valuable insights into security at scale.

Across both sources, the initial counts of total and valid manifests varied significantly. While the Artifact Hub dataset provided 5,055 projects and manifests, with 2,107 valid ones, GitHub contained 3,408 projects but yielded a much larger 183,225 manifests, of which 34,191 were valid. For a Kubernetes manifest to be considered valid, it had to meet specific criteria: it must not be empty, malformed, or a Helm template. For a project to be considered valid, at least one manifest must be valid.

The processing times and resource consumption differed notably between platforms. Analyzing Artifact Hub's manifests took only 27 seconds, with an average CPU usage of 28.64% and peak usage reaching 98.46%. In contrast, the larger GitHub dataset required 9 minutes and 14 seconds, with an average CPU usage of 56.74% and a peak of 98.12%. Memory usage was efficient on both platforms, though Artifact Hub required slightly more (average of 22.87 MiB, peaking at 38.89 MiB) compared to GitHub (average of 19.61 MiB, peaking at 25.22 MiB). As Artifact Hub manifests encompass larger manifest files, which could use more server memory to process them, it could justify the higher memory usage rather than GitHub manifests. Table 6.1 shows a comparison between these metrics.

Table 6.1 – Initial metrics comparison.

	Artifact Hub	**GitHub**
Total projects	5,055	3,408
Valid projects	2,107	2,253
Total manifests	5,055	183,225
Valid manifests	2,107	34,191
Time taken	27 seconds	9 minutes and 14 seconds
Mean CPU usage	28.64%	56.74%
Peak CPU usage	98.46%	98.12%
Mean memory usage	22.87 MiB	19.61 MiB
Peak memory usage	38.89 MiB	25.22 MiB

Source: made by author (2024).

6.1 Presence of Security Smells in Kubernetes Manifests

This section analyzes the prevalence of different Kubernetes workloads, declared in the manifests, that may be affected by at least one security smell, from Artifact Hub and GitHub datasets. It aims to answer the $\mathbf{RQ}_1$: *how frequently do security smells occur in Kubernetes manifests?*

6.1.1 Artifact Hub

Table 6.2 and Figure 6.1 present a summary of workloads that had at least one security smell found. The total number of decoded workloads is 4,549, of which 2,941 were affected by at least one security smell, representing 64.65% of the total. The breakdown by workload type reveals notable trends: Pods had the highest percentage of affected workloads by security smells, with 547 out of 619 instances (88.37%) affected. DaemonSets also exhibited a significant presence of security smells, affecting 122 of 149 instances (81.88%).

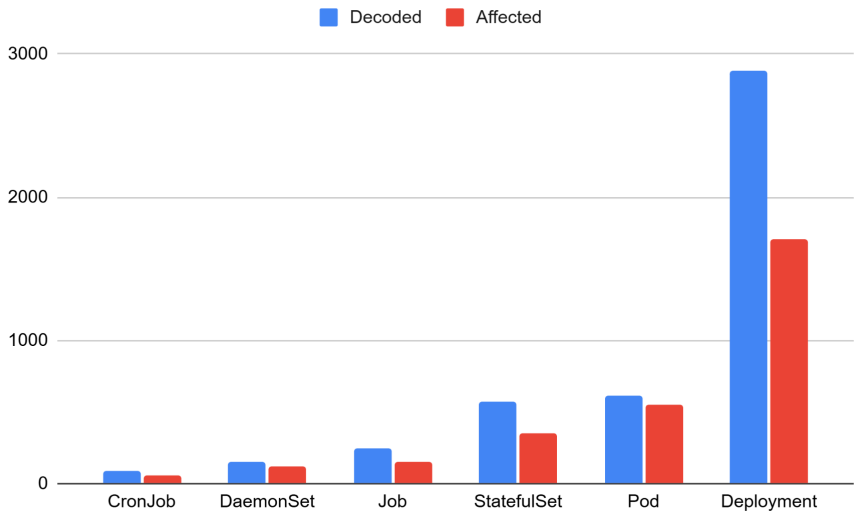
Deployments, one of the most widely used workload types, showed vulnerabilities in 1,708 out of 2,883 instances, marking a 59.24% impact rate. CronJobs and Jobs, used for scheduled and batch processing tasks, were impacted in 70.24% and 63.22% of cases, respectively. StatefulSets, which manage applications requiring persistent state, had 61.54% of instances

Table 6.2 – Artifact Hub - Workloads that had at least one security smell found.

	Decoded	**Affected**	**Decoded/Affected**
CronJob	84	59	70.24%
DaemonSet	149	122	81.88%
Deployment	2883	1708	59.24%
Job	242	153	63.22%
Pod	619	547	88.37%
ReplicaSet	0	0	0%
StatefulSet	572	352	61.54%
Total	**4549**	**2941**	**64.65%**

Source: made by author (2024).

Figure 6.1 – Artifact Hub - Affected Workloads.



Source: made by author (2024).

showing security smells, suggesting specific risks associated with stateful workload management. No security smells were identified in ReplicaSets, as the dataset did not contain any instances of this workload.

6.1.2 GitHub

Table 6.3 and Figure 6.2 present a summary of workloads that had at least one security smell found. Out of 42,798 decoded workloads, 33,913—approximately 79.24%—were found

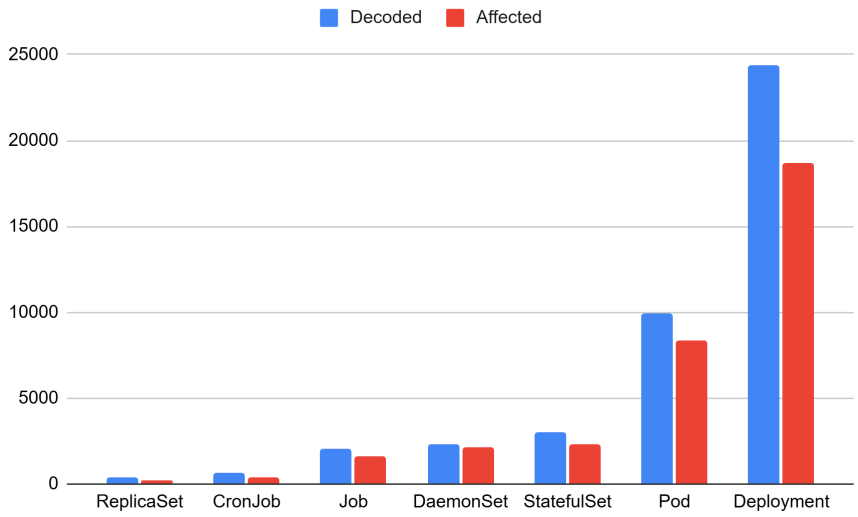
to have at least one security smell. This high percentage indicates that a significant portion of workloads is affected by these smells.

Table 6.3 – GitHub - Workloads that had at least one security smell found.

	Decoded	**Affected**	**Decoded/Affected**
CronJob	663	396	59.73%
DaemonSet	2346	2175	92.71%
Deployment	24398	18723	76.74%
Job	2049	1646	80.33%
Pod	9936	8386	84.40%
ReplicaSet	374	234	62.57%
StatefulSet	3032	2353	77.61%
Total	**42798**	**33913**	**79.24%**

Source: made by author (2024).

Figure 6.2 – GitHub - Affected Workloads.



Source: made by author (2024).

Among these, DaemonSets exhibited the highest rate of security smells, with 2,175 out of 2,346 instances affected, representing 92.71%. Pods followed closely, with 8,386 out of 9,936 instances showing security smells, resulting in an impact rate of 84.40%. Deployments, which are extensively used for managing applications, had 18,723 affected instances out of 24,398, marking a substantial 76.74% rate of security smells.

Jobs, that handle batch processing tasks, showed security smells in 1,646 out of 2,049 instances (80.33%). Similarly, StatefulSets, known for managing stateful applications, had 2,353 of 3,032 instances affected, yielding a 77.61% rate. CronJobs, used for scheduled task execution, had a lower rate compared to other types, with 396 out of 663 instances (59.73%) displaying security smells. ReplicaSets, which ensure a specified number of pod replicas are running at any given time, had the lowest observed rate at 62.57%, affecting 234 of 374 instances.

6.1.3 Results Discussion

The findings indicate that security smells are prevalent in Kubernetes manifests across Artifact Hub and GitHub. For Artifact Hub, 64.65% of workloads were affected by at least one security smell, with the highest percentage seen in Pods (88.37%). GitHub data showed a higher overall rate, with 79.24% of workloads affected, and DaemonSets reaching 92.71%. Deployment was the most present decoded workload in both scenarios, but Artifact Hub had more maturity in terms of being affected by at least one security smell (59.24%) rather than GitHub (76.74%).

6.2 Security Smells Across Workloads

This section provides an analysis of the distribution of security smells across various Kubernetes workloads, highlighting descriptive statistics and identifying potential areas for improved security attention. This section answers the $\mathbf{RQ}_2$: *what is the proportion of security smells by Kubernetes workloads?*

6.2.1 Artifact Hub

Table 6.4 summarizes the distribution of identified security smells across different Kubernetes workloads in the Artifact Hub dataset. A total of 27,967 security smells were found, distributed among various types of workloads as follows: Pods account for 4,261 security smells, representing 15.24%. Deployments contain the majority of security smells, with a total

of 17,638, which constitutes 63.07% of the total. Jobs, exhibit 1,413 security smells, making up 5.05% of the total. Although Jobs are less frequent, they still present notable security concerns.

Table 6.4 – Artifact Hub - Security smells found by workload.

	Security smells	**Total/Workload**
CronJob	513	1,83%
DaemonSet	1299	4.64%
Deployment	17638	63.07%
Job	1413	5.05%
Pod	4261	15.24%
ReplicaSet	0	0%
StatefulSet	2843	10.17%
Total	**27967**	**100%**

Source: made by author (2024).

CronJobs has 513 security smells, accounting for 1.83% of the total, and DaemonSets has 1,299, or 4.64%. StatefulSets presents 2,843 security smells, making up 10.17% of the total, while ReplicaSets, in this dataset, contains no identified security smells as there were no ReplicaSets in the dataset. The distribution of security smells across these workloads highlights areas that may require specific security attention in Kubernetes environments.

6.2.2 GitHub

Table 6.5 provides a summary of the distribution of identified security smells across different Kubernetes workloads in the GitHub dataset. A total of 282,963 security smells were detected.

Deployments account for the highest number of security smells, with 154,655 instances, making up 54.66% of the total. This large share suggests that the complexity of managing replicas and scaling in Deployments can lead to significant security challenges. Pods, as the basic deployable units, follow with 70,284 security smells, representing 24.84% of the total. This emphasizes that Pods, despite their simpler structure, are not immune to security vulnerabilities.

Table 6.5 – GitHub - Security smells found by workload.

	Security smells	**Total/Workload**
CronJob	4747	1.68%
DaemonSet	18418	6.51%
Deployment	154655	54.66%
Job	11861	4.19%
Pod	70284	24.84%
ReplicaSet	1766	0.62%
StatefulSet	21232	7.50%
Total	**282963**	**100%**

Source: made by author (2024).

DaemonSets display 18,418 security smells, accounting for 6.51% of the total. The continuous nature of DaemonSets, ensuring a copy runs on each node, could contribute to this share. StatefulSets, which manage stateful applications, exhibit 21,232 security smells, making up 7.50% of the total, showing that managing state brings its own set of security concerns.

Jobs, utilized for batch processing tasks, show 11,861 security smells, which is 4.19% of the total, indicating that even infrequent tasks require attention to security. CronJobs, responsible for scheduled jobs, account for 4,747 security smells or 1.68% of the total, while ReplicaSets have the fewest identified security smells at 1,766, making up only 0.62%.

6.2.3 Results Discussion

The results reveal that security smells are prevalent across different workloads. In Artifact Hub, Deployments had the highest percentage of security smells (63.07%), followed by Pods (15.24%) and StatefulSets (10.17%). In GitHub, Deployments also led with 54.66% of the total security smells, with Pods (24.84%) and StatefulSets (7.50%) contributing significantly. This distribution highlights that different Kubernetes workloads on Artifact Hub and GitHub have varying smells, with Deployments and Pods as the primary contributors. This analysis suggests enhanced security measures and scrutiny should focus on these workloads to minimize potential risks.

6.3 Security Smells by Security Smell Categories

This section provides insights into which types of security smells are most common, emphasizing the need for stringent adherence to security best practices to safeguard Kubernetes environments. The showcase includes highlighting both the absence of crucial configurations, denoted by `_UNSET` suffix, and the presence of non-compliant values in existing configurations, denoted by `_VALUE` suffix, representing a forced misconfiguration that introduces potential security risks. It aims to answer the **RQ**$_3$: *what is the proportion of security smells by security smell categories?* and **RQ**$_4$: *what is the density of security smells?*

6.3.1 Artifact Hub

The Table 6.6 highlights the findings from analyzing security smells in Artifact Hub, categorized by different types of security smells as detected by Smelly Kube.

Table 6.6 – Artifact Hub - Security smells found by security smell category.

	Security smells	**Total/Smell**
SCRNR_VALUE	22	0.08%
SCAPE_VALUE	23	0.08%
SCRU_VALUE	44	0.16%
SCC_VALUE	83	0.30%
SCROFS_VALUE	94	0.34%
SCP_VALUE	140	0.50%
RR_UNSET	3434	12.28%
RL_UNSET	3597	12.86%
SCRU_UNSET	3686	13.18%
SCRNR_UNSET	4021	14.38%
SCAPE_UNSET	4217	15.08%
SCC_UNSET	4244	15.18%
SCROFS_UNSET	4362	15.60%
Total	**27967**	**100%**

Source: made by author (2024).

The data indicates that the most prevalent security smells are associated with unset configurations, particularly with the ‘Security Context Capabilities Unset (SCC_UNSET)’ and ‘Security Context Read-Only Root File System Unset (SCROFS_UNSET)’, each representing 15.18% and 15.60% of the total findings, respectively. These results emphasize the importance of defining proper security contexts and limiting unnecessary permissions to enhance the overall security posture. Conversely, security smells linked to non-compliant values, such as ‘Security Context Privileged Value (SCP_VALUE)’—indicating configurations set with insecure values like `privileged: true`—were found less frequently.

Table 6.7 highlights the density of security smells detected in Artifact Hub, measured per 1,000 lines of code. This dataset contains 1,495,236 lines of code. Smells related to specific values, such as ‘Security Context Run As Non-Root Value (SCRNR_VALUE)’ and ‘Security Context Add Capabilities Value (SCAPE_VALUE)’, exhibit low densities, all below 0.1. By the other side, unset configurations, such as ‘Security Context Read-Only Root File System Unset (SCROFS_UNSET)’ and ‘Security Context Capabilities Unset (SCC_UNSET)’, present significantly higher densities, exceeding 2.8 per 1,000 lines.

Table 6.7 – Artifact Hub - Security smells density per 1,000 lines of code.

	Density
SCRNR_VALUE	0.01
SCAPE_VALUE	0.01
SCRU_VALUE	0.02
SCC_VALUE	0.05
SCROFS_VALUE	0.06
SCP_VALUE	0.09
RR_UNSET	2.29
RL_UNSET	2.4
SCRU_UNSET	2.46
SCRNR_UNSET	2.68
SCAPE_UNSET	2.82
SCC_UNSET	2.83
SCROFS_UNSET	2.91

Source: made by author (2024).

This suggests that while configurations are often set, their proper specification still requires attention to avoid security risks. The analysis visually depicts the distribution of these security smells by category, further supporting the need for thorough security audits and adherence to best practices when configuring Kubernetes manifests.

6.3.2 GitHub

Table 6.8 shows the distribution of security smells detected in the GitHub dataset, categorized by different security configurations. The most frequently occurring security smells were related to unset configurations, with 'Security Context Read-Only Root File System Unset (SCROFS_UNSET)' being the most common at 15.37% of the total findings, followed by 'Security Context Capabilities Unset (SCC_UNSET)' at 14.83%.

Table 6.8 – GitHub - Security smells found by security smell category.

	Security smells	**Total/Smell**
SCRNR_VALUE	177	0.06%
SCROFS_VALUE	179	0.06%
SCAPE_VALUE	432	0.15%
SCRU_VALUE	491	0.17%
SCC_VALUE	901	0.32%
SCP_VALUE	1953	0.69%
RR_UNSET	33951	12.00%
RL_UNSET	35734	12.63%
SCRNR_UNSET	40981	14.48%
SCRU_UNSET	41292	14.59%
SCAPE_UNSET	41399	14.63%
SCC_UNSET	41970	14.83%
SCROFS_UNSET	43503	15.37%
Total	**282963**	**100%**

Source: made by author (2024).

This highlights a significant gap in securing Kubernetes manifests, as key configurations are often missing. On the other hand, security smells associated with non-compliant values,

such as ‘Security Context Privileged Value (SCP_VALUE)’ — which implies that containers are running in privileged mode — were less common but still notable at 0.69%.

Table 6.9 highlights the density of security smells detected in GitHub repositories, measured per 1,000 lines of code. The dataset contains 4,453,214 lines of code. Similar to the findings from Artifact Hub, unset configuration smells dominate the dataset, with ‘Security Context Read-Only Root File System Unset (SCROFS_UNSET)’ reaching the highest density at 9.76. Other unset configurations, such as ‘Security Context Capabilities Unset (SCC_UNSET)’ and ‘Security Context Run As Non-Root Unset (SCRNR_UNSET)’, also exhibit high densities, exceeding 9.0 per 1,000 lines. In contrast, security smells related to specific values, such as ‘Security Context Privileged Value (SCP_VALUE)’, have significantly lower densities, with the highest in this category reaching only 0.43.

Table 6.9 – GitHub - Security smells density per 1,000 lines of code.

Security Smell	**Density**
SCRNR_VALUE	0.03
SCROFS_VALUE	0.04
SCAPE_VALUE	0.09
SCRU_VALUE	0.11
SCC_VALUE	0.2
SCP_VALUE	0.43
RR_UNSET	7.62
RL_UNSET	8.02
SCRNR_UNSET	9.2
SCRU_UNSET	9.27
SCAPE_UNSET	9.29
SCC_UNSET	9.42
SCROFS_UNSET	9.76

Source: made by author (2024).

6.3.3 Results Discussion

The analysis shows that the most prevalent types of security smells are SCROFS_UNSET (15.60%, 2.91 per 1,000 LOC[1]), SCC_UNSET (15.18%, 2.83 per 1,000 LOC), and SCAPE_UNSET

[1] LOC: Lines Of Code

(15.08%, 2.82 per 1,000 LOC) for Artifact Hub. Similarly, GitHub data reflects that SCROFS_UNSET (15.37%, 9.76 per 1,000 LOC), SCC_UNSET (14.83%, 9.42 per 1,000 LOC), and SCAPE_UNSET (14.63%, 9.29 per 1,000 LOC) are the most common security smells. Both sources highlight that unset security attributes are significant contributors, indicating common lapses in configuring essential security settings that should be enforced. Also, the results suggest that while configurations are often present, their security implications need careful attention to prevent misconfigurations that could be exploited. In this case, the smell SCP_VALUE (0.50%, 0.09 per 1,000 LOC for Artifact Hub, and 0.69%, 0.43 per 1,000 LOC for GitHub occurrences) was the most present among value-related smells, meaning these containers would run in privilege mode with unnecessary permissions.

6.4 Most Prevalent Security Smell per Workload

This section describes the most prevalent security smells within different Kubernetes workloads for both ArtifactHub and GitHub datasets. The analysis helps prioritize mitigation efforts and provides a targeted approach to improving the security of Kubernetes manifests. This section aims to answer the **RQ**$_5$: *what is the most prevalent security smell per workload?*

6.4.1 Artifact Hub

Table 6.10 presents the most prevalent security smells identified in different Kubernetes workloads from the Artifact Hub dataset. Within the Deployment workload, the most prevalent security smell is SCROFS_UNSET, appearing 2,759 times. Similarly, the StatefulSet workload exhibits the same security smell with 534 occurrences. In the case of DaemonSets, the most frequent smell is SCRNR_UNSET, with 187 instances, while RR_UNSET is the most common in Pods, with 617 cases. Jobs and CronJobs predominantly exhibit RL_UNSET (229 occurrences) and SCC_UNSET (75 occurrences), respectively. Notably, ReplicaSet is marked as "Not applicable" because there was no occurrence of this workload in the dataset.

Table 6.10 – Artifact Hub - Most prevalent security smell per workload.

Workload	Security smell	Quantity
CronJob	SCC_UNSET	75
DaemonSet	SCRNR_UNSET	187
Deployment	SCROFS_UNSET	2759
Job	RL_UNSET	229
Pod	RR_UNSET	617
ReplicaSet	Not applicable	0
StatefulSet	SCROFS_UNSET	534

Source: made by author (2024).

6.4.2 GitHub

Table 6.11 presents the most prevalent security smells identified in different Kubernetes workloads from the GitHub dataset. Within the Deployment workload, the most prevalent security smell is SCROFS_UNSET, appearing 23,831 times. The same security smell is also predominant in StatefulSets, with 3,417 occurrences, and in Pods, with 10,720 occurrences. DaemonSets also exhibit SCROFS_UNSET as the most frequent issue, with 2,786 instances, while CronJobs shows 698 occurrences of this smell. In the case of Jobs, the most prevalent security smell is SCC_UNSET, with 1,796 occurrences, while ReplicaSets exhibit the same smell with 261 occurrences.

Table 6.11 – GitHub - Most prevalent security smell per workload.

Workload	Security smell	Quantity
CronJob	SCROFS_UNSET	698
DaemonSet	SCROFS_UNSET	2786
Deployment	SCROFS_UNSET	23831
Job	SCC_UNSET	1796
Pod	SCROFS_UNSET	10720
ReplicaSet	SCC_UNSET	261
StatefulSet	SCROFS_UNSET	3417

Source: made by author (2024).

6.4.3 Results Discussion

The analysis reveals contrasting patterns between the Artifact Hub and GitHub datasets. In GitHub, SCROFS_UNSET and SCC_UNSET dominate, with Deployments showing 23,831 occurrences of SCROFS_UNSET, while Jobs and ReplicaSets frequently exhibit SCC_UNSET. In contrast, Artifact Hub presents a spread distribution of security smells across workloads, though SCROFS_UNSET remains the most common in Deployments (2,759 occurrences). The higher concentration of specific smells in GitHub suggests frequent misconfigurations in user-contributed manifests, whereas Artifact Hub's diversity indicates varied security gaps. These findings highlight the need for targeted security improvements, prioritizing Deployments while addressing workload-specific security smells.

Table 6.12 – Artifact Hub - Top 10 lowest and highest projects.

Top 10 lowest		**Top 10 highest**	
Project	**Security smells**	**Project**	**Security smells**
janusgraph	0	oes	180
pagerduty-exporter	0	sciencebox	163
azure-scheduledevents-manager	0	ddosify	161
hnc	0	galoy	153
mailpit	0	smile	145
flux	0	enav	140
speckle-server-branch-testing	0	rhacs-demo-applications	134
minecraft-exporter	0	anteon	133
azure-keyvault	0	opentelemetry-demo	126
jiralert	0	stackstorm-ha	124

Source: made by author (2024).

6.5 Project Ranking

This section presents an overview of the top 10 projects with the lowest and highest number of security smells for both datasets, highlighted by Table 6.12 for Artifact Hub and Table 6.13 for GitHub. On the left side of these tables, can be seen the projects that excel in security, each registering zero security smells, indicating adherence to robust security practices.

These projects include notable examples such as *janusgraph* and *superbrothers*, showcasing exemplary standards in their Kubernetes configurations, mainly as there were no found security smells.

Conversely, the right side of the tables highlights projects with a substantial number of detected security smells, suggesting areas where security enhancements are necessary. Notably, the project *oes* leads the Artifact Hub dataset with 180 security smells, followed by *sciencebox* and *ddosify*, emphasizing potential risks that could affect their reliability and safety. Similarly, the GitHub dataset features *kubernetes* with 11,243 security smells, marking it as the project with the highest number of security issues, trailed by *kyverno* and *kubernetes-sigs*, which also exhibit significant findings. These insights provide a clear indication of where further scrutiny and security improvements should be directed to mitigate risks and enhance overall security.

Table 6.13 – GitHub - Top 10 lowest and highest projects.

Top 10 lowest		**Top 10 highest**	
Project	**Security smells**	**Project**	**Security smells**
superbrothers	0	kubernetes	11243
flux-iac	0	kyverno	9199
hyperledger-labs	0	kubernetes-sigs	7284
helm-unittest	0	percona	5587
bb-Ricardo	0	kubeflow	5540
dignajar	0	solo-io	3766
ballerina-platform	0	replicatedhq	3496
rudderlabs	0	kubescape	3031
ietf-tools	0	grafana	2891
Zenika	1	luksa	2865

Source: made by author (2024).

6.5.1 Results Discussion

In analyzing the security profiles of Kubernetes projects from both Artifact Hub and GitHub, it becomes evident that while certain projects demonstrate zero identified security smells—exemplified by `janusgraph`, `superbrothers`, and other projects—the landscape also highlights areas of concern where security practices can be improved. The highest number

of security smells was observed in `oes` (Artifact Hub) with 180 instances and in `kubernetes` (GitHub) with 11,243, indicating significant security issues that may impact their stability and safety.

Despite the promising results from the presented experiments, Smelly Kube does not cover some misconfiguration as Rahman et al. (2023) tool does, which could be added in the future. By the definition of this tool's limitations, it answers the $\mathbf{RQ}_6$: *what are the limitations of this research?*

The analysis conducted in this chapter has revealed important insights into the prevalence and types of security smells present in Kubernetes manifests. With these findings established, the next chapter will synthesize the overarching conclusions of this work and propose avenues for future research and development, aimed at fortifying Kubernetes security practices and addressing the gaps identified through this study.

7 CONCLUSION

The security of Kubernetes manifests is paramount in maintaining the integrity, confidentiality, and availability of containerized applications. As Kubernetes continues to attain traction in the industry, organizations should adopt a proactive approach to security, ensuring that their manifests are not only functional but also robust against potential threats. The industry surveys and studies referenced in Chapter 1 provide the importance of securing Kubernetes applications. These sources highlight the increasing reliance on Kubernetes for container orchestration and emphasize the potential security risks if best practices are not followed.

This work develops and introduces Smelly Kube in Chapter 4, an automation for identifying security smells in Kubernetes manifests. This chapter defines the scope of this work, detailing the Kubernetes workloads and security smells addressed within the study. In addition, a Visual Studio Code extension was created to integrate with Smelly Kube, providing a way to use this solution at the early stages of the SDLC.

When comparing Smelly Kube with related works, as presented in Chapter 3, it presents notable key features. Unlike Kube-score, which detects security smells such as resource requests and limits, Smelly Kube extends its scope to critical areas like capabilities and privilege escalation settings. Compared to previous research by Rahman et al. (2023), Smelly Kube identifies additional smells, including enforcing a read-only root filesystem and running containers as non-root users, while also offering more nuanced categorizations, such as distinguishing between absent resource requests and absent resource limits. Moreover, Smelly Kube broadens the coverage of the dataset by analyzing not only the GitHub repositories but also the Artifact Hub Helm projects, filling a gap left by the studies cited in the same chapter.

The tests were run on real-world Kubernetes manifests provided by Artifact Hub and GitHub datasets, as explained in Chapter 5. These datasets can be downloaded from the dissertation repository described in the Appendix A. The results demonstrate that Smelly Kube can effectively detect a wide range of smells, offering valuable insights for security and infrastructure professionals, and contributing to the continuous improvement of Kubernetes security.

The execution and evaluation of the experiments occur in Chapter 6. These experiments revealed a high prevalence of security smells in Kubernetes manifests across Artifact Hub and GitHub, with 64.65% and 79.24% affected workloads, respectively. Pods exhibited the highest percentage of smells in Artifact Hub (88.37%), while DaemonSets led in GitHub (92.71%).

Deployments were the most commonly decoded workloads in both datasets, with Artifact Hub showing more maturity (59.24%) compared to GitHub (76.74%) in terms of workloads affected by at least one smell. These findings highlight Deployments and Pods as primary contributors to security issues, with a need to improve security configuration practices.

The most prevalent security smells in both sources were SCROFS_UNSET, SCC_UNSET, and SCAPE_UNSET, reflecting common lapses in essential security settings. The density analysis further reinforces these findings, showing that unset security configurations appear at higher rates, with SCROFS_UNSET reaching 2.91 and 9.76 per 1,000 lines of code in Artifact Hub and GitHub, respectively, SCC_UNSET appearing at 2.83 and 9.42, and SCAPE_UNSET at 2.82 and 9.29 per 1,000 lines of code in each dataset.

The most prevalent security smells in Kubernetes workloads across both datasets reveal significant security gaps. SCROFS_UNSET and SCC_UNSET were the most common smells, with GitHub showing a higher concentration of these smells between their workloads when compared to Artifact Hub. Although Artifact Hub presents a more diverse range of security smells, GitHub has a more consistent pattern of common lapses in security settings.

7.1 Future Work

For future work, it is suggested that new features be implemented. More security rules could be added for Kubernetes manifests, such as detecting hardcoded secrets exposed in the files, improper RBAC configurations, and default namespace settings. Other kinds of rules that could be implemented include other infrastructure manifests, such as the Istio service mesh, enabling the adoption of more infrastructure components. Furthermore, a more proactive approach could be taken in evaluating security configurations for each Pod running in a cluster, rather than the preventive approach described in this work.

REFERENCES

BALLA, D.; SIMON, C.; MALIOSZ, M. Adaptive scaling of kubernetes pods. In: IEEE. **NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2020. p. 1–5.

BARHAM, P. et al. Xen and the art of virtualization. **ACM SIGOPS operating systems review**, ACM New York, NY, USA, v. 37, n. 5, p. 164–177, 2003.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE cloud computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BOETTIGER, C. An introduction to docker for reproducible research. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 49, n. 1, p. 71–79, 2015.

BOSE, D. B.; RAHMAN, A.; SHAMIM, S. I. 'under-reported' security defects in kubernetes manifests. In: **2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)**. [S.l.: s.n.], 2021. p. 9–12.

CHANDRAMOULI, R. Microservices-based application systems. **NIST Special Publication**, v. 800, n. 204, p. 800–204, 2019.

CHEN, Y. et al. Why not mitigate vulnerabilities in helm charts? **arXiv preprint arXiv:2312.15350**, 2023.

CHONDAMRONGKUL, N.; SUN, J.; WARREN, I. Automated security analysis for microservice architecture. In: **2020 IEEE International Conference on Software Architecture Companion (ICSA-C)**. [S.l.: s.n.], 2020. p. 79–82.

CIS. **CIS Kubernetes Benchmarks**. 2024. Available at: <https://www.cisecurity.org/benchmark/kubernetes>. Accessed: Jun, 2024.

COSENTINO, V.; IZQUIERDO, J. L. C.; CABOT, J. A systematic mapping study of software development with github. **Ieee access**, IEEE, v. 5, p. 7173–7192, 2017.

DEMIR, M. et al. Security smells in smart contracts. In: **2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.: s.n.], 2019. p. 442–449.

DOCKER. **What is a Container?** 2022. Available at: <https://www.docker.com/resources/what-container/>.

DOCUMENTATION, D. **Docker overview**. 2022. Available at: <https://docs.docker.com/get-started/overview/>.

DOCUMENTATION, D. **Dockerfile reference**. 2022. Available at: <https://docs.docker.com/engine/reference/builder/>.

EBERT, C. et al. Devops. **IEEE software**, IEEE, v. 33, n. 3, p. 94–100, 2016.

ETCD. **etcd**. 2013. Available at: <https://etcd.io/>.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. **2015 IEEE international symposium on performance analysis of systems and software (ISPASS)**. [S.l.], 2015. p. 171–172.

FENRICH, K. Securing your control system: the" cia triad" is a widely used benchmark for evaluating information system security effectiveness. **Power Engineering**, PennWell Publishing Corp., v. 112, n. 2, p. 44–49, 2008.

FORCE, J. T. **Security and privacy controls for information systems and organizations**. [S.l.], 2017.

FOUNDATION, C. N. C. **Cloud Native Computing Foundation (CNCF) 2020**. 2020. Available at: <https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf>.

FOUNDATION, C. N. C. **With Kubernetes, the U.S. Department of Defense Is Enabling DevSecOps on F-16s and Battleships**. 2020. Available at: <https://www.cncf.io/case-studies/dod/>.

FOUNDATION, C. N. C. **CNCF Annual Survey 2023**. [S.l.], 2023.

FOWLER, M. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 2018.

GERMAN, K.; PONOMAREVA, O. An overview of container security in a kubernetes cluster. In: **2023 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)**. [S.l.: s.n.], 2023. p. 283–285.

GHAFARI, M.; GADIENT, P.; NIERSTRASZ, O. Security smells in android. In: **2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.: s.n.], 2017. p. 121–130.

GOKHALE, S. et al. Creating helm charts to ease deployment of enterprise application and its related services in kubernetes. In: IEEE. **2021 international conference on computing, communication and green engineering (CCGE)**. [S.l.], 2021. p. 1–5.

HABBAL, N. **Enhancing availability of microservice architecture: a case study on Kubernetes security configurations**. 2020.

HALE, J. S. et al. Containers for portable, productive, and performant scientific computing. **Computing in Science & Engineering**, IEEE, v. 19, n. 6, p. 40–50, 2017.

HAT, R. **What is Docker?** 2018. Available at: <https://www.redhat.com/en/topics/containers/what-is-docker>.

HAT, R. **Understanding DevOps**. 2020. Available at: <https://www.redhat.com/en/topics/devops>.

HAT, R. **What is Kubernetes?** 2020. Available at: <https://www.redhat.com/en/topics/containers/what-is-kubernetes>.

HAT, R. **The State of Kubernetes Security Report: 2024 Edition**. [S.l.], 2024.

IBM. **Introduction: Clusters**. 2023. Available at: <https://www.ibm.com/docs/en/was-nd/9.0.5?topic=servers-introduction-clusters>.

IBM. **Cost of a Data Breach Report 2024**. [S.l.], 2024.

ISO. **ISO/IEC 27001:2022 Information technology — Security techniques — Information security management systems — Requirements**. 2022.

JUNG, C. F. **Metodologia para pesquisa e desenvolvimento: aplicada a novas tecnologias, produtos e processos**. [S.l.]: Axcel Books, 2004.

KAMBHAMPATI, V.; MOHAMMED, N. H.; FARD, A. M. Characterizing javascript security code smells. **arXiv preprint arXiv:2411.19358**, 2024.

KAMPA, S. Navigating the landscape of kubernetes security threats and challenges. **Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)**, v. 3, n. 4, p. 274–281, 2024.

KHAN, M. U. A.; ZULKERNINE, M. On selecting appropriate development processes and requirements engineering methods for secure software. In: IEEE. **2009 33rd Annual IEEE International Computer Software and Applications Conference**. [S.l.], 2009. v. 2, p. 353–358.

KLEINROCK, L. Distributed systems. **Communications of the ACM**, ACM New York, NY, USA, v. 28, n. 11, p. 1200–1213, 1985.

KUBERNETES. **Case Studies**. 2020. Available at: <https://kubernetes.io/case-studies/>.

KUBERNETES. **Architecture**. 2022. Available at: <https://kubernetes.io/docs/concepts/architecture/>.

KUBERNETES. **Cloud Controller Manager**. 2022. Available at: <https://kubernetes.io/docs/concepts/architecture/cloud-controller/>.

KUBERNETES. **Components**. 2022. Available at: <https://kubernetes.io/docs/concepts/overview/components/>.

KUBERNETES. **CronJob**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>.

KUBERNETES. **DaemonSet**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>.

KUBERNETES. **Deployment**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>.

KUBERNETES. **Job**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/job/>.

KUBERNETES. **Kube Controller Manager**. 2022. Available at: <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/>.

KUBERNETES. **Managing Resources**. 2022. Available at: <https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/>.

KUBERNETES. **Production-Grade Container Orchestration**. 2022. Available at: <https://kubernetes.io/>.

KUBERNETES. **ReplicaSet**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>.

KUBERNETES. **StatefulSet**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>.

KUBERNETES. **Workloads**. 2022. Available at: <https://kubernetes.io/docs/concepts/workloads/>.

KUTE, S. S.; THORAT, S. D. A review on various software development life cycle (sdlc) models. **International Journal of Research in Computer and Communication Technology**, v. 3, n. 7, p. 778–779, 2014.

LI, W.; KANSO, A.; GHERBI, A. Leveraging linux containers to achieve high availability for cloud services. In: IEEE. **2015 IEEE International Conference on Cloud Engineering**. [S.l.], 2015. p. 76–83.

LI, W. et al. Service mesh: Challenges, state of the art, and future research opportunities. In: IEEE. **2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. [S.l.], 2019. p. 122–1225.

LIN, X. et al. A measurement study on linux container security: Attacks and countermeasures. In: **Proceedings of the 34th annual computer security applications conference**. [S.l.: s.n.], 2018. p. 418–429.

MERKEL, D. et al. Docker: lightweight linux containers for consistent development and deployment. **Linux j**, v. 239, n. 2, p. 2, 2014.

MICROSOFT. **Privileged Container**. 2023. Accessed: 2024-11-09. Available at: <https://microsoft.github.io/Threat-Matrix-for-Kubernetes/techniques/Privileged%20container/>.

MOHINO, J. de V. et al. The application of a new secure software development life cycle (s-sdlc) with agile methodologies. **Electronics**, MDPI, v. 8, n. 11, p. 1218, 2019.

OWASP. **OWASP Docker Top 10**. 2021. Available at: <https://owasp.org/www-project-docker-top-10/>.

OWASP. **OWASP Top 10:2021 - A05:2021 – Security Misconfiguration**. 2021. Available at: <https://owasp.org/Top10/A05_2021-Security_Misconfiguration/>.

OWASP. **OWASP Kubernetes Top 10**. 2022. Available at: <https://owasp.org/www-project-kubernetes-top-ten/>.

OWASP. **OWASP Kubernetes Top 10 — K01: Insecure Workload Configurations**. 2022. Available at: <https://owasp.org/www-project-kubernetes-top-ten/2022/en/src/K01-insecure-workload-configurations>.

OZTOPRAK, K.; TUNCEL, Y. K.; BUTUN, I. Technological transformation of telco operators towards seamless iot edge-cloud continuum. **Sensors**, MDPI, v. 23, n. 2, p. 1004, 2023.

PONCE, F. et al. Smells and refactorings for microservices security: A multivocal literature review. **Journal of Systems and Software**, Elsevier, v. 192, p. 111393, 2022.

RAHMAN, A.; PARNIN, C.; WILLIAMS, L. The seven sins: Security smells in infrastructure as code scripts. In: **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2019. p. 164–175.

RAHMAN, A. et al. Security misconfigurations in open source kubernetes manifests: An empirical study. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, USA, v. 32, n. 4, p. 1–36, 2023.

RAHMAN, A.; WILLIAMS, L. Different kind of smells: Security smells in infrastructure as code scripts. **IEEE Security & Privacy**, IEEE, v. 19, n. 3, p. 33–41, 2021.

SCHNEIDER, S.; SCANDARIATO, R. Automatic extraction of security-rich dataflow diagrams for microservice applications written in java. **Journal of Systems and Software**, v. 202, p. 111722, 2023. ISSN 0164-1212. Available at: <https://www.sciencedirect.com/science/article/pii/S0164121223001176>.

SERIES, O. C. S. **Docker Security Cheat Sheet**. 2019. Available at: <https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html>. Accessed: Jun, 2024.

SERIES, O. C. S. **Kubernetes Security Cheat Sheet**. 2020. Available at: <https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html>. Accessed: May, 2024.

SOLDANI, J.; TAMBURRI, D. A.; Van Den Heuvel, W.-J. The pains and gains of microservices: A systematic grey literature review. **Journal of Systems and Software**, v. 146, p. 215–232, 2018. ISSN 0164-1212. Available at: <https://www.sciencedirect.com/science/article/pii/S0164121218302139>.

SQUIRE, M. Forge++: The changing landscape of floss development. In: IEEE. **2014 47th Hawaii International Conference on System Sciences**. [S.l.], 2014. p. 3266–3275.

STACKROX, R. H. **Stackrox Red Hat Advanced Cluster Security for Kubernetes**. 2020. Available at: <https://www.redhat.com/en/technologies/cloud-computing/openshift/advanced-cluster-security-kubernetes>.

STEEN, M. V.; TANENBAUM, A. S. A brief introduction to distributed systems. **Computing**, Springer, v. 98, p. 967–1009, 2016.

ZDUN, U. et al. Microservice security metrics for secure communication, identity management, and observability. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 32, n. 1, feb 2023. ISSN 1049-331X. Available at: <https://doi.org/10.1145/3532183>.

ZEROUALI, A.; OPDEBEECK, R.; ROOVER, C. D. Helm charts for kubernetes applications: Evolution, outdatedness and security risks. In: IEEE. **2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2023. p. 523–533.

ZHANG, Y.; MURPHY, J.; RAHMAN, A. Come for syntax, stay for speed, write secure code: an empirical study of security weaknesses in julia programs. **Empirical Software Engineering**, Springer, v. 30, n. 2, p. 58, 2025.

A APPENDIX – REQUIREMENTS AND RESOURCES

The source code for Smelly Kube API and Visual Studio Code plugin can be found at <https://github.com/VitorOriel/ppgcc-ufla> in the README.md content. This repository also includes a sample Kubernetes manifest, a curl HTTP request for this manifest to the server, and the expected JSON response. Both client[1] and server[2] have their repositories, with a brief overview about each software, technologies used, and instructions to run them.

A.1 Dependencies and Installation

This section describes the dependencies for installing and running the server and the client applications. As they were developed on a Linux environment, it is recommended that the following instructions run on Linux also. For instance, the O.S. used was Ubuntu 22.04.4 LTS.

A.1.1 Server

The server dependencies may vary in the way that it will be executed. If the server would be run using Docker *recommended*, the user needs to install it[3] and install the Docker Compose plugin[4]; and the Docker should be at least version 24.0.5 or higher. To run outside Docker, it is needed to install Golang[5], at least version 1.22 or higher. The Smelly Kube API download can be done by either cloning or downloading the following repository: <https://github.com/VitorOriel/security-smells-api>.

A.1.2 Client

Using the extension as a client, the Visual Studio Code[6] needs to be installed, at least version 1.92.0 or higher. To build and run the extension, it needs to be installed the NodeJS[7],

[1] <https://github.com/VitorOriel/smelly-kube-vscode-plugin>
[2] <https://github.com/VitorOriel/security-smells-api>
[3] <https://docs.docker.com/engine/install/ubuntu/>
[4] <https://docs.docker.com/compose/install/linux/>
[5] <https://go.dev/doc/install>
[6] <https://code.visualstudio.com/download>
[7] <https://nodejs.org/en/download/package-manager>

with at least version 20.13.1 or higher, which already includes the npm dependency. To download the plugin, it can be done by either cloning or downloading the following repository: <https://github.com/VitorOriel/smelly-kube-vscode-plugin>.

After downloading it, the dependencies from *package.json* must be installed. In the Linux terminal, navigate to the repository directory and run the command `npm install .` to install the dependencies.

A.2 Minimal Test

A Kubernetes manifest example can be found at PPGCC-UFLA[8] repository, in the samples directory, and it may be used to execute minimal tests. The repository also contains an expected JSON response when sending this Kubernetes manifest to the Smelly Kube API. For a quick test, if the user does not want to deploy the Visual Studio Code extension, a curl request was provided in an executable bash script in the same directory.

First, the server must be started. Running the Smelly Kube API through Docker can be done by navigating to the repository directory, and executing the command `docker compose up server`. It executes on localhost via port 3000. To execute the minimal tests with Visual Studio Code as a client, follow these instructions:

a) Open the Visual Studio Code at the client repository. It can be done by navigating to the repository directory in the Linux terminal, and executing `code .`;
b) In the same directory, create a *.env* file with the environment variable described in the client's repository;
   e.g. <API_URL=http://localhost:3000/api/v1/smelly>
c) On Visual Studio Code window, press `Ctrl+Shift+D` to open the *Run and Debug* bar, and click at the Play icon;
d) A new Visual Studio Code window will be opened, with the extension running into it. At this window, create a file *manifest.yaml* and past the Kubernetes manifest provided in the sample directory, and save the file;
e) With this file opened, press F1 and write *Analyze file for vulnerabilities* in the command palette and executes it.

[8] <https://github.com/VitorOriel/ppgcc-ufla>

A.3 Dataset Extraction for Artifact Hub

The bash script in Listing A.1 automates the process of extracting a dataset of Kubernetes manifests from charts hosted on `artifacthub.io`. Initially, it sends a request to the helm-exporter API to retrieve metadata on publicly available Helm projects, utilizing the user-specific `API_KEY_ID` and `API_KEY_SECRET` as outlined in the dissertation repository. For each project entry returned by the API, the script captures essential information—such as the project's name, version, and repository URL—then registers each Helm chart repository with the local Helm client.

After adding the repository, the script simulates a Helm installation in the local environment by using the `-dry-run` flag, which processes the Helm templates without deploying them. This produces a consolidated Kubernetes manifest file that merges the Helm templates with their values, enabling inspection and analysis of the resulting resource definitions without making any modifications to the cluster.

Listing A.1 – Bash script to extract the manifests.

```
#!/bin/bash

# Command to retrieve the JSON
json=$(curl -X 'GET' \
    'https://artifacthub.io/api/v1/helm-exporter' \
    -H 'Accept: application/json' \
    -H 'X-API-KEY-ID: ${API_KEY_ID}' \
    -H 'X-API-KEY-SECRET: ${API_KEY_SECRET}')

# Loop through each entry in the JSON
for entry in $(echo "$json" | jq -c '.[]'); do
    # Extract the name, version, and repository URL
    # from the resource
    name=$(echo "$entry" | jq -r '.name')
    version=$(echo "$entry" | jq -r '.version')
    repoUrl=$(echo "$entry" | jq -r '.repository.url')
    # Add the repository
    helm repo add "$name" "$repoUrl"
```

```
    # Install the Helm chart with --debug and --dry-run,
    # outputting to a manifest file
    helm install "${name}-release" "$name/$name" \
        --version "$version" \
        --debug --dry-run > "${name}-manifest.yaml"
done
```

Source: made by author (2024).

A.4 Treat Result

This section contains the detailed algorithms for the `treat_result` function. The algorithm processes the result dictionary to update the global metadata related to security smells in Kubernetes manifests, as seen in Algorithm 8. The algorithm begins by updating the total number of smells in the ‘smells_meta‘ dictionary (line 2), using the value provided in the ‘meta‘ section of the ‘result‘. Then, the algorithm loops through the ‘decodedWorkloads‘ section of the ‘result[’meta’]‘ dictionary (line 3), where it updates the ‘workload_meta[’decoded’]‘ dictionary by adding the decoded workload count for each key. If a key is not present in the dictionary, it is initialized to zero (lines 5-7). The algorithm then proceeds to process the ‘data‘ section of the ‘result‘ (line 10), which contains the actual security smells. For each key in the ‘data‘, it updates the total number of smells recorded in ‘workload_meta[’smells’]‘. If a key is not already present in this dictionary, it is initialized (lines 11-13). The algorithm then checks if there are any smells associated with the key. If there are smells, the `treat_smells` function is called to process the smells further (lines 16-18).

The `treat_smells` algorithm processes the list of smells associated with a specific Kubernetes workload, see Algorithm 9. It begins by updating the total number of affected workloads in the ‘workload_meta[’affected’]‘ dictionary (line 2), incrementing it by one. If the ‘key‘ is not already present in the ‘affected‘ section of ‘workload_meta‘, it initializes it to zero (lines 3-5). The algorithm then checks if the ‘key‘ is present in the ‘workload_with_rules‘ dictionary and initializes it as an empty dictionary if necessary (lines 7-9). Next, it loops through each smell in the ‘kube_smells‘ list (line 10). For each smell, it retrieves the ‘rule‘ associated with it and updates the ‘smells_meta‘ dictionary by incrementing the count of occurrences for that rule (lines 12-14). If the rule is not already present in the ‘workload_with_rules‘ dictionary for the given ‘key‘, it initializes it to zero (lines 16-18). Finally, the algorithm increments the

count of occurrences of the rule in the ‘workload_with_rules‘ dictionary for the corresponding workload (lines 19). This ensures that both the global smell statistics and the workload-specific statistics are updated for each detected security smell.

Algorithm 8 - Treat Result.

Algorithm 8 Treat Result

Require: *result*, a dictionary with keys *meta* and *data*
Ensure: Updates global dictionaries: *workload_meta*, *smells_meta*, *workload_with_rules*

```
1: global workload_meta, smells_meta, workload_with_rules
2: smells_meta["total"] ← smells_meta["total"] + result["meta"]["totalOfSmells"]
3: for all (key, value) in result["meta"]["decodedWorkloads"] do
4:     workload_meta["decoded"]["total"] ← workload_meta["decoded"]["total"] + int(value)
5:     if key not in workload_meta["decoded"] then
6:         workload_meta["decoded"][key] ← 0
7:     end if
8:     workload_meta["decoded"][key] ← workload_meta["decoded"][key] + int(value)
9: end for
10: for all (key, value) in result["data"] do
11:     if key not in workload_meta["smells"] then
12:         workload_meta["smells"][key] ← 0
13:     end if
14:     kube_smells ← list(value)
15:     workload_meta["smells"][key] ← workload_meta["smells"][key] + length(kube_smells)
16:     if kube_smells is not empty then
17:         treat_smells(key, kube_smells)
18:     end if
19: end for
```

Source: made by author (2024).

Algorithm 9 - Treat Smells.

Algorithm 9 Treat Smells

Require: *key*, a string; *kube_smells*, a list of dictionaries
Ensure: Updates global dictionaries: *smells_meta*, *workload_meta*, *workload_with_rules*

```
1:  global smells_meta, workload_meta, workload_with_rules
2:  workload_meta["affected"]["total"] ← workload_meta["affected"]["total"] + 1
3:  if key not in workload_meta["affected"] then
4:      workload_meta["affected"][key] ← 0
5:  end if
6:  workload_meta["affected"][key] ← workload_meta["affected"][key] + 1
7:  if key not in workload_with_rules then
8:      workload_with_rules[key] ← {}
9:  end if
10: for all smell in kube_smells do
11:     rule ← smell["rule"]
12:     if rule not in smells_meta then
13:         smells_meta[rule] ← 0
14:     end if
15:     smells_meta[rule] ← smells_meta[rule] + 1
16:     if rule not in workload_with_rules[key] then
17:         workload_with_rules[key][rule] ← 0
18:     end if
19:     workload_with_rules[key][rule] ← workload_with_rules[key][rule] + 1
20: end for
```

Source: made by author (2024).