HIGH PERFORMANCE INTRUSION PREVENTION FOR SOFTWARE-DEFINED CONTAINER NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YALGINAY YALTIRIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

OCTOBER 2024

Approval of the thesis:

HIGH PERFORMANCE INTRUSION PREVENTION FOR SOFTWARE-DEFINED CONTAINER NETWORKS

submitted by **YALGINAY YALTIRIK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Naci Emre ALTUN
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Mehmet Halit S. Oğuztüzün
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Pelin Angın Ülküer
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. İbrahim Körpeoğlu
Computer Engineering, Bilkent University _____

Assoc. Prof. Dr. Pelin Angın Ülküer
Computer Engineering, METU _____

Prof. Dr. Ertan Onur
Computer Engineering, METU _____

_____

_____

Date:25.10.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Yalgınay Yaltırık

Signature :

ABSTRACT

HIGH PERFORMANCE INTRUSION PREVENTION FOR SOFTWARE-DEFINED CONTAINER NETWORKS

Yaltırık, Yalgınay
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Pelin Angın Ülküer

October 2024, 146 pages

The rapid growth of Internet of Things, along with advancements in mobile networks, big data, and cloud computing has increased the need for faster and more reliable networks. Software-Defined Networks have become crucial to meeting the performance needs of applications such as smart homes, autonomous vehicles and remote healthcare services. Simultaneously, microservice architectures in cloud computing have become more valuable due to the flexibility and efficiency of containers. As more container-based applications are used, the risk of cyberattacks targeting these networks has also risen. This thesis presents an intrusion prevention model designed for software-defined container networks. The model includes a system for collecting traffic data from containers and a health monitoring module aimed at stopping detected security threats. By continuously monitoring the network and responding to potential attacks, this approach offers an efficient and scalable solution to protect container-based cloud environments from increasing cybersecurity risks.

Keywords: Container Security, Software Defined Network, Cyberattacks on contain-

ers, Google Kubernetes Engine, Intrusion Prevention

ÖZ

YAZILIM TANIMLI KAPSAYICI AĞLARI İÇİN YÜKSEK BAŞARIMLI SALDIRI ENGELLEME

Yaltırık, Yalgınay
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Pelin Angın Ülküer

Ekim 2024 , 146 sayfa

Nesnelerin İnternetindeki hızlı büyüme, mobil ağlar, büyük veri ve bulut bilişimdeki gelişmelerle birlikte daha hızlı ve daha güvenilir ağlara olan ihtiyacı artırmıştır. Yazılım Tanımlı Ağlar, akıllı evler, otonom araçlar ve uzaktan sağlık hizmetleri gibi uygulamaların performans ihtiyaçlarını karşılamak için önemli hale gelmiştir. Aynı zamanda, bulut bilişimdeki mikroservis mimarileri, konteynerlerin esneklikleri ve verimlilikleri sayesinde daha değerli hale gelmiştir. Daha fazla konteyner tabanlı uygulama kullanıldıkça, bu ağlara yönelik siber saldırı riski de artmıştır. Bu tez, yazılım tanımlı konteyner ağları için tasarlanmış bir sızma önleme modeli sunmaktadır. Model, konteynerlerin trafik verilerini toplamak için bir sistem ve tespit edilen güvenlik tehditlerini durdurmaya yönelik bir sağlık izleme modülü içermektedir. Ağın sürekli izlenmesi ve olası saldırılara yanıt verilmesiyle, bu yaklaşım, konteyner tabanlı bulut ortamlarını artan siber güvenlik risklerine karşı korumak için verimli ve ölçeklenebilir bir çözüm sunmaktadır.

Anahtar Kelimeler: Konteyner Güvenliği, Yazılım Tanımlı Ağ, Konteynerlere Yönelik Siber Saldırılar, Google Kubernetes Motoru, Saldırı Önleme

To my parents, whose steadfast love and endless support have been the cornerstone of all my accomplishments, providing me guidance through their strength.

ACKNOWLEDGMENTS

I would like to extend my heartfelt appreciation to my supervisor, Assoc. Prof. Dr. Pelin Angın Ülküer, for her invaluable guidance and support throughout the course of this research. Her expertise and insightful feedback were instrumental in the development of this thesis.

I am also profoundly grateful to my family for their unwavering support and encouragement, whose love and patience have been a continuous source of strength and motivation.

Finally, I wish to express my sincere thanks to Middle East Technical University for providing the essential resources for this study and my thesis committee members Prof. Dr. Ertan Onur and Prof. Dr. İbrahim Körpeoğlu for their valuable feedback, allowing me to improve the quality of this thesis.

This research has been supported by TÜBİTAK under grant number 120E537. The entire responsibility of the thesis belongs to the author of the thesis. The support received from TÜBİTAK does not mean that the content of the publication is approved in a scientific sense by TÜBİTAK.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xvi

LIST OF FIGURES xx

LIST OF ABBREVIATIONS xxiii

CHAPTERS

1 INTRODUCTION 1

2 PRELIMINARIES 5

2.1 Container 5

2.2 Containers vs Virtual Machines 6

2.3 Software Defined Network (SDN) 6

2.4 Kubernetes 7

2.5 Tcpdump 8

2.6 Container Network Interface (CNI) 8

2.7 Sidecar Containers 8

2.8 Network Address Translation (NAT) 8

3 LITERATURE REVIEW 11

3.1 AI-Driven Container Security Approaches for 5G and Beyond [1] . . 11

3.2 HoneyKube: Designing a Honeypot Using Microservices-Based Architecture [2] 11

3.3 Zero Trust Network Security Model in Containerized Environments [3] 12

3.4 A Study on Analysis and Detection of Container Escape Vulnerabilities in Docker [4] 12

3.5 Misuse Detection and Response for Orchestrated Microservices Based Software [5] 13

3.6 An Empirical Analysis of IDS Approaches in Container Security [6] . 13

3.7 Kubernetes Dataset for Misuse Detection [7] 14

3.8 ProtÉdge: A few-shot ensemble learning approach to software-defined networking assisted edge security [8] 14

3.9 Unsupervised Anomaly Detection for Container Cloud Via BILSTM-Based Variational Auto-Encoder [9] 15

3.10 Securing Communication and Identifying Threats in Cloud Microservices [10] 15

3.11 Automated Traces-based Anomaly Detection and Root Cause Analysis in Cloud Platforms [11] 16

3.12 ContainerGuard: A Real-Time Attack Detection System in Container-Based Big Data Platform [12] 16

3.13 Container Anomaly Detection Using Neural Networks Analyzing System Calls [13] 17

3.14 A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest [14] 17

3.15 Leveraging Computational Intelligence Techniques for Defensive Deception: A Review, Recent Advances, Open Problems and Future Directions [15] 18

3.16 A Comprehensive Analysis of Machine Learning and Deep Learning-Based Solutions for DDoS Attack Detection in SDN [16] 18

3.17 An integrated SDN Framework for Early Detection of DDoS Attacks in Cloud Computing [17] 19

3.18 Detection and Prevention of DoS attacks in Software-Defined Cloud Networks [18] 19

3.19 DDoS attack Mitigation in SDN-based Cloud Computing Environment [19] 19

3.20 Efficient Distributed DoS Attack Defense in SDN-based Cloud [20] . 20

3.21 Mitigating DDoS Attacks using SDN-based Network Security Measures [21] 20

3.22 A DDoS Attack Detection and Mitigation With Software-Defined Internet of Things Framework [22] 20

3.23 Blockchain as a Service for Software Defined Networks: A DoS Attack Perspective [23] 21

3.24 Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments [24] 21

3.25 DDoS Attack Mitigation in IoT Using SDN [25] 22

3.26 Flow Based Security for IoT Devices Using an SDN Gateway [26] . . 22

3.27 Information Classification Strategy for Blockchain-based Secure SDN in IoT Scenario [27] 23

3.28 Learning-Driven Detection and Mitigation (LEDEM) of DDoS Attack in IoT via SDN-cloud Architecture [28] 23

3.29 Defending Against Packet-In Messages Flooding Attack under SDN Context [29] 23

3.30 Security Analysis of SDN Cloud Applications [30] 24

3.31 Blockchain and Random Subspace Learning-Based IDS for SDN-Enabled Industrial IoT Security [31] 24

3.32 Detection and Mitigation of DoS and DDoS Attacks in IoT-Based Stateful SDN: An Experimental Approach [32] 25

3.33 Securing Distributed Systems: A survey on Access Control Techniques for Cloud, Blockchain, IoT and SDN [33] 25

3.34 A Review on Cyber security of Cloud Computing for Supporting Connected Vehicle Applications [34] 26

3.35 Powering Hands-on Cybersecurity Practices with Cloud Computing [35] 26

3.36 Securing Cloud, SDN and Large Data Network Environments from Emerging DDoS Attacks [36] 27

3.37 SDN-Enabled Hybrid DL-Driven Framework for the Detection of Emerging Cyber Threats in IoT [37] 27

4 METHODOLOGY 29

4.1 Pod Labeling 32

4.2 Tcpdumper (Sidecar Container) 33

4.3 Fetch Log Module (Log Collector) 34

4.4 Health Monitor Module (IPS) 34

5 EXPERIMENTS 37

5.1 Fetch Log Module - Initial Version Analysis 37

5.1.1 Varying Test Duration 37

5.1.2 Varying Pod Count 39

5.2 Fetch Log Module - Updated Version Analysis 40

5.2.1 Varying Test Duration 40

5.2.2 Varying Pod Count 41

5.2.3 CPU and Memory Usage 42

5.3 Node Performance Analysis Based On Pod Type 45

5.3.1 All Pods in The Same Node with Tcpdumper 45

5.3.2 Node Performance Analysis Without Tcpdumper 47

5.3.3 Single Server with Clients in Separate Nodes 49

5.3.4 2-Server with Clients in Separate Nodes 50
5.4 Impact of Communication Period on Response Delay 53
5.4.1 With Tcpdumper 53
5.4.2 Without Tcpdumper 54
5.5 Analysis of Health Monitor Module 56
5.5.1 HM - Initial Version with Tcpdumper 56
5.5.2 HM - Initial Version without Tcpdumper 66
5.5.3 HM Initial Version with Nginx and Tcpdumper 66
5.5.4 Comprehensive Analysis of HM's Initial Version 67
5.5.5 HM - Updated Version with Tcpdumper 69
5.5.6 HM - Updated Version with Nginx and Tcpdumper 71
5.5.7 Comprehensive Analysis of Initial and Updated Version of HM 73
5.6 Serial 3-Pod Topology 79
5.7 Serial 10-Pod Topology 91
5.8 7-Pod Topology 99
5.9 Multi-Client Analysis on 7-Pods Topology 112
5.10 5-Endpoints Topology 130
6 CONCLUSION 139
REFERENCES 141

LIST OF TABLES

TABLES

Table 5.1 Fetch Log Module Performance with Initial Design 38

Table 5.2 Fetch Log Module - Initial Version Performance Analysis Based on Pod Count 39

Table 5.3 Fetch Log Module Performance with Updated Design 41

Table 5.4 Fetch Log Module - Updated Version Performance Analysis Based on Pod Count 42

Table 5.5 Fetched Log Sizes Based on The Number of Pods 42

Table 5.6 Fetch Log Module - Updated Version Analysis Based on Different Process Numbers 43

Table 5.7 Node Performance Analysis Based on Pod Counts (with tcpdumper) 46

Table 5.8 Node Performance Analysis Based on Pod Counts (without tcpdumper) 49

Table 5.9 Node Performance vs Pod Count (with tcpdumper, single server) . . 51

Table 5.10 Node Performance vs Pod Count (with tcpdumper, dual server) . . . 52

Table 5.11 Client Message Transmit Time (with tcpdumper) 53

Table 5.12 Client Message Transmit Time (without tcpdumper) 55

Table 5.13 Pod Labels Before Network Update 58

Table 5.14 Pod Labels After Network Update 61

Table 5.15 Network Update Analysis Based on Communication Period (with tcpdumper) 62

Table 5.16 Network Update Analysis Based on Communication Period (without tcpdumper) 66

Table 5.17 Network Update Analysis Based on Com. Period (with nginx, tcpdumper) 67

Table 5.18 Network Update Analysis Based on Communication Period and Container Type 68

Table 5.19 Network Update Analysis on Pod Communication (with tcpdumper) 71

Table 5.20 Network Update Analysis Based on Com. Period (with nginx) . . . 72

Table 5.21 Comprehensive Network Update Analysis 78

Table 5.22 Average Response Times Before Network Update 83

Table 5.23 Client Pod Timestamps During Network Update - Pod3 Fail Case . . 84

Table 5.24 Pod Response Times During Network Update - Pod3 Fail Case . . . 84

Table 5.25 Client Response Time Analysis (without tcpdumper, Pod3 Failed) . 85

Table 5.26 Client Response Time Analysis (with tcpdumper, Pod3 Failed) . . . 85

Table 5.27 Average Client Response Time Analysis - Pod3 Fail Case 86

Table 5.28 Client Pod Timestamps During Network Update - Pod2 Fail Case . . 87

Table 5.29 Average Client Response Time Analysis - Pod2 Fail Case 87

Table 5.30 Client Pod Timestamps During Network Update - Pod1 Fail Case . . 88

Table 5.31 Average Client Response Time Analysis - Pod1 Fail Case 88

Table 5.32 Average Client Response Time Analysis 89

Table 5.33 Client and Pods Response Duration 93

Table 5.34 Client Pod Timestamps During Network Update (Pod1 failed case) . 97

Table 5.35 Average Client Response Time Analysis - Pod1 fail case) 97

Table 5.36 Client Pod Timestamps During Network Update - Pod5 fail case . . 97

Table 5.37 Average Client Response Time Analysis- Pod5 fail case 98

Table 5.38 Client Pod Timestamps During Network Update - Pod10 fail case . . 98

Table 5.39 Average Client Response Time Analysis - Pod10 fail case 98

Table 5.40 Average Client Response Time Analysis Based On Failed Pod . . . 98

Table 5.41 Pods Average Response Duration (50 Communication Sample) . . . 102

Table 5.42 Client Pod Timestamps During Network Update - Pod1 fail case) . . 104

Table 5.43 Average Client Response Time Analysis (Pod1 fail case) 104

Table 5.44 Client Pod Timestamps During Network Update(Pod5 fail case) . . 106

Table 5.45 Average Client Response Time Analysis (Pod5 fail case) 106

Table 5.46 Client Pod Timestamps During Network Update (Pod3 fail case) . . 107

Table 5.47 Average Client Response Time Analysis (Pod3 fail case) 107

Table 5.48 Client Pod Timestamps During Network Update(Pod7 fail case) . . 107

Table 5.49 Average Client Response Time Analysis (Pod7 fail case) 108

Table 5.50 Average Client Response Time Analysis 109

Table 5.51 1 Client Average Response Time 116

Table 5.52 2 Clients Average Response Times 117

Table 5.53 5 Clients Average Response Times 118

Table 5.54 10 Clients Average Response Times 120

Table 5.55 25 Clients Average Response Times 123

Table 5.56 Part 1 - 50 Clients Average Response Times (1-25-clients) 124

Table 5.57 Part 2 - 50 Clients Average Response Times (26-50-clients) 125

Table 5.58 Part 1 - 100 Clients Average Response Times (1-25-clients) 126

Table 5.59 Part 2 - 100 Clients Average Response Times (26-50-clients) 127

Table 5.60 Part 3 - 100 Clients Average Response Times (51-75 clients) 128

Table 5.61 Part 4 - 100 Clients Average Response Times (76-100 clients) 129

Table 5.62 Client and Pods Response Duration 132

Table 5.63 Client Pod Timestamps During Network Update(Pod1 fail case) . . 135

Table 5.64 Average Client Response Time Analysis (Pod1 fail case) 135

Table 5.65 Client Pod Timestamps During Network Update (Pod6 fail case) . . 136

Table 5.66 Average Client Response Time Analysis (Pod6 fail case) 136

Table 5.67 Client Pod Timestamps During Network Update(Pod7 fail case) . . 137

Table 5.68 Average Client Response Time Analysis (Pod7 fail case) 138

LIST OF FIGURES

FIGURES

Figure 4.1 Containers with no attacker in the network 29

Figure 4.2 Attacker targeting pod1 30

Figure 4.3 Pod1 under attack 30

Figure 4.4 Blocking the attacker 31

Figure 4.5 System model 32

Figure 5.1 CPU and Memory Usages of Server 47

Figure 5.2 CPU and Memory Usages of Clients 48

Figure 5.3 OutofPods fail case 50

Figure 5.4 GKE Nodes 50

Figure 5.5 Client Pod Log 54

Figure 5.6 Initial Network Topology 57

Figure 5.7 Cluster IP Addresses 57

Figure 5.8 HM Action 59

Figure 5.9 Updated Network Topology 60

Figure 5.10 Pod3 Log (1) 63

Figure 5.11 Pod3 Log (2) 64

Figure 5.12 Pod3rep Log 65
Figure 5.13 Cluster IP addresses 69
Figure 5.14 Updated HM Action 70
Figure 5.15 Pod3 Log 74
Figure 5.16 Pod3rep Log 75
Figure 5.17 Updated HM Action 76
Figure 5.18 Pod3 Log 77
Figure 5.19 Pod3rep Log 77
Figure 5.20 Serial 3-Pod Topology 79
Figure 5.21 Cluster IP Addresses 80
Figure 5.22 Pod3 Log with Attacker Pod 81
Figure 5.23 HM Action 82
Figure 5.24 Pod3rep Log 83
Figure 5.25 Client Log 89
Figure 5.26 Pod1rep Log 90
Figure 5.27 Serial 10-Pods Topology 91
Figure 5.28 Responses of the Client Requests 92
Figure 5.29 HM Action 94
Figure 5.30 Pod1 Log 95
Figure 5.31 Pod1rep log 96
Figure 5.32 7-Pod Network Topology 99
Figure 5.33 Client Responses 101

Figure 5.34 HM Action 103
Figure 5.35 Client Responses (Pod1 Fail Case) 105
Figure 5.36 Client Responses (Pod7 Fail Case) 108
Figure 5.37 Pod7 Last Log Before Network Update 109
Figure 5.38 Pod7rep First Log After Network Update 110
Figure 5.39 Client Log During Network Update 111
Figure 5.40 7-Pods Network Topology with Corresponding Nodes 113
Figure 5.41 Pod, Node and Service IP's (without Attacker Scenario) 114
Figure 5.42 Client Log without Attacker Scenario 115
Figure 5.43 Pod, Node and Service IP's (with Attacker Scenario) 116
Figure 5.44 Pod and Node IPs (with Attacker 5-Clients Scenario) 117
Figure 5.45 HM Action in 10-Clients Scenario 119
Figure 5.46 Group of Deployed Pods in The 100 Pods Cluster 121
Figure 5.47 5-Endpoints Network Topology 130
Figure 5.48 Pod IP's in Cluster 131
Figure 5.49 Client Responses 132
Figure 5.50 Pod1 Log 133
Figure 5.51 Pod1rep Log 134
Figure 5.52 Pod1 Log 137

LIST OF ABBREVIATIONS

CNI	Container Network Interface
DDoS	Distributed Denial of Service
DoS	Denial of Service
GKE	Google Kubernetes Engine
HM	Health Monitor
ICMP	Internet Control Message Protocol
IPS	Intrusion Prevention System
IDS	Intrusion Detection System
IoT	Internet of Things
QoS	Quality of Service
SDN	Software Defined Network
SDCN	Software Defined Container Networks
TCP	Transmission Control Protocol

CHAPTER 1

INTRODUCTION

The rapid advancements in mobile networks, big data, and cloud computing have led to a steady increase in the use cases for connected devices and the Internet of Things (IoT). IoT applications, which simplify everyday life, are increasingly being utilized in areas such as smart homes, autonomous vehicles, security systems, smart cities and remote healthcare services. Given the immense volume of data generated by IoT devices, there is a growing need for high-speed networks and computing systems to ensure the Quality of Service (QoS) required by these diverse applications. Software-defined networks (SDNs), with their ability to meet these demands, are expected to play a vital role in the future of IoT.

The evolution of cloud computing has also driven the development of microservices-based applications to meet the high-performance requirements of cloud systems. These applications follow a complex architecture composed of independent services interacting via specific APIs. Containers, which offer a lighter and faster alternative to traditional virtual machines, are frequently employed in microservices [1].

The rapid expansion of cloud-based systems has led to widespread deployment of software-defined container networks (SDCNs) across industries, from enterprise applications to critical infrastructure. This surge in containerization, however, has exposed SDCNs to an evolving landscape of cybersecurity threats, with attackers increasingly leveraging vulnerabilities in containerized environments. Common attack types such as denial of service (DoS), privilege escalation, image poisoning, container escape and man-in-the-middle attacks have posed substantial risks to container networks. When such attacks occur, containers can no longer provide reliable services, leading to faulty responses or response delays on the client side. In some cases,

clients may not receive responses at all. Traditional security solutions often fail to provide adequate protection due to their inability to adapt to the highly dynamic, distributed nature of containerized environments, highlighting the urgent need for a tailored, high-performance IPS capable of effectively defending against these threats.

Motivated by this pressing security challenge, this thesis introduces a novel IPS model specifically designed for SDCNs. The proposed model actively prevents attackers within the network. In the event of detected threats, the model provides the ability to dynamically update the network configuration, rerouting traffic or isolating compromised containers as necessary. This proactive approach effectively minimizes the potential impact of ongoing attacks while continuously enhancing the resilience of the network. By leveraging real-time traffic data collection and health monitoring modules, the IPS architecture enables seamless traffic analysis and rapid identification of abnormal behavior, making it highly effective for maintaining system integrity and security.

The contributions of this study are twofold: We present a scalable and robust IPS model tailored specifically for SDCNs and introduce an advanced network response mechanism that dynamically reconfigures the network to contain and mitigate threats. This dual approach not only strengthens the defensive capabilities of container networks but also maintains the high performance and scalability required for large-scale cloud deployments. By integrating these security features, the study provides a comprehensive solution to safeguard containerized environments against a broad spectrum of cybersecurity threats, ultimately enhancing the security posture, operational continuity, and resilience of modern cloud computing infrastructures.

The thesis consists of six chapters.

- Chapter 1 provides an overview of the research problem, introducing key concepts and outlining the structure of the subsequent chapters.
- Chapter 2, the "Preliminaries", provides an overview of containers, SDN and related concepts.
- In Chapter 3, the existing literature on container security is thoroughly analyzed.

- Chapter 4, the "Methodology," forms the core of the thesis. It provides an in-depth explanation of the attack prevention process within an SDCN in the cloud. The sub-modules of the proposed method and their respective functions are detailed in this chapter.
- Chapter 5 evaluates the performance of the proposed model and its components across various environments. The results from these experiments are then used to assess the model's effectiveness in different scenarios.
- Chapter 6 concludes the thesis by summarizing the key findings, discussing their significance and proposing directions for future research.

CHAPTER 2

PRELIMINARIES

2.1 Container

Containers are isolated runtime environments that share the same operating system (OS) kernel, but operate independently of each other. They have an encapsulated environment that has an allocated memory. With the help of the encapsulation, they have a consistent runtime environment across different development, testing, and production systems.

Docker [38] is a platform designed to simplify the process of developing, shipping, and running applications. It enables developers to package applications and their dependencies into a container. The Docker architecture consists of the following core components:

- **Images**: A Docker image is a file that contains everything needed to run a program, including the code and system tools. It is used to create Docker containers.
- **Dockerfile**: This is a script with instructions on how to build a Docker image. It tells Docker what to include in the image and how to set it up as a text file that contains a series of instructions on how to build a Docker image. It includes details such as the base image to use, the application code to include and the commands to run. The Dockerfile automates the process of creating Docker images.
- **Docker Hub**: This is a cloud service where we can store and share Docker images. It provides a central repository for both public and private images,

facilitating easy access and collaboration.

2.2 Containers vs Virtual Machines

Virtual machines (VMs) interact with a physical machine's hardware through a hypervisor which provides an abstraction layer for access to CPU, memory and storage. In contrast, containers encapsulate an executable along with its necessary dependencies [39].

Containers share the underlying hardware and OS kernel of the host machine with other containers, making them more resource-efficient compared to VMs. Containers, being lighter and more portable, are ideal for applications requiring rapid deployment and optimized compute resources. Their efficiency in resource use and speed of deployment offer significant advantages over VMs in many scenarios.

2.3 Software Defined Network (SDN)

SDN [40] is a network architecture approach that separates the network control plane from the data plane. By this way, it allows network management and configuration to be handled through software rather than traditional hardware. This separation enables centralized control of network traffic and resources and allows for greater flexibility and efficiency in managing network infrastructure. In an SDN environment, the network is managed through a software-based controller that communicates with network devices using open protocols.

The main advantages of using SDNs over traditional network architectures can be summarized as follows:

- **Flexibility**: SDNs enable easy changes and management of network configurations through software, allowing for quick adjustments to new needs or issues without physical hardware modifications.
- **Centralized Control**: SDNs allow to control and manage the entire network from a single location. This makes it simpler to monitor and optimize network

performance across different development, testing and production systems.

- **Improved Efficiency**: Automating network management in SDNs mitigate the need for manual tasks. This reduces errors and improves the use of network resources.

- **Scalability**: SDNs simplify the process of adjusting network resources according to demand. This means we can increase or decrease them without significant changes to the physical setup.

- **Cost Savings**: Using SDNs can lower costs by reducing the need for specialized hardware and manual network management.

2.4 Kubernetes

Kubernetes [41] is an open-source platform designed to manage containerized applications distributed across various machines. It automates essential tasks such as scaling, load balancing and deploying applications. In short, it serves as an orchestration tool. The fundamental component of Kubernetes is the **pod**, which can contain one or more containers. The individual machines utilized within a Kubernetes cluster are referred to as **node**s.

- **Google Kubernetes Engine (GKE)**: GKE is a managed service that makes it easier to run Kubernetes clusters on Google Cloud. It handles the setup and maintenance of the Kubernetes infrastructure by allowing to focus on deploying and managing applications.

- **Calico Plugin**: The plugin Calico [42] is used in Kubernetes to handle networking and network security. It provides efficient network routing and security policies for communication between containers. In Google Kubernetes Engine, Calico helps manage network traffic and enforce security rules.

2.5 Tcpdump

Tcpdump [43] is a command-line tool used to capture and analyze network packets in real-time. It works with the libpcap library to monitor network traffic and provide details about data packets such as source and destination addresses. Tcpdump is useful for diagnosing network issues and understanding network behavior. In this work, it was integrated into the **tcpdumper container** to capture the network traffic of the pods.

2.6 Container Network Interface (CNI)

A CNI is a framework for dynamic configuration of networking resources in containers. The interface allows configuring the network, provisioning IP addresses, and maintaining connectivity with multiple hosts. When using a CNI [44] or a network overlay that supports traffic mirroring, directing traffic to a specific destination is simple. However, not every CNI implementation offers traffic mirroring support. Asking users to change to a compatible CNI or network overlay is not a practical solution.

2.7 Sidecar Containers

Sidecar containers [44] are lightweight containers deployed within the same pod that share the same network namespace as the primary containers. This gives them direct access to the network traffic flowing to and from the container. This approach necessitates the development of a dedicated packet-sniffing application within the sidecar. It offers the advantage of being independent of Kubernetes-specific components, such as CNIs and network overlays. It operates at the container level.

2.8 Network Address Translation (NAT)

NAT is a technique that involves altering network address information in the IP header of packets as they pass through a traffic routing device, thereby mapping one IP ad-

dress space to another [45]. In this work, it is used with PRE routing and POST routing rules to route network packets originating from an attacker pod IP.

CHAPTER 3

LITERATURE REVIEW

This section reviews related studies in the literature with a focus on analyzing the various measures employed to address cyber attacks in SDCN.

3.1 AI-Driven Container Security Approaches for 5G and Beyond [1]

The core concept of this paper is the evaluation of AI-driven methodologies for enhancing container security, particularly within 5G and future network environments. It emphasizes the significance of effective intrusion detection, malware identification and anomaly detection to protect containerized applications and Virtual Network Functions. The study highlights various machine learning (ML) and deep learning (DL) techniques, assessing their performance based on specific use cases and algorithms. While the surveyed models demonstrate high accuracy and efficiency in identifying threats, the paper also addresses challenges such as computational costs, generalizability and the interpretability of AI-driven systems. Ultimately, it advocates for the integration of advanced AI techniques in securing containers against emerging threats in increasingly complex network architectures.

3.2 HoneyKube: Designing a Honeypot Using Microservices-Based Architecture [2]

This paper does not propose a direct intrusion prevention model but introduces HoneyKube, a honeypot designed using microservices architecture to gather data on attacks targeting such systems. While the honeypot does not prevent attacks, it collects

valuable information for developing future security measures in microservices and cloud-native environments. The paper's strengths include its novel design, comprehensive monitoring and use of realistic application code, though it is limited by short experiment duration and gaps in capturing certain attacker activities. The study gathered significant attack data offering insights into patterns specific to microservices architectures, which could enhance security solutions. While not focused on SDN, the design elements, such as Role-Based Access Control and network policies, indirectly benefit SDN security. Overall, the research enhances understanding of distributed system vulnerabilities, providing a foundation for improved security strategies.

3.3 Zero Trust Network Security Model in Containerized Environments [3]

This paper focuses on implementing a Zero Trust security model in containerized environments, particularly for securing east-west traffic between microservices, rather than introducing a dedicated intrusion prevention model. It highlights the use of key components such as micro-segmentation, mutual TLS encryption and network visibility tools like Istio, Cilium and Hubble to enhance security in cloud-native and SDN-based architectures. The advantages include robust traffic encryption, security isolation and fine-grained control, though the complexity of implementation and potential performance overheads are notable drawbacks. Although the paper does not specifically address how SDN is protected from attackers, its Zero Trust approach offers inherent safeguards through segmentation and encrypted communications. Overall, it provides valuable insights into improving security in containerized and SDN environments.

3.4 A Study on Analysis and Detection of Container Escape Vulnerabilities in Docker [4]

This paper provides a comprehensive analysis of container escape vulnerabilities in Docker environments, focusing on various Common Vulnerabilities and Exposures (CVEs) and testing them on ARM-based architectures. While the paper does not propose a specific intrusion prevention model, it demonstrates effective detection meth-

ods using system call logging and rule-based log analysis. This approach, though adaptable, is more reactive, emphasizing detection over prevention. The study's key strengths lie in its practical applicability and extensive vulnerability analysis, though its narrow focus on Docker limits its generalizability. Furthermore, the use of Auditd may introduce performance concerns in production systems. Despite these limitations, the findings can be applied to enhance security in SDN and cloud infrastructures, especially by informing better isolation and monitoring mechanisms.

3.5 Misuse Detection and Response for Orchestrated Microservices Based Software [5]

This paper introduces a security framework designed for Kubernetes-based microservices, focusing on detecting and responding to misuse rather than preventing intrusions outright. It evaluates the performance of many ML models, with OneClassSVM achieving the highest accuracy (98.11%) and precision (1.0) in anomaly detection. The framework's key strengths include its high detection accuracy, real-time dynamic responses to threats and effective integration with Kubernetes and Kube-OVN, benefiting both SDN and cloud environments. However, its complexity and potential performance overhead in high-traffic networks present challenges. While primarily applicable to Kubernetes, the model's principles such as dynamic network topology adjustments and AI-driven threat detection can be extended to secure SDN environments.

3.6 An Empirical Analysis of IDS Approaches in Container Security [6]

This paper provides a detailed comparison of host-based and network-based IDS within containerized environments, using ML techniques to evaluate their performance. Network flow-based detection, particularly using REPTree and Random Forest models, outperform syscall-based methods with nearly perfect F-measure scores, making it more suitable for large-scale, distributed environments like cloud computing and SDNs. The study's strengths include its high accuracy rates and practical relevance, using real-world CVEs, but it is limited by its focus on detection (not

prevention), the use of a single vulnerable application and an imbalanced dataset. Though not specifically aimed at SDN, the findings on scalable network flow-based detection offer valuable insights for improving security in SDN and cloud contexts.

3.7 Kubernetes Dataset for Misuse Detection [7]

The paper primarily focuses on creating a dataset for misuse detection in Kubernetes environments, without directly proposing an intrusion prevention model. The dataset was generated using a microservice-based application deployed in Kubernetes, incorporating 10 different attack scenarios. Its primary contribution is providing publicly available, realistic data that can be used to develop and evaluate security solutions for cloud and SDN environments. The preliminary analysis using a one-class Support Vector Machine (SVM) achieved high performance with an accuracy of 96.57% and an F1 score of 98.25%. The paper's strengths include the diversity of attack scenarios, use of real-world vulnerabilities and relevance to Kubernetes. However, it is limited by its narrow scope, lack of a prevention model and basic analysis. Despite these limitations, the dataset serves as a valuable resource for advancing intrusion detection and prevention in containerized environments.

3.8 ProtÉdge: A few-shot ensemble learning approach to software-defined networking assisted edge security [8]

This paper proposes an SDN-based intelligent security model for edge-enabled IoT networks, focusing on intrusion detection and prevention. It employs an ensemble of Prototypical Networks and SVMs to detect intrusions with minimal training data. Evaluation on the Bot-IoT, UNSW-NB15 and a custom SDN dataset shows superior performance compared to traditional models, particularly in few-shot scenarios. The model's key advantages include high accuracy, adaptability to new threats and low processing overhead, making it suitable for resource-constrained edge environments. However, it presents challenges in implementation complexity, potential false positives and reliance on SDN infrastructure. While the system excels in SDN environments with real-time detection and dynamic mitigation capabilities, its scalability

to larger, real-world networks remains limited.

3.9 Unsupervised Anomaly Detection for Container Cloud Via BILSTM-Based Variational Auto-Encoder [9]

This paper presents a security solution for container environments using a hybrid BiLSTM-VAE detection system. The system, named Pudding, enables real-time anomaly detection by analyzing system call sequences without requiring labeled data. It effectively captures long-term dependencies in system calls, enhancing the detection of subtle anomalies. By leveraging the Linux ptrace mechanism, it monitors container behavior non-intrusively, maintaining minimal performance impact (8.41%). While not a traditional IDS/IPS, Pudding functions similarly, continuously evaluating system behavior to identify security risks. Compared to other unsupervised methods like LOF and one-class SVM, it demonstrates higher accuracy and robustness against evolving attack patterns. Ultimately, Pudding offers an unsupervised, scalable and efficient solution for intrusion detection in microservices architectures, particularly where signature-based IDS methods may be less effective.

3.10 Securing Communication and Identifying Threats in Cloud Microservices [10]

This paper examines the security challenges in cloud microservices and evaluates the performance of traditional IDS, particularly Zeek, in these environments. It emphasizes that microservices with their complex communication patterns and expansive attack surfaces present difficulties for conventional IDS approaches. The study includes a detailed analysis of Zeek in both monolithic and microservice-based architectures, highlighting its strength in logging network activities and detecting specific attacks. However, the paper identifies limitations in traditional IDS for microservices and proposes the use of service meshes like Istio. Istio's use of mutual TLS (mTLS) and centralized security management for traffic control, authentication and authorization is presented as a solution for enhancing microservice security. While Zeek remains effective for detecting certain threats, service meshes provide more comprehensive

protection through a zero-trust security model.

3.11 Automated Traces-based Anomaly Detection and Root Cause Analysis in Cloud Platforms [11]

This paper introduces a framework for automated anomaly detection and root cause analysis (RCA) in cloud platforms using kernel traces. By modeling the dependencies between cloud components, the framework detects anomalies and identifies the underlying causes of system faults. It is designed for cloud environments like Kubernetes and is tested on a key-value store application, demonstrating high accuracy (80%-99.3%) in both detecting and diagnosing faults. The system operates at the kernel level, monitoring system calls independently of specific applications, enabling fault detection across distributed microservices without intrusive monitoring. Though not a dedicated IDS/IPS, the framework effectively serves as an anomaly detection tool that correlates abnormal behavior with system faults. Its key advantages include high detection accuracy, application independence and suitability for managing complex cloud systems like Kubernetes.

3.12 ContainerGuard: A Real-Time Attack Detection System in Container-Based Big Data Platform [12]

This paper introduces ContainerGuard, a real-time attack detection system designed for container-based big data platforms, focusing on vulnerabilities associated with Meltdown and Spectre attacks that can leak sensitive data across co-resident containers. By employing variational autoencoders (VAEs), ContainerGuard monitors multivariate time-series performance data to identify anomalies indicative of such attacks. Its non-intrusive monitoring capability allows it to gather performance data without altering the host or container environments, while an ensemble approach enhances detection accuracy with minimal overhead of approximately 4.5%. The system demonstrates impressive detection performance, achieving an AUC score of 0.9917. Although ContainerGuard does not fit the traditional definitions of an IDS or IPS, it continuously monitors for abnormal patterns, effectively detecting poten-

tial exploits and providing strong resilience against noise, thereby ensuring robust security in containerized environments.

3.13 Container Anomaly Detection Using Neural Networks Analyzing System Calls [13]

This paper introduces a novel method for anomaly detection in containerized applications by employing neural networks to analyze system calls. It develops two detection techniques: one focused on the distribution of system calls and the other on the file and directory paths utilized by those calls. The methods are evaluated against scientific computing workloads, including OpenFOAM and FLUKA, in conjunction with simulated malicious activities. By capitalizing on the unique features of containers, such as host-level monitoring without additional agents, the approach offers enhanced security relevant to microservices and container environments. However, the authors caution that this method may produce false positives and recommend its integration with other security measures for a comprehensive solution.

3.14 A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest [14]

This paper introduces an online anomaly detection system tailored for Docker containers, utilizing an optimized Isolation Forest algorithm to analyze multidimensional resource metrics. Key contributions include the development of a weighted feature selection method to enhance detection accuracy by accounting for application workload characteristics, as well as the implementation of dynamic monitoring period adjustments to minimize detection delays and system overhead. Additionally, the system incorporates log analysis to identify the causes of anomalies. Functioning as a host-based intrusion detection system, it leverages specific container characteristics and resource usage patterns to bolster security monitoring in containerized settings. Overall, the system effectively detects various types of anomalies—such as CPU, memory, disk I/O and network issues—demonstrating strong performance in both simulated and real cloud environments.

3.15 Leveraging Computational Intelligence Techniques for Defensive Deception: A Review, Recent Advances, Open Problems and Future Directions [15]

This paper offers an in-depth review of computational intelligence techniques, specifically focusing on ML and DL, in the realm of defensive deception for cyber security. It evaluates various strategies, such as honeypots, moving target defense and obfuscation, emphasizing how these can be improved through the integration of AI technologies. The authors categorize defensive deception methods and explore implementation strategies while identifying open challenges and future research avenues in AI-enabled defense mechanisms. Although not centered on container security, the discussed deception techniques, including honeypots and moving target defense, are applicable to containerized settings. Furthermore, the paper highlights the potential of ML and DL to enhance IDS and IPS, suggesting effective applications for attack pattern recognition and anomaly detection in container environments.

3.16 A Comprehensive Analysis of Machine Learning and Deep Learning-Based Solutions for DDoS Attack Detection in SDN [16]

This study explores ML and DL techniques for detecting DDoS attacks in SDN. The core objective of the proposed intrusion prevention framework is to identify DDoS attacks by employing a range of ML and DL algorithms. These solutions harness the centralized control and enhanced programmability of SDN, facilitating the implementation of anomaly detection methods in network traffic. The integration of ML/DL models improves detection accuracy by analyzing traffic patterns. The centralized SDN architecture supports the deployment and management of these models throughout the network. Advanced DL models, such as Long Short-Term Memory (LSTM), are adept at recognizing complex traffic patterns, thus enhancing the detection of sophisticated attacks. Nevertheless, SDN faces vulnerabilities, particularly controller overload due to the large volume of unmatched packets.

3.17 An integrated SDN Framework for Early Detection of DDoS Attacks in Cloud Computing [17]

The study proposes an SDN-based framework that is named RDAER. It is used for early detection of DDoS attacks by monitoring SDN switches and identifying network anomalies before reaching the controller. Combining ML and rule-based event correlation, the model achieved 99.92% accuracy and a detection time of 20 seconds using the CICIDS 2019 dataset. While outperforming other methods in reducing detection time and false positives, the framework faces challenges such as computational overhead and scalability issues as network complexity increase. Further tuning is needed to maintain accuracy in diverse environments.

3.18 Detection and Prevention of DoS attacks in Software-Defined Cloud Networks [18]

This paper introduces a distributed firewall with an IPS for Software-Defined Clouds that monitors traffic at both SDN switches and the controller to detect and prevent DoS attacks, such as ICMP and SYN flooding. The model uses traffic flow analysis and signature-based detection to filter out malicious packets and offers real-time threat mitigation. While effective in simulations, its reliance on signature databases could limit its defense against novel attacks and real world implementation may reveal additional challenges.

3.19 DDoS attack Mitigation in SDN-based Cloud Computing Environment [19]

This paper presents a flow-table sharing mechanism to defend against flow table overloading attacks in SDN-based cloud environments where attackers target the limited entries in OpenFlow switches. By redistributing flow entries across multiple switches, the model prevents any single switch from being overwhelmed. Thus, it reduces controller communication overhead and enhances scalability. However, this approach may introduce latency requiring further optimization for complex network topolo-

gies.

3.20 Efficient Distributed DoS Attack Defense in SDN-based Cloud [20]

This study introduces a hybrid ML model that integrates Support Vector Machine (SVM) and Self-Organizing Map (SOM) algorithms along with an enhanced History-based IP Filtering (eHIPF) scheme. The proposed model improves DDoS attack detection in SDN-based cloud environments. The model focuses on identifying anomalous IP traffic patterns, reducing controller overload and safeguarding service function chaining. While it offers effective scalability and real time detection capabilities, the model requires significant computational resources for training and may need regular updates to maintain its effectiveness in dynamic environments.

3.21 Mitigating DDoS Attacks using SDN-based Network Security Measures [21]

This paper introduces an SDN-based framework aimed at mitigating DDoS attacks through centralized control and network programmability. By utilizing real-time traffic monitoring and dynamic rerouting, the model effectively detects and isolates malicious traffic, preventing it from overwhelming the network while allowing legitimate traffic to flow unobstructed. The framework shows significant improvements in attack detection and response times compared to traditional systems, although it emphasizes the importance of securing the SDN controller and addressing scalability challenges in large networks.

3.22 A DDoS Attack Detection and Mitigation With Software-Defined Internet of Things Framework [22]

The paper introduces a Software-Defined Internet of Things (SD-IoT) framework designed to detect and mitigate DDoS attacks. By leveraging the centralized control and programmability of SDN, the model enables real-time management of IoT devices.

The detection mechanism utilizes cosine similarity to analyze packet-in message rates at software-defined IoT (SD-IoT) switch ports, identifying anomalous traffic patterns indicative of DDoS attacks. Experimental results demonstrate that the model can effectively detect these attacks in real-time, significantly reducing the volume of packet-in messages directed to the SD-IoT controller. The model's strengths include high detection accuracy and efficient traffic management, while its complexities include implementation challenges and reliance on centralized control. It emphasizes the importance of securing IoT devices, which often lack robust security features. Overall, the framework offers a scalable solution for defending IoT networks against DDoS threats.

3.23 Blockchain as a Service for Software Defined Networks: A DoS Attack Perspective [23]

This paper introduces BlockSDSec, a Blockchain as a Service model that aims at enhancing the security of SDN against DDoS attacks. By integrating blockchain technology into the data and control planes, the model secures communication between switches and the controller, preventing overload from malicious entities. It utilizes virtual controllers for load distribution and blockchain for verifying OpenFlow entries. As a result, this method effectively mitigates DDoS attacks with minimal latency. While offering decentralized security and resilience, the model faces challenges related to latency and blockchain management complexity. Overall, BlockSDSec provides a robust solution for strengthening SDN against common vulnerabilities.

3.24 Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments [24]

This paper introduces Brew, a security policy analysis framework for OpenDaylight SDN controllers that focuses on detecting and resolving security policy conflicts in distributed SDN-cloud environments. It ensures a consistent implementation of security policies by preventing conflicts through global prioritization and cross-layer conflict detection. Evaluations show near-linear scalability with up to 100,000 atomic

flow rules, achieving a worst-case runtime of O(n) for conflict detection. While Brew automates conflict resolution and effectively manages complex networks, it still requires manual input for interpretative conflicts. Overall, Brew enhances SDN security by maintaining a global view of flow rules, mitigating risks associated with policy conflicts.

3.25 DDoS Attack Mitigation in IoT Using SDN [25]

This paper introduces a model for mitigating DDoS attacks in IoT environments using SDN. The model leverages SDN's centralized control and real-time traffic monitoring to detect and mitigate attacks more effectively than traditional methods that rely on packet sampling. Its key strengths include high detection accuracy, reduced network congestion and scalability across large IoT deployments. However, the model requires significant computational resources and may encounter challenges in non-SDN or hybrid networks. Additionally, efficient inter-domain communication between SDN controllers is necessary to prevent delays in cross-domain attack detection. Despite these drawbacks, the model provides an effective solution for protecting IoT networks from DDoS threats.

3.26 Flow Based Security for IoT Devices Using an SDN Gateway [26]

This paper presents a flow-based security model for IoT devices using an SDN gateway, focusing on real-time detection and mitigation of TCP and ICMP flood attacks. The model leverages SDN to analyze traffic patterns dynamically and block malicious flows. Thus, it enhances IoT security without requiring additional hardware. While the system shows promise in detecting and mitigating attacks, it experiences a decrease in flow installation rates which could limit its performance in larger networks. Furthermore, the model primarily addresses basic attack types and may require further development to tackle more sophisticated threats. Overall, the SDN gateway offers an effective solution for securing small-scale IoT networks, though its reliance on the SDN controller presents scalability challenges.

3.27 Information Classification Strategy for Blockchain-based Secure SDN in IoT Scenario [27]

The paper presents a blockchain-based SDN monitoring system aimed at improving security in IoT environments. Blockchain's decentralized and immutable properties protect SDN from unauthorized access and tampering. An information classification strategy boosts security by grouping data according to importance. A multicast routing algorithm is also integrated to balance the load and optimize traffic flow. However, the model has limitations, such as potential latency from blockchain processing and high computational demands, which may affect its scalability and real-time performance in large or resource-limited IoT networks.

3.28 Learning-Driven Detection and Mitigation (LEDEM) of DDoS Attack in IoT via SDN-cloud Architecture [28]

This paper presents the LEDEM model designed to detect and mitigate DDoS attacks in IoT environments. It uses a semi-supervised ML approach within an SDN-cloud architecture. LEDEM achieves high detection accuracy (96.28%) and improves throughput by 21% in comparison to other models. Its decentralized control approach enhances scalability and real-time response capabilities, making it well-suited for resource-constrained IoT networks. However, its performance is reliant on the quality of the training data and managing decentralized controllers can be complex. The paper emphasizes the importance of mitigating controller overload while maintaining network security and efficiency.

3.29 Defending Against Packet-In Messages Flooding Attack under SDN Context [29]

This paper addresses the challenge of defending SDN controllers against Packet-In message flooding attacks, a form of DDoS attack. It introduces an intrusion detection model that combines Bayesian Networks and fuzzy c-means clustering to classify compromised switches and detect anomalies. Effectively, it reduces the load on SDN

controllers by limiting the number of monitored switches. Through simulations, the model demonstrates low overhead, high accuracy and a significant reduction in CPU usage and network traffic. However, it has a false positive rate of 19.64% and may struggle with large-scale attacks involving multiple compromised switches. Despite these limitations the model offers a promising approach to mitigating SDN-based DDoS attacks, though further refinement may be needed for deployment in complex real-world networks.

3.30 Security Analysis of SDN Cloud Applications [30]

The paper provides a comprehensive evaluation of security vulnerabilities in three SDN-based cloud applications: Meridian, CloudNaaS and HPE Virtual Cloud Network (VCN), using the STRIDE threat model to assess risks such as spoofing, tampering and DoS. The applications employ various security mechanisms like encryption and authentication protocols to address these threats. However, their reliance on external solutions such as TLS and Kerberos introduces computational overhead. While centralized SDN control facilitates scalable security and real-time threat response, certain vulnerabilities remain, especially related to DoS attacks and repudiation in systems like CloudNaaS. The paper highlights both the strengths, including virtualization-based security and scalable threat management and the weaknesses, notably the risk of controller flooding and performance degradation in high-traffic environments. Ultimately, it provides mitigation strategies but recognizes the persistent risks inherent in SDN architectures.

3.31 Blockchain and Random Subspace Learning-Based IDS for SDN-Enabled Industrial IoT Security [31]

This paper proposes a security framework combining Blockchain and SDN to safeguard Industrial IoT systems. It employs an ML-based IDS named RSL-KNN, alongside a Blockchain-based Integrity Checking System to detect forged commands and prevent misrouting attacks. The model shows high detection accuracy particularly with a 96.73% rate for binary classifications and ensures 100% integrity for SDN

flow rules. However, computational overhead and synchronization delays due to blockchain could deteriorate system performance. The model's scalability for large industrial control systems coupled with low detection time, makes it suitable for defending distributed SDN environments, though false positives remain an area for improvement. Overall, it provides a robust defense against cyber attacks on command integrity in SDN-enabled Industrial IoT.

3.32 Detection and Mitigation of DoS and DDoS Attacks in IoT-Based Stateful SDN: An Experimental Approach [32]

This paper explores an entropy-based detection and mitigation mechanism for addressing DoS and DDoS attacks in IoT networks using stateful SDN. By calculating entropy variations, it detects anomalies in traffic patterns and mitigates attacks through intelligent flow management at the switch level, minimizing reliance on the SDN controller. The model shows efficiency in real-time detection and mitigation, with low computational complexity and scalability for IoT environments. However, the system's performance depends on careful parameter tuning to avoid false positives and it may struggle against more sophisticated attacks. Overall, the paper presents a promising approach but requires further refinement for broader application.

3.33 Securing Distributed Systems: A survey on Access Control Techniques for Cloud, Blockchain, IoT and SDN [33]

This paper offers a comprehensive survey of access control techniques across distributed systems, including cloud computing, blockchain, IoT and SDN. Although it does not focus on a specific IPS model, it explores how access control mechanisms such as Role-Based Access Control and Attribute-Based Access Control, enhance security by preventing unauthorized access. For SDN, these techniques improve network management and guard against API attacks, while cloud environments benefit from fine-grained access control and regulatory compliance. The strengths of these models include scalability and security flexibility but challenges arise in managing complex policies and high-volume environments. Overall, the paper highlights the

role of access control in safeguarding distributed systems without directly addressing IPS models.

3.34 A Review on Cyber security of Cloud Computing for Supporting Connected Vehicle Applications [34]

This paper offers a comprehensive overview of the cyber security challenges, existing solutions and future research directions in cloud computing for connected vehicle (CV) applications. While it does not focus on a specific IPS model, it discusses various security measures and technologies such as blockchain, AI and elliptic-curve cryptography that could enhance security in these environments. One strength is its broad coverage of both traditional and emerging cyber security approaches, but it lacks a unified model for intrusion prevention and raises concerns about scalability in real-world CV applications. The paper highlights benefits for SDN, including role-based access control and improved network management while also discussing SDN-specific security measures such as flow-based access control and decentralized authentication via blockchain. Overall, the paper serves as a valuable resource for understanding the current state of cyber security in CV-related cloud computing but falls short in proposing a developed implementable model.

3.35 Powering Hands-on Cybersecurity Practices with Cloud Computing [35]

This paper introduces EZSetup, a cloud-based platform designed to facilitate hands-on cyber security education rather than an intrusion prevention model. While some labs address topics like intrusion detection and prevention, the focus is on providing a scalable and flexible environment for cyber security training using cloud resources. The platform's advantages include its scalability, flexibility, accessibility and cost-effectiveness, though it depends on cloud infrastructure and lacks a detailed security model. While the paper does not directly address SDN security, it enables learning about SDN and cloud technologies, indirectly contributing to security knowledge in these areas. Overall, EZSetup enhances cyber security education by simplifying access to practical, cloud-based learning environments.

3.36 Securing Cloud, SDN and Large Data Network Environments from Emerging DDoS Attacks [36]

This paper examines an automated intrusion prevention model aimed at mitigating DDoS attacks in cloud, SDN and large-scale network environments. The model, Radware's Attack Mitigation System, particularly highlights its DefensePro and DefenseFlow components, which offer rapid DDoS detection and mitigation. While the system's automation, scalability and use of behavioral analysis are advantageous for handling unknown threats, for large distributed networks, it relies on proprietary technology and lacks detailed performance metrics or comparisons with alternative solutions. The integration with SDN environments via DefenseFlow enables effective traffic redirection for attack mitigation, though specific details on SDN protection are limited. Overall, the paper introduces a potentially effective solution, though its real-world performance would benefit from more extensive testing and evaluation.

3.37 SDN-Enabled Hybrid DL-Driven Framework for the Detection of Emerging Cyber Threats in IoT [37]

This paper introduces an SDN-enabled hybrid DL framework designed to detect cyber threats in IoT environments, employing CuDNNLSTM and CuDNNGRU models that achieve impressive performance metrics, including 99.74% accuracy and a fast detection time of 9.33 ms. Its key strengths lie in its high detection accuracy, scalability and integration with SDN's centralized control, providing an efficient means of securing cloud-based IoT applications. However, the model emphasizes detection over prevention and requires substantial computational resources for training, making it less applicable to real-world IoT deployments without further optimization. The framework's reliance on SDN's programmability and global network view enhances security by enabling dynamic policy updates and efficient threat identification, though concerns remain regarding overreliance on the centralized SDN controller. Overall, the paper offers a significant contribution to IoT security but would benefit from additional focus on preventive measures.

CHAPTER 4

METHODOLOGY

Container networks built with microservice architectures have gained increasing popularity in recent years. This widespread adoption, however, has also made container networks a target for cyber attacks. When such attacks occur, containers can no longer provide reliable services, leading to response delays on the client side. In some cases, clients may not receive responses at all. Figure 4.1 illustrates a representative container network. On the left, two clients from different nodes are seen sending requests to the network. Pod1 and Pod2 are the containers forming this network. In the absence of an attacker, Pod1 and Pod2 operate normally. Figure 4.2, on the other hand, depicts a scenario in which an attacker has infiltrated the network, specifically targeting Pod1, though no attack has commenced yet. Subsequently, the attacker within the network can launch any type of attack, such as a DoS attack, on Pod1. Figure 4.3 shows the scenario in which the attacker has successfully attacked Pod1, causing it to fail and preventing clients from receiving responses to their requests.

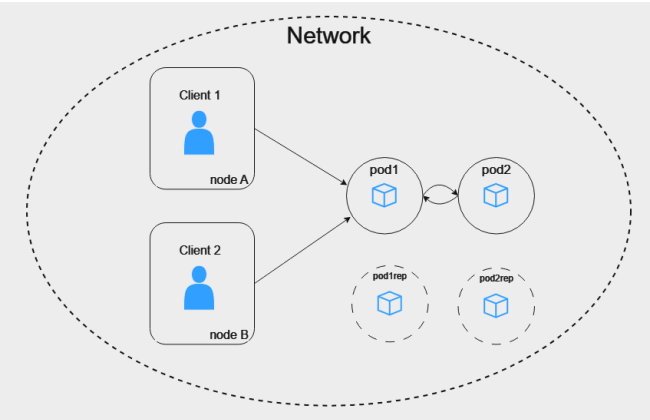


Figure 4.1: Containers with no attacker in the network

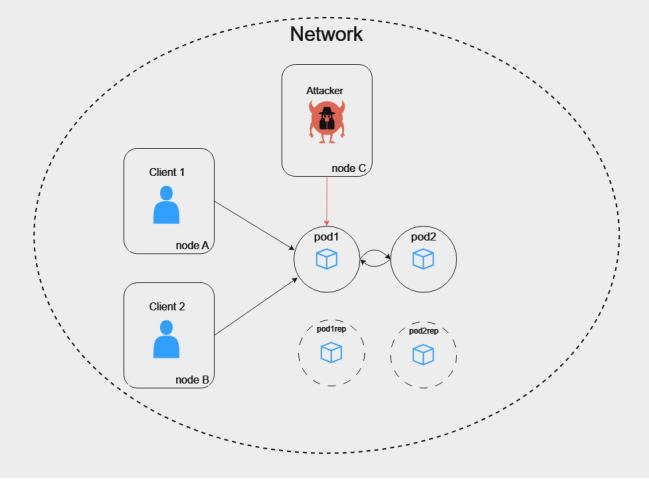


Figure 4.2: Attacker targeting pod1

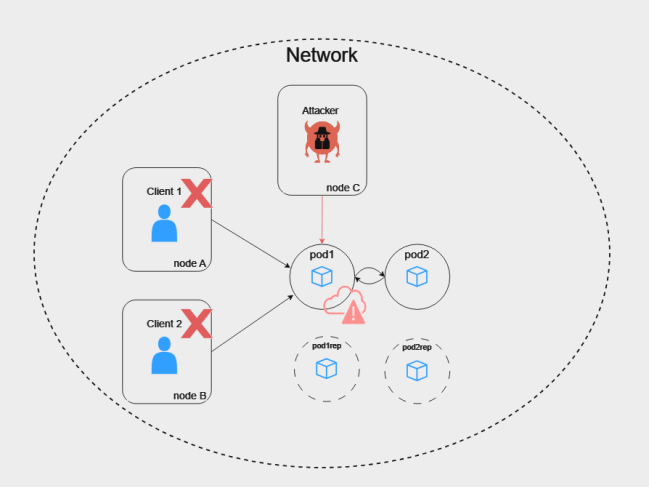


Figure 4.3: Pod1 under attack

To address this problem, we propose an IPS, which involves introducing a replica of the targeted pod into the network and converting the targeted container into a honeypot. Figure 4.4. demonstrates the actions taken by the IPS against the attacker and the compromised container. After the update, clients can send requests through pod1's replica (pod1rep) and reach pod2 without issues. Meanwhile, the attacker, unaware of the update, continues sending requests to the compromised container, now designated as a honeypot.

Figure 4.5 presents an overview of the proposed IPS model. The diagram includes three serially connected circles, symbolizing pods within the Kubernetes environ-

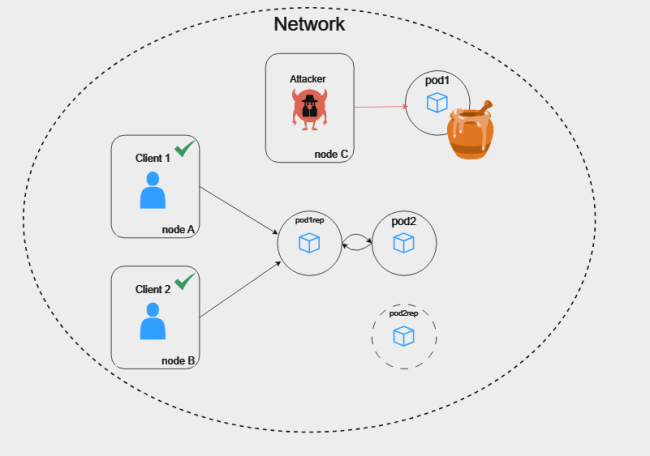


Figure 4.4: Blocking the attacker

ment. Inside each pod, blue and green cubes represent containers: the blue container serves as a sidecar, capturing network traffic generated by the green container, which handles primary communication. This configuration of three serially connected pods demonstrates the microservice architecture responsible for transmitting and returning client requests across the network. Additionally, replica pods are pre-configured to ensure that these three pods are unique, so if an attack occurs, the targeted pod is removed and replaced with a replica, isolating all replicas using a custom network policy.

The structure shown at the bottom left in Figure 4.5 represents the Log Collector module, also referred to as "fetch log" in the experiments. Its primary function is to retrieve network traffic logs stored by the sidecar containers locally. This data is then fed into the IDS module, which could be based on machine learning, deep learning, or a hybrid model. By processing the network traffic data, the IDS identifies attackers and risky pods in the network. Then, it sends this information to the custom IPS module, also referred to as HM in later sections. The IPS module updates the network based on this intelligence by replacing the risky pod with its replica, achieved through pod label updates. Subsequently, it designates the risky pod as a honeypot, creating a decoy for the attacker through a NAT table update on the node under attack.

The general operation of this model is outlined, with more detailed explanations of specific processes provided in the following subsections.

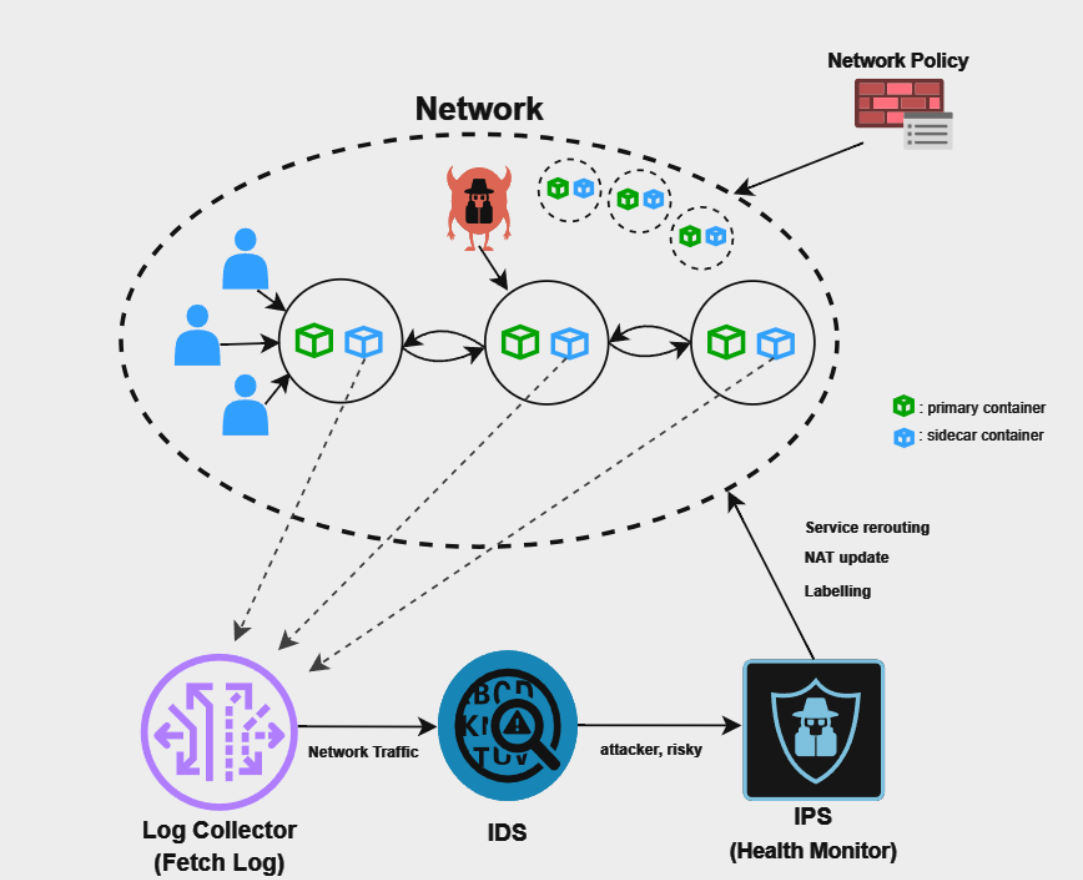


Figure 4.5: System model

4.1 Pod Labeling

To achieve this automated structure, specific labels were predefined in the configuration files prior to pod creation. These labels are crucial for facilitating attack detection and subsequent network updates. For instance, to distinguish between main and backup pods, the **type** labels in the configuration were designated as either **main** or **backup**. While the main pods operate within the network, the backup pods were intended to be integrated only if the corresponding main pod was attacked. A network policy was established to block the communications of backup pods until needed as shown in Figure 4.5. Additionally, both primary and backup pods shared the same **app** label (e.g. pod1), enabling the system to identify which pod's replica should be activated during an attack. Furthermore, **role** labels were used to assign each pod's specific function within the network, playing a vital role in identifying the attacker

and the risky pod. For example, when a risky pod is identified, its role label is updated to indicate its new status as a honeypot. Following this update, the honeypot is isolated from the network. A complementary network policy was also developed, allowing the attacker identified by the role label to communicate exclusively with the honeypot, while blocking any interaction with other pods.

4.2 Tcpdumper (Sidecar Container)

Following the labeling of pods, investigations were conducted to identify methods for detecting attacker and risky pods within the network. Based on these analyses, it was determined that decisions would be derived from the network traffic logs associated with the relevant pods. To achieve this, tcpdumper containers (sidecar containers) were deployed alongside the main containers within the pods. The tcpdumper containers are dedicated to capturing and storing the network traffic of the main container by filtering all packets directed to or originating from the pod's IP address. In the development of tcpdumper containers, a Bash script was employed for their preparation. Once the container collects 100 logs, captured logs are saved in a designated directory and this cycle persists indefinitely. Input parameters are optimized for tcpdumper to extract more comprehensive information regarding the captured packets. The attributes of the network packets include the timestamp, network interface, packet direction (inbound/outbound), source IP, destination IP, TCP flags (such as Synchronize and Acknowledge flags), window size, TCP options (including maximum segment size, selective acknowledgment, TCP timestamps, window scaling factor) and TCP payload length. Furthermore, the logs are generated using system time rather than relative time. This distinction is crucial, because it ensures that when the stored files are merged the timestamps remain unique. Thereby, it enables to observe network flows to identify the potential attackers. Overlapping time values would otherwise compromise this identification process.

Subsequently, two distinct modules, termed **FetchLog** and **Health Monitor** were developed to automate the processes of attack detection and prevention by utilizing the recorded network traffic data and corresponding pod labels.

4.3 Fetch Log Module (Log Collector)

The FetchLog module operates by periodically retrieving network traffic logs recorded by sidecar containers as depicted in Figure 4.5. It transfers all logs to the local repository. After logs are fetched, the log directory in the tcpdumper containers are cleared to prevent memory overflow. The retrieved network traffic files are organized into distinct folders named after their respective pods. The periodically collected logs are merged to reflect the network traffic associated with each pod. These logs are utilizable by the intrusion detection system.

In the initial design phase of the FetchLog module, the pcap files generated by the tcpdumper containers were decoded post-retrieval. Consequently, the FetchLog module was responsible for both fetching and decoding, resulting in reduced performance in high-traffic network environments. To address this limitation, a design modification was implemented, whereby tcpdumper containers would directly record the network traffic in a decoded text format. This modification allowed the FetchLog module to focus solely on retrieving the recorded network traffic files. As detailed in the Experiments section, this update resulted in a significant enhancement in performance.

4.4 Health Monitor Module (IPS)

The primary function of the Health Monitor (HM) module is to continuously monitor the pods' status and initiate updates when attacker and risky pods are identified, as shown in Figure 4.5. It is assumed that the IDS module detects risky and attacker pods using the flow data which is received from the FetchLog module. The role labels of the relevant pods are updated as attacker and risky based on this information. While monitoring the network, the HM module detects pods with updated labels. It saves the IP address of the attacker pod along with the node it is on and its node zone. It then modifies the NAT table of the attacker pod's node to redirect requests from the attacker pod to the risky pod with the help of PRE and POST routing rules. Consequently, requests previously directed from the attacker pod's IP to the service of the risky pod are now routed directly to the risky pod's IP via the attacker node's IP. Moreover, the endpoint of the risky pod service is designated as its backup pod, with

its type label updated to *main* for inclusion in the network. This ensures that other pods or clients interacting with the relevant pod can continue their operations without disruption. Upon completion of the rerouting process, the HM module classifies the attacker pod as a known-attacker and the risky pod as a honeypot, indicating that appropriate operations have been performed and mitigating the need for repetitive actions regarding these pods.

In the initial design of the HM module, monitoring encompassed various pod attributes, including pod name, pod IP, service IP, service name and node name. However, during experimentation, it became evident that as the number of pods increased, the HM module faced challenges in effectively monitoring the network, leading to performance degradation. Consequently, a design revision was made, whereby the HM module would focus exclusively on monitoring pods based on their role labels. This decision was grounded in the understanding that the other attributes were only pertinent for pods categorized as attackers or risky. This targeted approach yielded significant enhancements in the efficiency of network updates.

CHAPTER 5

EXPERIMENTS

In this section, the performances of the developed model and sub-modules are analyzed based on experiments conducted under various scenarios. Google Kubernetes Engine (GKE) was used as the test environment for all experiments.

5.1 Fetch Log Module - Initial Version Analysis

In these experiments, the performance of the initial version of the fetch log module was evaluated for varying test durations and pod counts.

5.1.1 Varying Test Duration

In this experiment, the performance of the initial version of the **fetch log** module, which sequentially executes both fetching and decoding operations was evaluated. To assess its performance, custom-built containers were utilized, which are server, client and tcpdumper. Each pod in the setup contained either a server or client container, along with a tcpdumper container. Communication between the client and server was established via the server service, where clients generated random messages and sent them to the server service IP at 5-second periods. After the server received messages, it returned a response to the client. Throughout the communication, the tcpdumper containers captured network packets by filtering them based on the pod IP address. Once 100 packets were captured, they were saved under a specific folder in the pod. The packets, stored in pcap format, were subsequently fetched and decoded by the fetch log module. To facilitate scalability in the number of clients, a deployment

configuration file was prepared.

Upon successful setup of the test environment, the performance of the fetch log module was evaluated. The parallel process limit of it was set to 10. Parallel process limit indicates how many fetching and decoding operations the fetch log module can perform in parallel at the same time. Initially, 9 client pods and 1 server pod were deployed with tcpdumpers. After the configuration files were applied, we waited for 10, 15, 20, 30, 60, 90 and 120 seconds respectively. At the end of each wait, the fetch log module was executed. Fetching and decoding times of all pcap files within the cluster were saved. The experiment was then repeated with 20-pod (19 clients and 1 server) and 50-pod (49 clients and 1 server) scenarios. Each test scenario was repeated three times and the average times were documented in Table 5.1. The measurement time indicates how many seconds after the pods creation command is entered the fetch log module is triggered.

Table 5.1: Fetch Log Module Performance with Initial Design

		Measurement Time (s)						
		10	15	20	30	60	90	120
Number of pods	10	5.474	5.508	6.231	6.794	9.631	12.740	14.967
	20	10.182	10.690	11.790	12.990	18.508	22.846	31.461
	50	20.442	22.100	23.385	27.663	36.658	45.833	50.983

Upon examining the experiment results, an important finding emerged. When the number of pods increased and measurement times were short, certain pods failed to generate pcap files. For instance, with the 20-pod scenario, some pods could not produce any pcap file within 30 seconds. Because, the number of pods was low and there was not enough network traffic in the selected measurement time. Therefore, the subsequent experiments were conducted with this observation in mind.

As another result of this experiment, it was observed that the fetch decode time increases as the test duration increases, but this increase appears to be more evident

especially after 30 seconds. The reason for this phenomenon is that all pods were now able to create at least one pcap. Thus, the number of pods became meaningful for the experiments after 30 seconds.

5.1.2 Varying Pod Count

In this experiment, the performance of the fetch log module was observed for increasing number of pods. Based on the findings from the previous experiment, the measurement time was set as 60 seconds. That is, the fetch log module was triggered 60 seconds after the pods were created. The parallel process limit remained at 10. The communication period between the clients and server was set to 5 seconds again. After waiting for 60 seconds, the fetch log module was executed and the total time of fetch and decode operations was recorded in Table 5.2.

Table 5.2: Fetch Log Module - Initial Version Performance Analysis Based on Pod Count

	Number of pods							
	2	4	10	25	50	100	200	500
Time (s)	4.788	5.713	9.500	23.445	42.892	71.965	126.641	238.451

An important observation was made during the experiments. For the 100-pods scenario, all pods transitioned to the running phase within 30 seconds. However, in the 200-pods scenario, this transition took 53 seconds. Furthermore, in the 500-pod scenario, some pods with an **OutOfPods** status were found. This indicates that as the number of pods increased, GKE was unable to create all the pods immediately. Therefore, the scenarios with 200 and 500 pods do not accurately reflect real world conditions.

5.2 Fetch Log Module - Updated Version Analysis

In the initial design, the fetch log module fetched and decoded the pcap files created by tcpdumper containers. This design could not achieve high performance, since the decode operation required extra work. Consequently, it was made a decision to revise the design. For this purpose, tcpdumper containers were employed to save network packets in "txt" format instead of "pcap"s format. As a result, the fetch log module focused solely on fetching. Due to the design update, the experiments in the previous section were repeated with the new design.

5.2.1 Varying Test Duration

In this experiment, the performance of the updated fetch log module, which now only handles log fetching, was evaluated. Custom-built containers, including server, client and tcpdumper were used. Each pod contained either a server or client container, along with a tcpdumper container. Client-server communication was established through the server service, where clients generated random messages and sent them to the service IP in every 5 seconds. Upon receiving the messages, the server responded with the requested content. Throughout the communication, tcpdumper containers captured network packets, filtering based on the pod IP. After capturing 100 packets, they were saved in a designated folder in "txt" format and fetched by the fetch log module. A deployment configuration file was prepared to allow scalability in the number of clients.

Once the test environment was set up, the fetch log module's performance was evaluated with a parallel process limit of 10. Parallel process limit indicates how many fetching operations the fetch log module can perform in parallel at the same time. Initially, 9 client pods and 1 server pod were deployed, all with tcpdumpers. After applying the configuration files, the system waited for 10, 15, 20, 30, 60, 90 and 120 seconds. At the end of each scenario, the fetch log module was executed and the fetching times for all "txt" files in the cluster were recorded. The experiment was repeated with 20 pods (19 clients and 1 server) and 50 pods (49 clients and 1 server). Each scenario was run three times and the average times were recorded in Table 5.3.

The measurement time in the table indicates how many seconds after the pod/pods creation command is entered the fetch log module is triggered.

Table 5.3: Fetch Log Module Performance with Updated Design

Number of pods \ Measurement Time (s)	10	15	20	30	60	90	120
10	5.123	4.957	5.178	5.820	5.247	5.353	5.274
20	8.824	9.233	8.428	8.904	9.839	9.100	9.738
50	19.314	19.147	18.941	19.655	19.611	19.322	19.964

When compared with the results of Experiment 5.1.1, a slight improvement was observed up to 30 seconds, regardless of the number of pods. However, the more significant improvement was seen after 30 seconds. At 60 seconds, the times were cut in half compared to the previous experiment. Even better, as the test duration increased, only minimal changes were observed. This suggests that whether the module runs for 60, 90 or 120 seconds, the duration remains the same, meaning the fetch log module can be executed over longer intervals without overloading the CPU.

5.2.2 Varying Pod Count

In this experiment, the performance of the updated fetch log module was analyzed as the number of pods increased. Following the results of the previous experiment, the test duration was fixed at 60 seconds. The parallel process limit remained set at 10, with the client-server communication frequency being every 5 seconds. After a 60-second waiting, the fetch log module was executed and the total time for the fetch and decode operations was recorded in Table 5.4.

When compared to Experiment 5.1.2, no significant improvement was observed when the number of pods was 2 or 4. However, when the pod count reached 10 or more, a substantial improvement was observed, with the time nearly halving. An exception was noted when the pod count reached 500. In this scenario, no significant improve-

Table 5.4: Fetch Log Module - Updated Version Performance Analysis Based on Pod Count

	Number of pods							
	2	4	10	25	50	100	200	500
Time (s)	4.219	4.435	5.297	12.914	19.646	36.333	73.460	226.408

ment was observed. As previously encountered, GKE was unable to create all the pods, resulting in an unhealthy cluster and a heavily loaded CPU. Therefore, it is more accurate to interpret the improvements for the cases in which pod counts are less than 500.

The fetched log sizes for each pod number are shown in Table 5.5.

Table 5.5: Fetched Log Sizes Based on The Number of Pods

	Number of pods							
	2	4	10	25	50	100	200	500
Size (KB)	92	200	564	1400	2500	4100	7400	17000

As the number of pods increased, the log size per unit time naturally increased, since the tcpdumper containers still create logs after the **fetch log** module is triggered. Thus, we should not make the mistake of thinking that all pods were recording 60-second logs. While some pods created 60-second logs, some can create 63-second logs and some can create 70-second logs. This is related to the information of which pod's log is processed first and which is processed last.

5.2.3 CPU and Memory Usage

After comparing the initial and updated versions of the fetch log module, it was decided to analyze the updated version in more detail. Thus, in this experiment, the CPU usage of the fetch log module and the total log sizes were evaluated for various

numbers of processes of the fetch log module. Parallel process number indicates how many fetching operations the fetch log module can perform in parallel at the same time. The communication frequency between client and server containers was set to every 5 seconds. Based on the inference from the previous experiment, it was assumed that a maximum of 100 pods could be created in the ideal scenario. Later, it was learned that when 100 pods in the cluster run for 60 seconds, a 4.1MB log file is created on average (while the process number is equal to 10). This experiment was performed with 100 pods. 100 pods were run for 60 seconds and traffic data was expected to be generated. A custom bash script was prepared for the experiment. It was employed for triggering the fetch log module after 60 seconds when the pod creation command was issued. Then, we analyzed how many seconds it takes for the fetch log module to fetch the traffic data, depending on the number of processes. The average file size and average CPU usage for each situation where the number of processes changes are available in Table 5.6. While file size was measured using Linux commands in the local folder, the kubectl top [46] command was used for average CPU usage.

Table 5.6: Fetch Log Module - Updated Version Analysis Based on Different Process Numbers

	Process Numbers								
	1	2	4	6	8	10	15	20	25
Fetching Time (s)	338.3	166.9	84.2	58.4	47.0	38.7	28.6	27.2	37.9
Avereage CPU (%)	2.6	6.1	15.6	23.1	33.6	42.6	60.6	88.1	95.6
Size (MB)	20	12	7.2	5.8	5.3	4.5	4.0	4.1	4.8

Before starting the experiment, the average CPU usage of the machine on which the logger would run was in the 0.7%-3.1% range. The resulting average CPU usages were average values obtained by subtracting this value from the last average CPU usage during the experiment.

While experimenting, it was realized that when the number of processes was 15, there was an instantaneous 75% CPU usage; when the number of processes was 20, CPU

usage was 100% at times and when the number of processes reached 25, we observed 100% CPU usage many times. The purpose of reporting these values is to show that average CPU usage and instantaneous CPU usage are different.

Looking at the 3rd row on the table, it was seen that the fetched log size decreases as the number of processes increases. Because, when the number of processes increases, the network traffic log is fetched faster. Therefore, it is fetched when the network traffic log has not yet increased sufficiently. Since the traffic log is smaller, the fetched log size is also smaller. This situation remained like this until the number of processes was 15. However, when the CPU usage was examined when the number of processes was 15 or more, it was determined that the node was not working properly. For this reason, it has been determined that the fetched log size, which is expected to decrease, does not decrease linearly. In the end, it was concluded that the process number threshold with which the system can operate properly is 10.

5.3 Node Performance Analysis Based On Pod Type

In Experiment 5.3.1, a server or client container alongside a tcpdumper container was included in the pods, and these were deployed on the same node. In Experiment 5.3.2, the tcpdumper containers were removed to assess their impact on communication between the server and client. Next, in Experiment 5.3.3, server and client pods were deployed on separate nodes and the resulting changes were observed. Finally, in Experiment 5.3.4, the number of server pods was increased to 2 and the servers were placed on a different node from the clients. The detailed effects of these configurations are described in the following experiments.

5.3.1 All Pods in The Same Node with Tcpdumper

In this experiment, the impact of increasing the number of pods on the node's CPU and memory usage was analyzed. Each pod was configured with server, client and tcpdumper containers and deployed on the same node. As the number of pods increased, only the number of client containers was scaled up, while the server count remained at one. Communication between the server and clients was set to occur every 5 seconds. After deploying the pods, a brief period was allowed for the system to stabilize. CPU and memory usage data were then gathered using the "*kubectl top node*" command, which updates itself every 15 seconds [46]. The recorded values represent the averages over these 15-second durations. The units returned by the command represent resource usage on the node, specifically CPU and memory. **Mi** represents memory usage in powers of two, while **m** denotes milicore.

In Table 5.7, the first column represents the average CPU and memory usage before any pods were deployed. The subsequent columns capture the average total CPU and memory usage values corresponding to the different pod counts. The primary reason for conducting the analysis based on total usage is to evaluate the overall performance of the node.

During the cluster stabilization period, it was noted that the instantaneous CPU usage exceeded the average levels. Specifically, the CPU usage was around 30% with 2 and 4 pods, increased to 55% with 10 pods, surged to 146% with 25 pods, reached 211%

Table 5.7: Node Performance Analysis Based on Pod Counts (with tcpdumper)

	Number of pods						
	Initial	2	4	10	25	50	100
CPU Core (m)	112	117	129	149	243	385	656
CPU (%)	12	12	13	15	24	39	66
Memory Bytes (Mi)	1451	1513	1556	1637	1591	1987	2680
Memory (%)	51	53	55	58	57	70	95

with 50 pods and was <unknown> when 100 pods were used. Consequently, it was deemed unnecessary to create more than 100 pods.

The CPU usage started at 12% and rose steadily as the number of pods increased. To ensure the proper functioning of the program, an average CPU usage of up to 10-30% is acceptable [47]. Since the HM module was not running during this experiment and considering that it would add some workload to the system. Thus, it is reasonable to limit the number of pods to a maximum of 25 on the same node based on these results. Creating more than 25 pods could lead to performance issues, potentially causing the application to malfunction due to the system operating beyond normal capacity.

On the other hand, as shown in the Table 5.7, memory usage did not exhibit any significant increase. Even if there is a slight rise, it does not present a major issue for the current design. This is because, after the fetch log module copies the log files recorded by the tcpdumper containers to the local folder, it deletes the files from the containers. Consequently, no memory bloating is expected during runtime.

Figure 5.1 and Figure 5.2 show the outputs of the "*kubectl top node*" command when pods were created. A 1m CPU usage is adequate for clients to communicate with the servers, but the server requires 17m CPU usage to handle all client interactions. Additionally, while the tcpdumper containers adjacent to the clients use 1m CPU, the tcpdumper recording the server's network packets shows a 12m CPU usage, which is

higher than that of the other tcpdumpers. It captured 6 times more network packets compared to other containers. The reason for this is that every client is trying to communicate with the server. In the end, a 0Mi value next to some tcpdumper containers suggests that these containers have not yet captured the network activity.

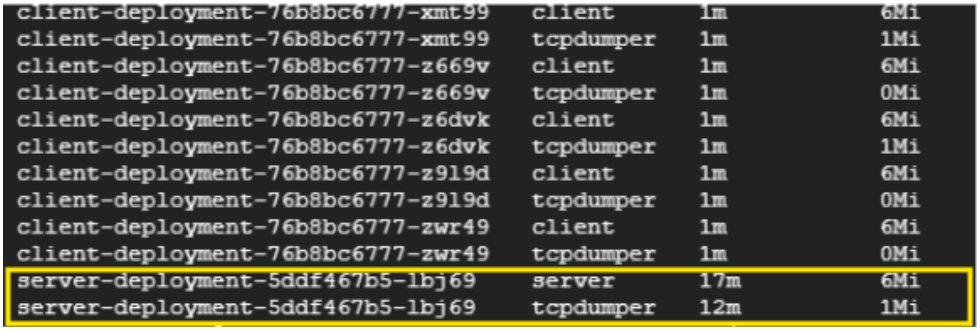


Figure 5.1: CPU and Memory Usages of Server

5.3.2 Node Performance Analysis Without Tcpdumper

The sole distinction between this experiment and the previous one is that the tcpdumper was excluded from the pods; only pods containing server or client containers were used. This setup was used to investigate how the node's performance was affected without tcpdumper. Based on previous findings, it's known that each client's tcpdumper has a CPU usage of 1m. Consequently, in this scenario where tcpdumpers were excluded, the increase in CPU usage with a growing number of pods was expected to be less compared to the previous scenario. Results are seen in Table 5.8.

As in the previous experiment, a rise in CPU usage was immediately noted after pod creation, deviating from the average. Performance data was only recorded after allowing the cluster to stabilize. The increase in usage persisted significantly longer when 100 pods were created. Additionally, many of the pods triggered **OutOfPods** warnings. This situation can be viewed in Figure 5.3. Also, by looking at Figure 5.3, it can be verified that all the pods were created in the same node.

In comparison to the previous experiment, a noticeable reduction in CPU usage was observed when the number of pods reached 25 and 50. Specifically, CPU usage decreased by 4% with 25 pods and by 9% with 50 pods. For the scenario with 2 pods,

POD	NAME	CPU (cores)	MEMORY (bytes)
client-deployment-76b8bc6777-2rmf7	client	1m	6Mi
client-deployment-76b8bc6777-2rmf7	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-2vmbs	client	1m	6Mi
client-deployment-76b8bc6777-2vmbs	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-42qxp	client	1m	6Mi
client-deployment-76b8bc6777-42qxp	tcpdumper	2m	0Mi
client-deployment-76b8bc6777-4hsbv	client	1m	6Mi
client-deployment-76b8bc6777-4hsbv	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-5t9tl	client	1m	6Mi
client-deployment-76b8bc6777-5t9tl	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-6kw7h	client	1m	6Mi
client-deployment-76b8bc6777-6kw7h	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-6rfkq	client	1m	6Mi
client-deployment-76b8bc6777-6rfkq	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-77ckc	client	1m	6Mi
client-deployment-76b8bc6777-77ckc	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-7cx7r	client	1m	6Mi
client-deployment-76b8bc6777-7cx7r	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-7th8c	client	1m	6Mi
client-deployment-76b8bc6777-7th8c	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-7v2zp	client	1m	6Mi
client-deployment-76b8bc6777-7v2zp	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-8mbl8	client	1m	6Mi
client-deployment-76b8bc6777-8mbl8	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-94rcn	client	1m	6Mi
client-deployment-76b8bc6777-94rcn	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-98wcv	client	1m	6Mi
client-deployment-76b8bc6777-98wcv	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-9p4tf	client	1m	6Mi
client-deployment-76b8bc6777-9p4tf	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-9wq8l	client	1m	6Mi
client-deployment-76b8bc6777-9wq8l	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-9zhhm	client	1m	6Mi
client-deployment-76b8bc6777-9zhhm	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-bd4db	client	1m	6Mi
client-deployment-76b8bc6777-bd4db	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-blkj8	client	1m	6Mi
client-deployment-76b8bc6777-blkj8	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-c4vdp	client	1m	6Mi
client-deployment-76b8bc6777-c4vdp	tcpdumper	1m	1Mi
client-deployment-76b8bc6777-c6mh7	client	1m	7Mi
client-deployment-76b8bc6777-c6mh7	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-cpsb6	client	1m	6Mi
client-deployment-76b8bc6777-cpsb6	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-cz95w	client	1m	6Mi
client-deployment-76b8bc6777-cz95w	tcpdumper	1m	0Mi
client-deployment-76b8bc6777-czzwb	client	1m	6Mi
client-deployment-76b8bc6777-czzwb	tcpdumper	1m	1Mi

Figure 5.2: CPU and Memory Usages of Clients

the CPU usage with tcpdumper was unexpectedly lower, which could be attributed to other programs running on the machine at the time. This discrepancy might also result from the fact that average values were used in the experimental results. No significant decrease in CPU usage was detected with 2, 4, or 10 pods. Based on these observations, while tcpdumper can be safely used with up to 10 clients, caution is advised as the number of pods increases. In such cases, adjusting tcpdumper's operating period and the number of packets it captures may improve its efficiency in the cluster.

Table 5.8: Node Performance Analysis Based on Pod Counts (without tcpdumper)

	Number of pods						
	Initial	2	4	10	25	50	100
CPU Core (m)	119	142	155	166	195	294	912
CPU (%)	12	14	16	17	20	30	97
Memory Bytes (Mi)	1047	1092	1174	1258	1440	1745	2329
Memory (%)	37	38	41	44	51	62	83

5.3.3 Single Server with Clients in Separate Nodes

In Experiment 5.3.1, both the server and client were running on the same machine. However, in this experiment, the server pod was hosted on *gke-cluster-1-default-pool-c8a57fb0-kt6k*, while the client pods were on *gke-cluster-1-default-pool-c8a57fb0-b8nf*. The nodes which shall be hosts of the pods can be seen in Figure 5.4.

A single server pod was used throughout the experiment, with the number of client pods scaled up based on the scenario. The number of pods shown in Table 5.9 represents the combined total of server and client pods.

Through this experiment, it was discovered that when the client and server are hosted on separate nodes, the server can handle client requests effectively, regardless of the increase in client numbers. The primary issue arises from the clients consuming excessive CPU resources on their respective machines as their numbers grow, leading to system crashes. This is also supported by the client node analysis, which showed results similar to those observed in Experiment 5.3.1, with CPU usage corresponding to the number of pods being consistently close to the average. To sum up, this experiment confirmed that no more than 25 pods should be created in the same node.

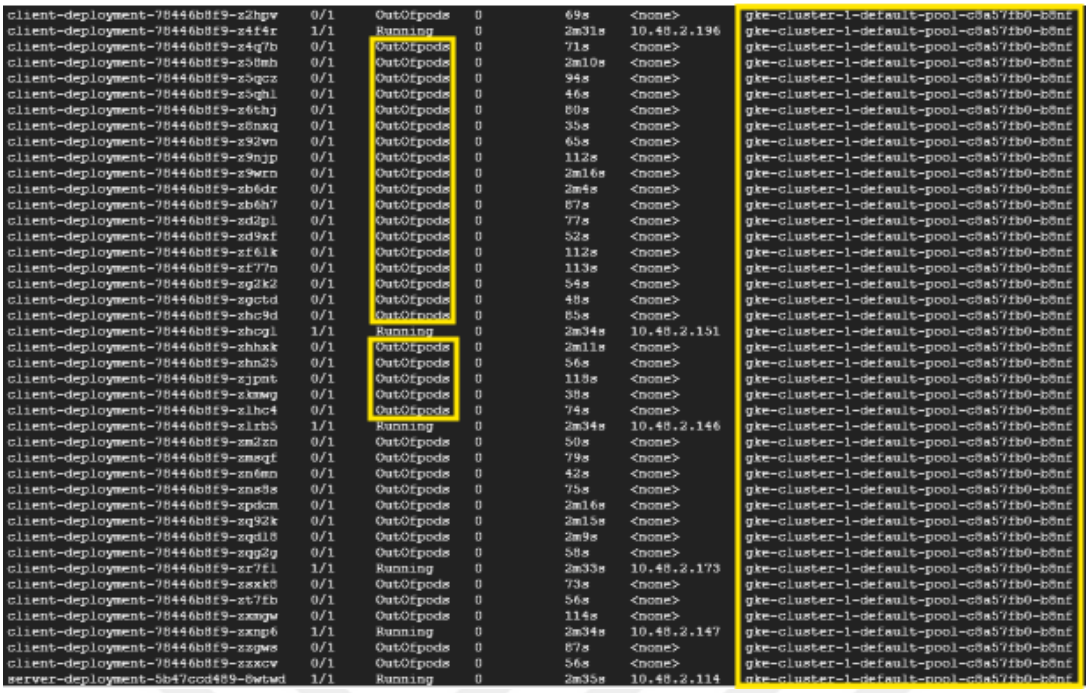


Figure 5.3: OutofPods fail case

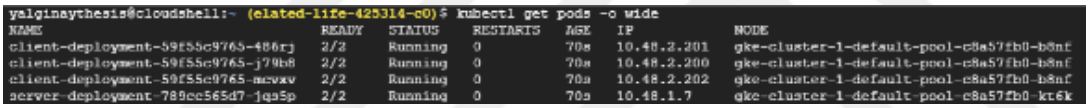


Figure 5.4: GKE Nodes

5.3.4 2-Server with Clients in Separate Nodes

In this experiment, unlike in Experiment 5.3.3, the scenario with 2 servers instead of 1 was evaluated. The goal was to assess whether having 2 servers reduced the load on the clients and led to any performance improvements on the node. The server pod was hosted on *gke-cluster-1-default-pool-c8a57fb0-kt6k*, while the client pods were on *gke-cluster-1-default-pool-c8a57fb0-b8nf*. Based on insights from the previous experiment, it was determined that node performance is directly related to the number of pods running on it. As the number of pods increased, CPU usage progressively rose, which ultimately hindered the proper functioning of the cluster. **OutOfPods** error was encountered again in the 100 client pods scenario. The recordings are shown in Table 5.10.

Table 5.9: Node Performance vs Pod Count (with tcpdumper, single server)

		Number of pods						
		Initial	2	4	10	25	50	100
Client Node	CPU Core (m)	128	138	155	186	239	333	933
	CPU%	13	14	16	19	25	35	99
	Memory Bytes (Mi)	1163	1180	1213	1286	1426	1774	2593
	Memory%	41	42	43	45	50	63	92
Server Node	CPU Core (m)	121	130	143	141	156	173	180
	CPU%	12	13	15	15	16	18	19
	Memory Bytes (Mi)	1393	1422	1430	1438	1440	1443	1451
	Memory%	49	50	50	51	51	51	51

Unlike the previous experiment, increasing the number of servers led to a decrease in the CPU usage of the client machine. In the previous experiment, the CPU usage was 25% with 24 clients, whereas in this experiment, the same usage was achieved with 32 clients. Similarly, the CPU usage observed with 49 clients in the previous experiment was reached in this one with 64 pods. In conclusion, carefully selecting the ratio between the number of servers and clients is crucial for determining the CPU usage of the nodes. Additionally, it was revealed that locating servers on different nodes gives better performance.

Table 5.10: Node Performance vs Pod Count (with tcpdumper, dual server)

		Number of pods							
		Initial	4	6	10	18	34	66	102
Client Node	CPU Core (m)	132	136	142	160	188	236	354	951
	CPU (%)	14	14	15	17	20	25	37	101
	Bytes (Mi)	1124	1154	1183	1202	1283	1498	1933	2322
	Memory (%)	40	41	42	43	45	53	68	82
Server Node	CPU Core (m)	132	137	143	145	151	156	180	207
	CPU (%)	14	14	15	15	16	16	19	22
	Bytes (Mi)	1426	1456	1462	1467	1459	1464	1463	1465
	Memory (%)	50	51	52	52	52	52	52	52

5.4 Impact of Communication Period on Response Delay

In this section, the impact of various communication periods between the server and client on response delay was observed. Additionally, the effect of changing the communication period was evaluated in environments both with and without the tcpdumper. They allowed for an examination of how the sidecar container influenced client request and response times.

5.4.1 With Tcpdumper

The cluster was configured with one server pod and one client pod, each augmented with tcpdumper containers as sidecars. In this scenario, the client generates a message of random size and transmits it to the server. The server then decodes the received message and returns an identical response to the client. The request and response times for both the client and the server were recorded using custom containers. This setup allowed for analyzing the message request and response times of the containers. The request and response times of the pods were recorded using the *kubectl logs pod* command. The initial 100 communication samples were collected and a response time analysis was conducted on these samples. Experiments were conducted with communication periods of 1, 2, 3, 5 and 10 seconds. Experiments were repeated 3 times and average values were calculated as shown in Table 5.11. The Figure 5.5 presents the log output from the client pod. The request time of clients was printed in the format of year-month-day hour:minute:second.milisecond.

Table 5.11 shows the message transmission times from the client to the server.

Table 5.11: Client Message Transmit Time (with tcpdumper)

	Communication Period (s)				
	1	2	3	5	10
Average Delay (ms)	1.475	1.341	1.408	1.490	1.308

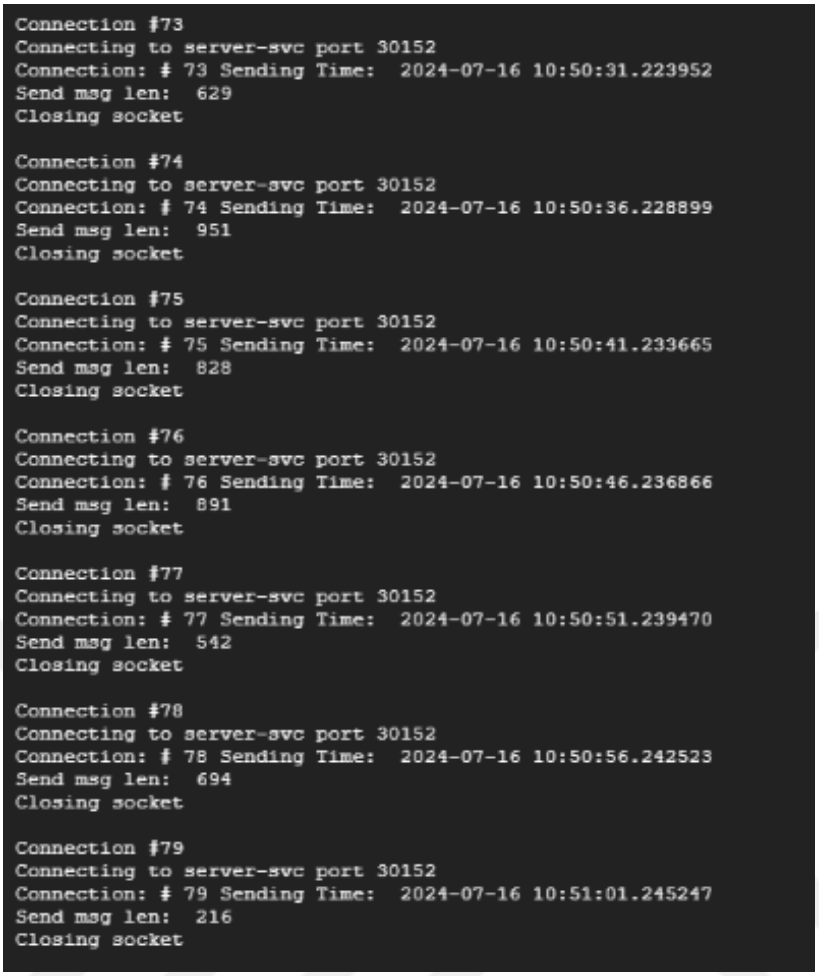


Figure 5.5: Client Pod Log

5.4.2 Without Tcpdumper

The sole variation from the preceding experiment is the exclusion of the tcpdumper container. This modification was implemented to investigate whether the presence of tcpdumper had any influence on the communication between the server and the client. Recordings are shown in Table 5.12.

Experimental results were assessed comprehensively. It was observed that, despite variations in the communication period, no significant delays in message transmission were detected. The observed variations were in the order of microseconds and can be considered negligible. Furthermore, the presence of a tcpdumper container within the pod did not adversely impact communication. Thus, it was proved that tcpdumper does not interfere with communication, its use for log collection in the current model

Table 5.12: Client Message Transmit Time (without tcpdumper)

	Communication Period (s)				
	1	2	3	5	10
Average Delay(ms)	1.511	1.241	1.360	1.637	1.175

is deemed appropriate and without negative effects.

5.5 Analysis of Health Monitor Module

This section presents a comprehensive analysis of the functionality of the HM module across various scenarios. The detection of attacker pods and network conditions, both before and following updates were thoroughly examined. Additionally, the duration required for network updates with different communication periods was evaluated. The initial version of the HM module was utilized in Experiments 5.5.1, 5.5.2 and 5.5.3, wherein HM performed regular scans of the cluster, accumulating extensive information on all pods within it. Then, a comprehensive analysis of these 3 experiments was made. Following this, several enhancements were implemented in the HM module. In this updated version, HM concentrated exclusively on the role labels of the pods rather than collecting exhaustive data on every pod in the cluster. When attackers and risky pods were identified, HM would only retrieve information pertinent to those specific pods. This improvement significantly enhanced the efficiency of network updates. The updated HM module was assessed in 5.5.5 and 5.5.6. At the end, all the experiments in this section were analyzed comprehensively to understand the impact of the updated HM.

5.5.1 HM - Initial Version with Tcpdumper

A network topology was established using 4 pods, each containing either a server or client container alongside a tcpdumper container. Figure 5.6 demonstrates the network topology.

According to the defined topology, pod1 and pod2 were employed as clients. They sent randomly generated messages to pod3 through service-3. Service-3 was configured to direct incoming messages to the pod with app label **pod3** and type label **main**. Upon receiving a message, pod3 responded by sending the message back to the corresponding client. The pod named pod3rep was excluded from the network and is intended to be integrated only upon the detection of an error. The pods within the network were labeled as seen in Table 5.13.

Following the deployment of the configuration files, the pods were assigned the IP addresses shown in Figure 5.7. The IP addresses for the services, endpoints and

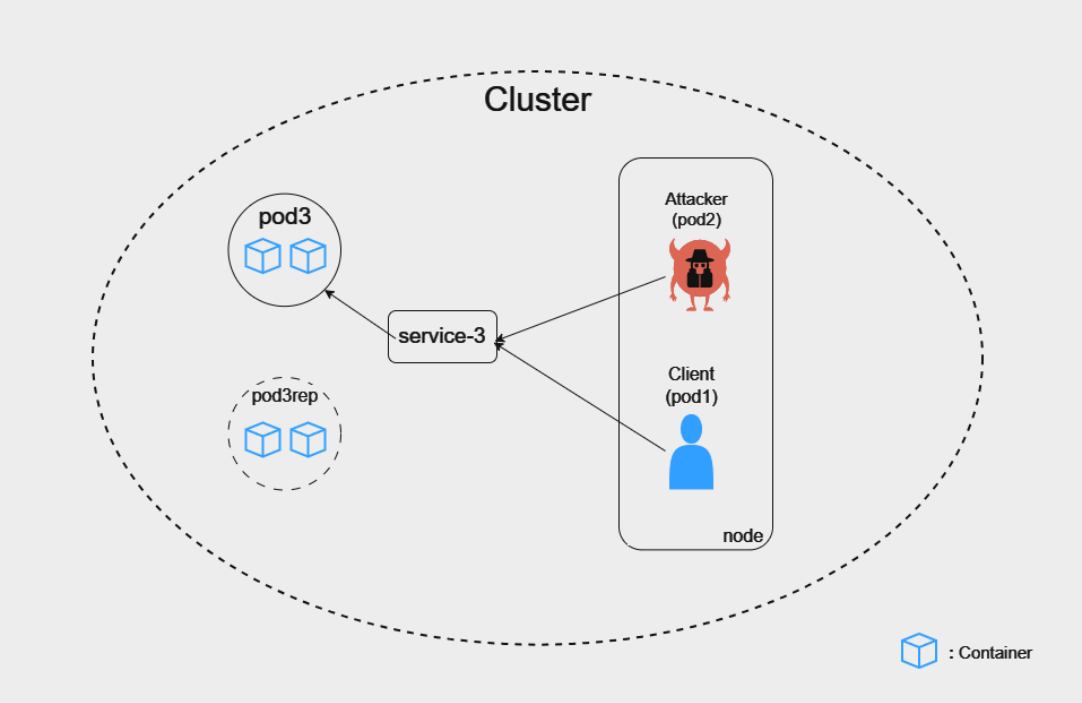


Figure 5.6: Initial Network Topology

nodes are also provided in the same figure. The IP addresses are essential for tracking the destination of the relevant packets when the HM module updates the network. By examining the endpoint of service-3, it becomes evident that the incoming requests to service-3 are routed to pod3.

```
yalginsythesis@cloudshell:~ (elated-life-425314-c0)$ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE                                              NOMINATED NODE   READ
pod1      2/2     Running   0          14s   10.88.1.37   gke-cluster-1-default-pool-003683cd-p9ng   <none>           <no
pod2      2/2     Running   0          14s   10.88.1.38   gke-cluster-1-default-pool-003683cd-p9ng   <none>           <no
pod3      2/2     Running   0          14s   10.88.1.39   gke-cluster-1-default-pool-003683cd-p9ng   <none>           <no
pod3rep   2/2     Running   0          14s   10.88.2.9    gke-cluster-1-default-pool-003683cd-q77s   <none>           <no
yalginsythesis@cloudshell:~ (elated-life-425314-c0)$ kubectl get svc -o wide
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE   SELECTOR
kubernetes   ClusterIP   34.118.224.1     <none>        443/TCP     136m  <none>
service-3    ClusterIP   34.118.226.203   <none>        30152/TCP   10m   app=pod3,type=main
yalginsythesis@cloudshell:~ (elated-life-425314-c0)$ kubectl get ep
NAME         ENDPOINTS           AGE
kubernetes   10.128.0.62:443     136m
service-3    10.88.1.39:30152    10m
yalginsythesis@cloudshell:~ (elated-life-425314-c0)$ kubectl get nodes -o wide
NAME                                              STATUS   ROLES    AGE   VERSION               INTERNAL-IP     EXTERNAL-IP
gke-cluster-1-default-pool-003683cd-jg6g   Ready    <none>   134m  v1.29.6-gke.1254000   10.128.0.63     34.69.222.208
gke-cluster-1-default-pool-003683cd-p9ng   Ready    <none>   134m  v1.29.6-gke.1254000   10.128.15.192   35.222.123.92
gke-cluster-1-default-pool-003683cd-q77s   Ready    <none>   134m  v1.29.6-gke.1254000   10.128.15.193   35.232.117.218
yalginsythesis@cloudshell:~ (elated-life-425314-c0)$
```

Figure 5.7: Cluster IP Addresses

The HM module identifies **attacker** and **risky** pods by examining their role labels. Once the pods labeled as attacker and risky are detected, the communication from the attacker pod to the risky pod is rerouted through the attacker pod's node IP instead

Table 5.13: Pod Labels Before Network Update

Pod Names	app label	role label	type label
pod1	pod1	client	main
pod2	pod2	client	main
pod3	pod3	server	main
pod3rep	pod3	server	backup

of the attacker pod IP. In this scenario, the risky pod is repurposed as a honeypot and its role label is updated as **honeypot**. Utilizing the pre-existing network policy, messages originating from the attacker pod are now strictly monitored; only those directed to the honeypot pod are permitted, while all other outbound communications from the attacker pod are blocked by the network policy. Additionally, the backup replica of the risky pod is integrated into the network by updating its type label to main. Service-3 recognizes the newly added pod (pod3rep) as the endpoint and redirects messages from non-attacker pods to this pod. Finally, the role label of the pod previously marked as attacker is changed to known-attacker and the risky pod is updated as known-risky. This approach allows the HM module to continue functioning without taking further action against previously detected attackers and risky pods.

In various scenarios, the communication period between the client and server was adjusted to 1, 2, 3, 5 and 10 seconds. Throughout all scenarios, the operational period of the HM module was consistently set to 1 second. A detailed analysis was conducted on the network updating time of the HM module. To simulate a network attack scenario, the role labels of pod2 and pod3 were modified. Pod2 was labeled as the **attacker** and pod3 as the **risky**. Figure 5.8 shows where the HM module detected the attacker pod. Attacker and risky pod IP addresses can be extracted from Figure 5.8 and matched with pod2 and pod3's IP in Figure 5.7. This allows us to verify that the attacker (pod2) and risky (pod3) pods were identified successfully.

After HM took action, the attacker pod was redirected to the honeypot. The network



Figure 5.8: HM Action

topology was reconfigured as depicted in Figure 5.9 and the labels of the pods were updated as shown in Table 5.14.

According to Table 5.14, the role and type labels of the pods were updated as planned. During the update, the correctness of the process was assessed by reviewing the logs of pod3 and pod3rep. The logs are shown in Figures 5.10 and 5.11.

Figure 5.10 illustrates that, following the network update, requests originally sent from pod2's IP address (10.88.1.38) to pod3 began to be sent from the IP address of the node hosting pod2 (10.128.15.192). No changes were observed in the requests from pod1's IP address (10.88.1.37) before and after the network update.

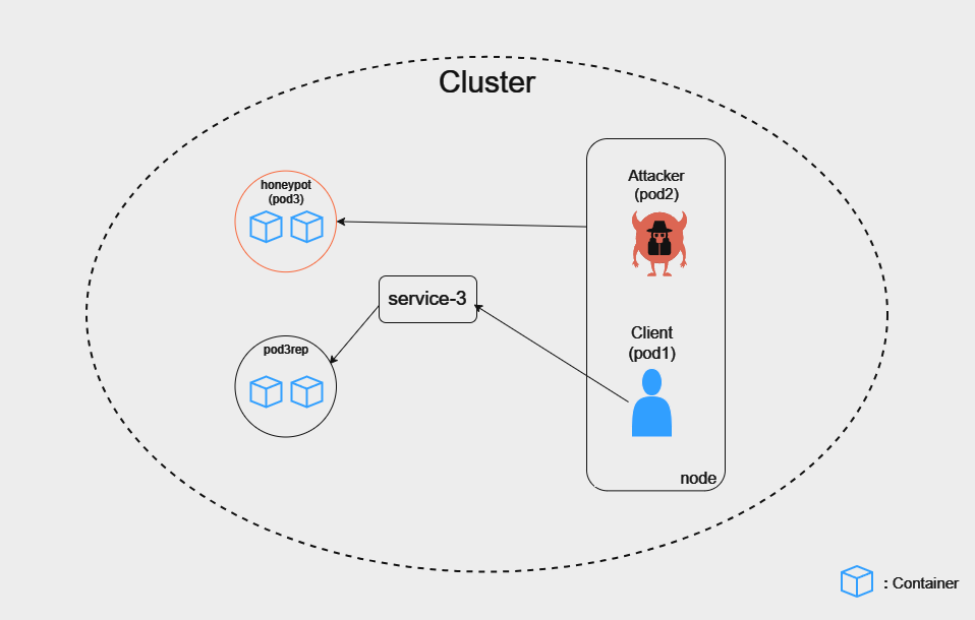


Figure 5.9: Updated Network Topology

In Figure 5.11, it is evident that requests from pod1's IP address (10.88.1.37) no longer reached pod3; instead, only requests from the node IP address of pod2 (10.128. 15.192) were observed. The requests from pod1 are now redirected to pod3rep. Figure 5.12 further clarifies the new routing path for requests originating from pod1.

The experimental results influenced by varying communication periods are presented in Table 5.15. Communication period indicates the period in which the client sends messages to the server. It sends 1 request per period. The first row records the timestamp of the last message sent from pod2's IP address to pod3. The second row contains the timestamp of the first message sent to pod3 from the IP address of the node hosting pod2. The third row is the calculation of the time difference between these two events, documenting the duration required for updating to the attacker pod communication. In the fourth row, the timestamp of the last message from pod1's IP address to pod3 is recorded. The fifth row notes the timestamp of the first message received by pod3rep from pod1. Finally, the sixth row records the time taken for the update to be processed on the risky pod.

Examining Table 5.15, during the update, the communication period continued to work as before the update. There was no noticeable latency during the update. This

Table 5.14: Pod Labels After Network Update

Pod Names	app label	role label	type label
pod1	pod1	client	main
pod2	pod2	known-attacker-pod	main
pod3	pod3	known-risky-pod	honeypot
pod3rep	pod3	server	main

client continued its life without experiencing any latency at the time of the update. This outcome demonstrates that the HM module performed its function as expected.

Table 5.15: Network Update Analysis Based on Communication Period (with tcp-dumper)

	Communication Period (s)				
	1	2	3	5	10
Pod2 to Pod3 t(s)	26.708	47.686	14.262	15.542	19.628
Node to Pod3 t(s)	27.711	49.688	17.265	20.545	29.632
Pod2 Update t(s)	1.004	2.003	3.004	5.003	10.004
Pod1 to Pod3 t(s)	35.597	57.507	23.903	26.262	29.311
Pod1 to Pod3rep t(s)	36.601	59.511	26.907	31.267	39.316
Pod1 Update t(s)	1.004	2.004	3.004	5.004	10.005

```
Waiting for a connection
From:  ('10.88.1.37', 40326)
Connection: # 181 Receive Time:  2024-08-06 09:26:21.182982
rcv msg len:  447
No data from ('10.88.1.37', 40326)

Waiting for a connection
From:  ('10.88.1.38', 45348)
Connection: # 182 Receive Time:  2024-08-06 09:26:21.246993
rcv msg len:  918
No data from ('10.88.1.38', 45348)

Waiting for a connection
From:  ('10.88.1.37', 40332)
Connection: # 183 Receive Time:  2024-08-06 09:26:22.190780
rcv msg len:  810
No data from ('10.88.1.37', 40332)

Waiting for a connection
From:  ('10.88.1.38', 45350)
Connection: # 184 Receive Time:  2024-08-06 09:26:22.250981
rcv msg len:  646
No data from ('10.88.1.38', 45350)

Waiting for a connection
From:  ('10.88.1.37', 40032)
Connection: # 185 Receive Time:  2024-08-06 09:26:23.194580
rcv msg len:  780
No data from ('10.88.1.37', 40032)

Waiting for a connection
From:  ('10.128.15.192', 60284)
Connection: # 186 Receive Time:  2024-08-06 09:26:23.254237
rcv msg len:  407
No data from ('10.128.15.192', 60284)

Waiting for a connection
From:  ('10.88.1.37', 40046)
Connection: # 187 Receive Time:  2024-08-06 09:26:24.197618
rcv msg len:  953
No data from ('10.88.1.37', 40046)

Waiting for a connection
From:  ('10.128.15.192', 60296)
Connection: # 188 Receive Time:  2024-08-06 09:26:24.257050
rcv msg len:  89
No data from ('10.128.15.192', 60296)
```

Figure 5.10: Pod3 Log (1)

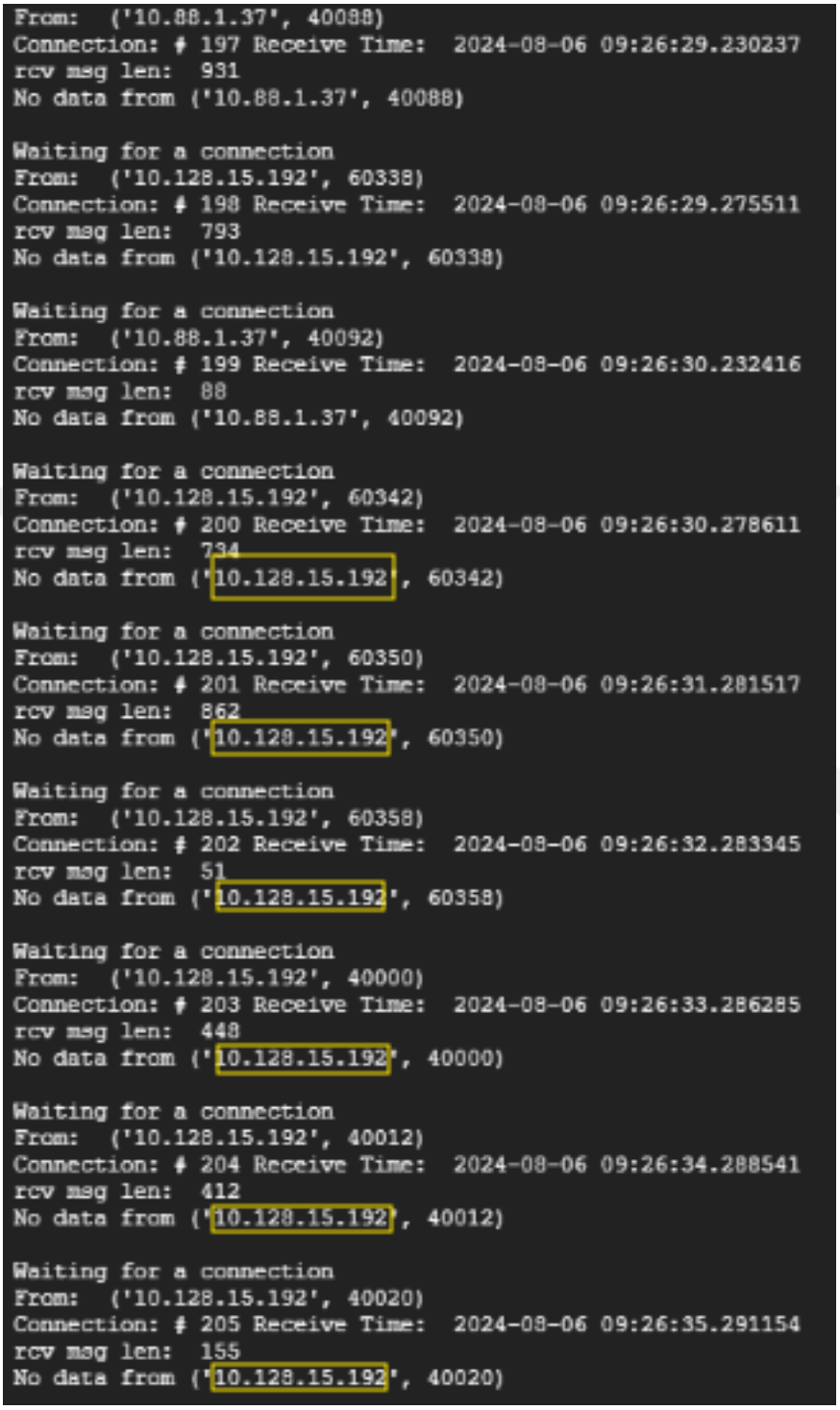


Figure 5.11: Pod3 Log (2)

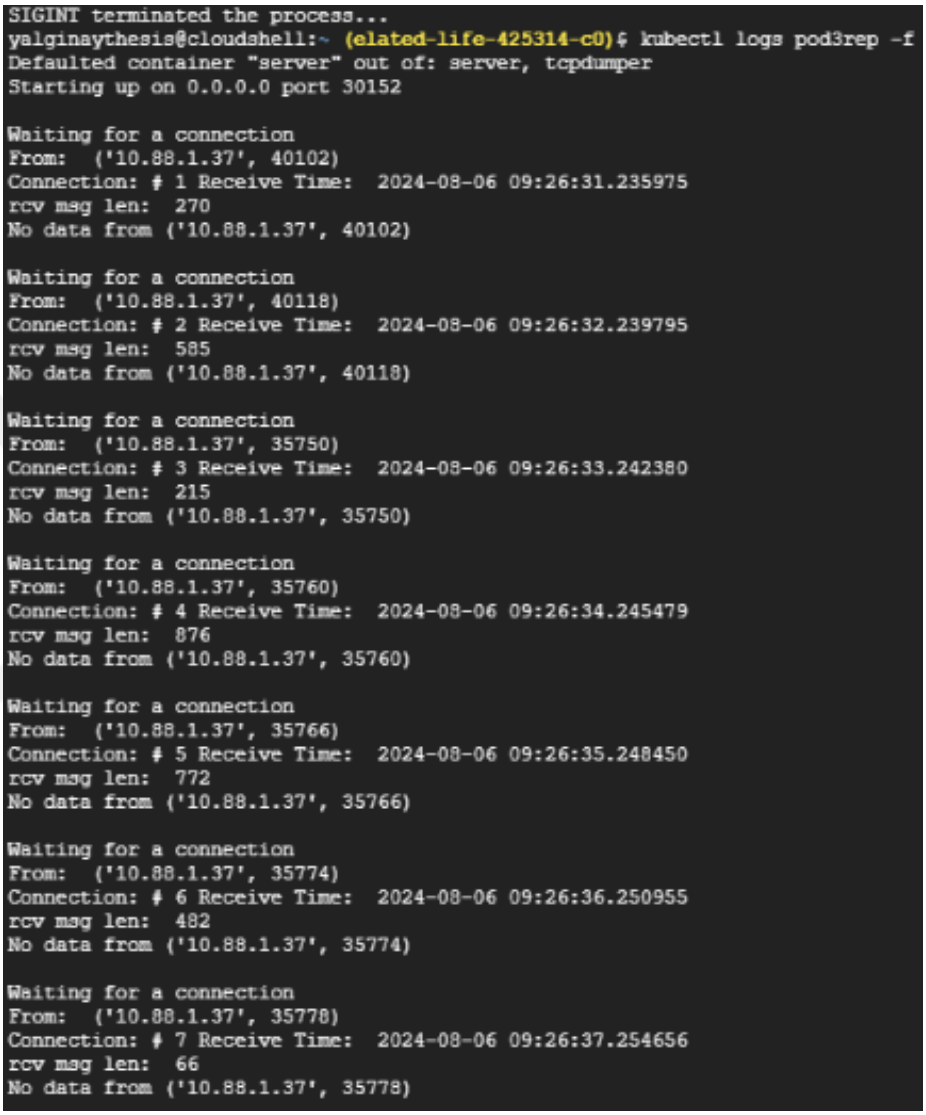


Figure 5.12: Pod3rep Log

5.5.2 HM - Initial Version without Tcpdumper

The primary distinction between this experiment and the experiment in 5.5.1 lies in the exclusion of the tcpdumper container from the pods. All other parameters remained consistent. This experiment aimed to analyze any potential changes in the communication between the server and the client, as well as any variations in the network update time in the absence of the tcpdumper container. The experiments results are given in Table 5.16. Communication period indicates the period in which the client sends messages to the server. It sends 1 request per period.

Table 5.16: Network Update Analysis Based on Communication Period (without tcpdumper)

	Communication Period (s)				
	1	2	3	5	10
Pod2 to Pod3 t(s)	24.869	45.669	05.762	09.791	30.889
Node to Pod3 t(s)	25.872	47.673	08.765	14.793	40.893
Pod2 Update t(s)	1.003	2.004	3.003	5.002	10.005
Pod1 to Pod3 t(s)	33.993	55.319	14.074	19.539	41.377
Pod1 to Pod3rep t(s)	34.996	57.323	17.078	24.543	51.380
Pod1 Update t(s)	1.004	2.004	3.004	5.004	10.003

The results are consistent with experiment 5.5.1, confirming the constant client response duration. This result indicates that the use of tcpdumper did not lead to any latency before and after the network update.

5.5.3 HM Initial Version with Nginx and Tcpdumper

In this experiment, the default nginx container was utilized in place of the Python based custom server and client containers employed in prior experiments. Addition-

ally, a tcpdumper container was incorporated as a sidecar alongside the default nginx container. To facilitate communication between the server and client, a bash script was used to insert a code block into the pod configuration files, thereby establishing the communication. The results of these tests are presented in Table 5.17. Communication period indicates the period in which the client sends messages to the server. It sends 1 request per period.

Table 5.17: Network Update Analysis Based on Com. Period (with nginx, tcpdumper)

	Communication Period (s)				
	1	2	3	5	10
Pod2 to Pod3 t(s)	25	46	37	33	39
Node to Pod3 t(s)	26	48	40	38	49
Pod2 Update t(s)	1	2	3	5	10
Pod1 to Pod3 t(s)	34	56	48	46	49
Pod1 to Pod3rep t(s)	35	58	51	51	59
Pod1 Update t(s)	1	2	3	5	10

Irrespective of the container within the pod, the HM module consistently updated the network within comparable time frames. The time resolution in these test results is recorded in seconds because the default nginx container generates output logs at this level of precision.

5.5.4 Comprehensive Analysis of HM's Initial Version

The total network update duration was assessed using the time data recorded during the previous experiments. The analysis focused on the time difference between the last message sent from pod1 to pod3 and the first message received by pod3rep. This interval encompasses the complete update times for both the attacker and risky pods. Additionally, how many cycles the network was updated was also noted. This value

was obtained by dividing the network's update time by the relevant communication period. The results are provided in Table 5.18.

Table 5.18: Network Update Analysis Based on Communication Period and Container Type

		Communication Period T(s)				
		1	2	3	5	10
– sv+cl+tcp –	Total Update Time(s)	9.89	11.83	12.65	15.72	19.69
	Update Cycle Count	~9.9	~5.9	~4.2	~3.1	~2.0
— sv+cl —	Total Update Time(s)	10.13	11.65	11.32	14.75	20.49
	Update Cycle Count	~10.1	~5.8	~3.8	~3.0	~2.0
–nginx+tcp–	Total Update Time(s)	10	12	14	18	20
	Update Cycle Count	~10	~6	~4.7	~3.1	~2

"sv" represents custom server, "cl" custom client, "tcp" tcpdumper and "nginx" default nginx containers expressed in Table 5.18. It indicates which containers were used in the relevant experiment. Examination of the table reveals that the total update times and cycles of the network yielded highly comparable results. This observation led to the conclusion that the developed HM module is independent of the specific containers used, as updates occur within similar time-frames regardless of container types.

According to the results, it was detected that the network is not updated fast enough, particularly during more frequent communication. For instance, with a communication period of 1 second, the network was updated approximately in 10 cycles. Following the detection of an attacker, the system continued to function incorrectly for

about 9 additional cycles, indicating that the system did not recover promptly. Consequently, performance improvements were deemed necessary for the HM module. Instead of continuously monitoring the redundant information of all pods, only the role labels were monitored. This modification was expected to enhance the performance of the HM module, particularly in clusters with a large number of pods. After this update, Experiments 5.4 and 5.5 were performed to see the changes. Based on previous experiments, the experiment was not repeated for the scenario without tcpdumper, as it is known that the presence or absence of tcpdumper would not change the experimental results.

5.5.5 HM - Updated Version with Tcpdumper

In this section, the analyses conducted in experiment 5.5.1 regarding the duration required to update the network with various containers were re-examined using the updated version of the HM module. It was anticipated that the updated HM module would lead to an enhancement in the overall update time. Furthermore, in contrast to previous experiments, the HM process period was extended from 1 second to 2 seconds. This modification was expected to enhance the CPU efficiency of the HM module. Figure 5.13 shows the initial status of the network. The relevant pod IP's were used while updating the network.

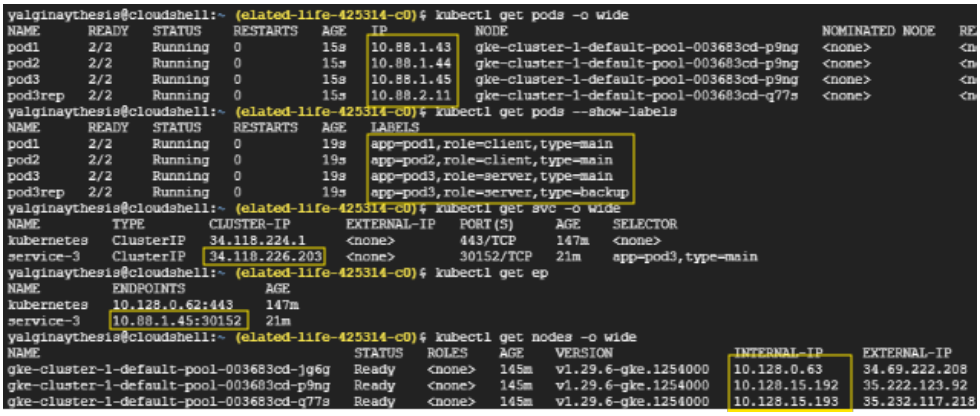


Figure 5.13: Cluster IP addresses

Consistent with previous experiments, pod2 and pod3 were designated as **attacker** and **risky** pods, respectively. The scenario in which the HM module detects the at-

tacker and risky pods is illustrated in Figure 5.14.



Figure 5.14: Updated HM Action

Examination of Figure 5.14 reveals that the HM module successfully identified pod2 and pod3 with the corresponding roles.

Referencing Figure 5.15, it was observed that requests from pod2 (10.88.1.44) to pod3 are routed through pod2's node IP (10.128.15.192) after a certain period. Initially, requests to pod3 from pod1 (10.88.1.43) continued uninterrupted. However, upon detection of the risky pod, requests originating from pod1 were redirected to pod3rep, as illustrated in Figure 5.16.

The experimental findings, affected by different communication periods are detailed in Table 5.19. Communication period indicates the period in which the client sends

messages to the server. It sends 1 request per period. The first row logs the timestamp of the final message transmitted from pod2's IP address to pod3. The second row records the timestamp of the initial message sent to pod3 from the IP address of the node hosting pod2. The third row computes the time difference between these two events, indicating the duration required to update communication with the attacker pod. In the fourth row, the timestamp of the last message from pod1's IP address to pod3 is documented. The fifth row captures the timestamp of the first message received by pod3 from pod1. Lastly, the sixth row measures the time taken for the update to be applied to the compromised pod.

Table 5.19: Network Update Analysis on Pod Communication (with tcpdumper)

	Communication Period (s)				
	1	2	3	5	10
Pod2 to Pod3 t(s)	21.946	26.046	9.835	02.038	15.952
Node to Pod3 t(s)	22.948	28.049	12.839	07.043	25.958
Pod2 Update t(s)	1.003	2.004	3.004	5.005	10.006
Pod1 to Pod3 t(s)	24.580	28.677	12.785	02.784	15.826
Pod1 to Pod3rep t(s)	25.585	30.682	15.790	07.788	25.831
Pod1 Update t(s)	1.004	2.005	3.005	5.003	10.005

Examining Table 5.19, during the update, the communication period continued to work as before the update. There was no noticeable latency during the update. The client continued its life without experiencing any latency at the time of the update. This outcome demonstrates that the HM module performed its function as expected.

5.5.6 HM - Updated Version with Nginx and Tcpdumper

In this experiment, the performance analysis of the updated HM module was performed in the network topology with default nginx and tcpdumper containers. The

action taken by HM when it detects attackers and risky pods can be seen in Figure 5.17.

As illustrated in Figure 5.18, it was observed that requests from pod2 (10.88.1.68) to pod3 are routed through the IP address of pod2's node (10.128.15.192) after a certain period. Initially, requests from pod1 (10.88.1.67) to pod3 proceeded without interruption. However, once the attacker pod was identified, requests originating from pod1 were redirected to pod3rep, as depicted in Figure 5.19.

Table 5.20: Network Update Analysis Based on Com. Period (with nginx)

	Communication Period (s)				
	1	2	3	5	10
Pod2 to Pod3 t(s)	55	13	30	25	7
Node to Pod3 t(s)	56	15	33	30	17
Pod2 Update t(s)	1	2	3	5	10
Pod1 to Pod3 t(s)	58	15	34	02	9
Pod1 to Pod3rep t(s)	59	17	37	33	19
Pod1 Update t(s)	1	2	3	5	10

Table 5.20 presents an analysis of whether delays occurred in the communication between the server and the client during the network update. Communication period indicates the period in which the client sends messages to the server. It sends 1 request per period. Consistent with previous experiments, no delays were detected. The disparity in the initiation times of communication between pod2 and pod1 with the server resulted in updates not occurring at precise multiples of the communication period. For instance, in a scenario where the communication period is 10 seconds, this discrepancy explains why messages received at the 7^{th} and 9^{th} seconds did not arrive simultaneously.

5.5.7 Comprehensive Analysis of Initial and Updated Version of HM

This section presents a comprehensive analysis of experiments conducted with varying communication periods and different container configurations for the HM. Table 5.21 presents the total update time and the number of update cycles for both the old and updated versions of the HM module.

"sv" represents custom server, "cl" custom client, "tcp" tcpdumper and "nginx" default nginx containers expressed in Table 5.21. It indicates which containers were used in the relevant experiment. In the first three container scenarios presented in the table, the previous version of the HM module was employed whereas the updated HM module was utilized in the fourth and fifth scenarios. Analyzing the cycle count parameters reveals that when communication periods are 5 and 10 seconds, the network can be updated with updated HM within the same cycle almost immediately after detecting the attacker pod. In scenarios with more frequent communication periods, the update cycles have significantly improved, even if the network cannot be updated within the same cycle. Specifically, in the case of a 1-second communication period, the network, which required 10 cycles to update with the old HM module, could be updated in 4 cycles with the updated HM module. As illustrated in Table 5.21, scenarios with 2 and 3-second communication periods demonstrate a clear twofold improvement in cycle count efficiency. In scenarios where the default nginx container was employed, pod1 and pod2 did not initiate communication simultaneously due to the manual preparation of the code responsible for sending the requests. This discrepancy also accounts for the fractional values observed in the cycle counts.

As a result, the HM module provided a deterministic behavior and updated the network independently of the container in the pod. In addition, in scenarios with a high communication period, it updated the network within the next cycle, while in cases with a low communication period, it was able to update the network after a few communication cycles.

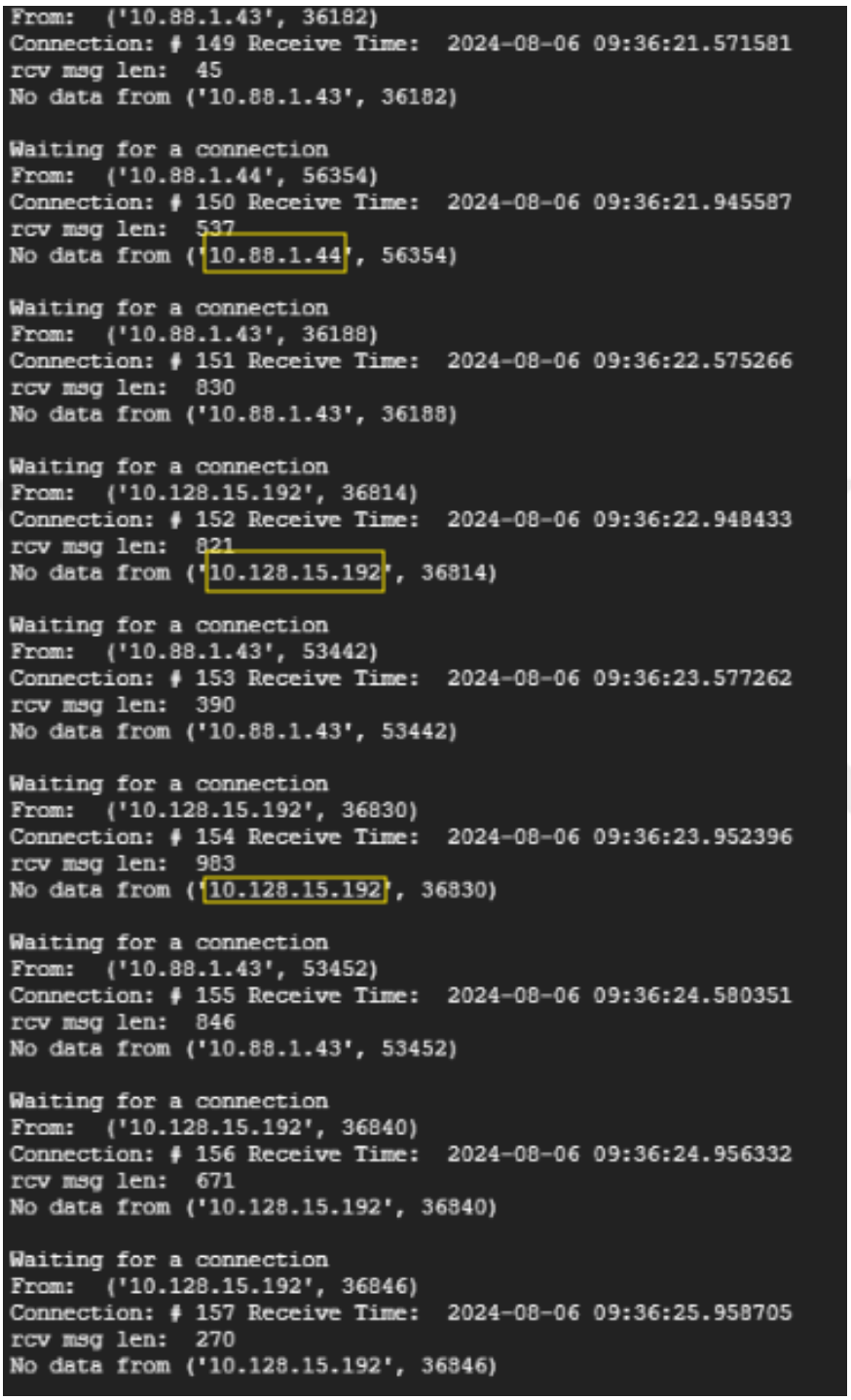


Figure 5.15: Pod3 Log

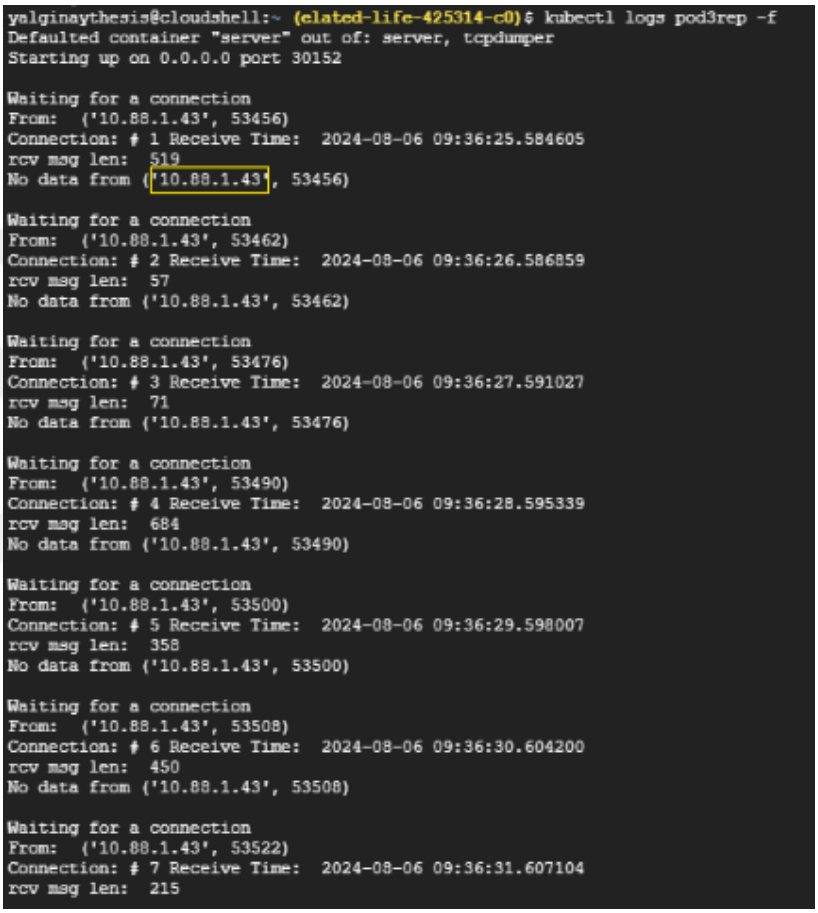


Figure 5.16: Pod3rep Log

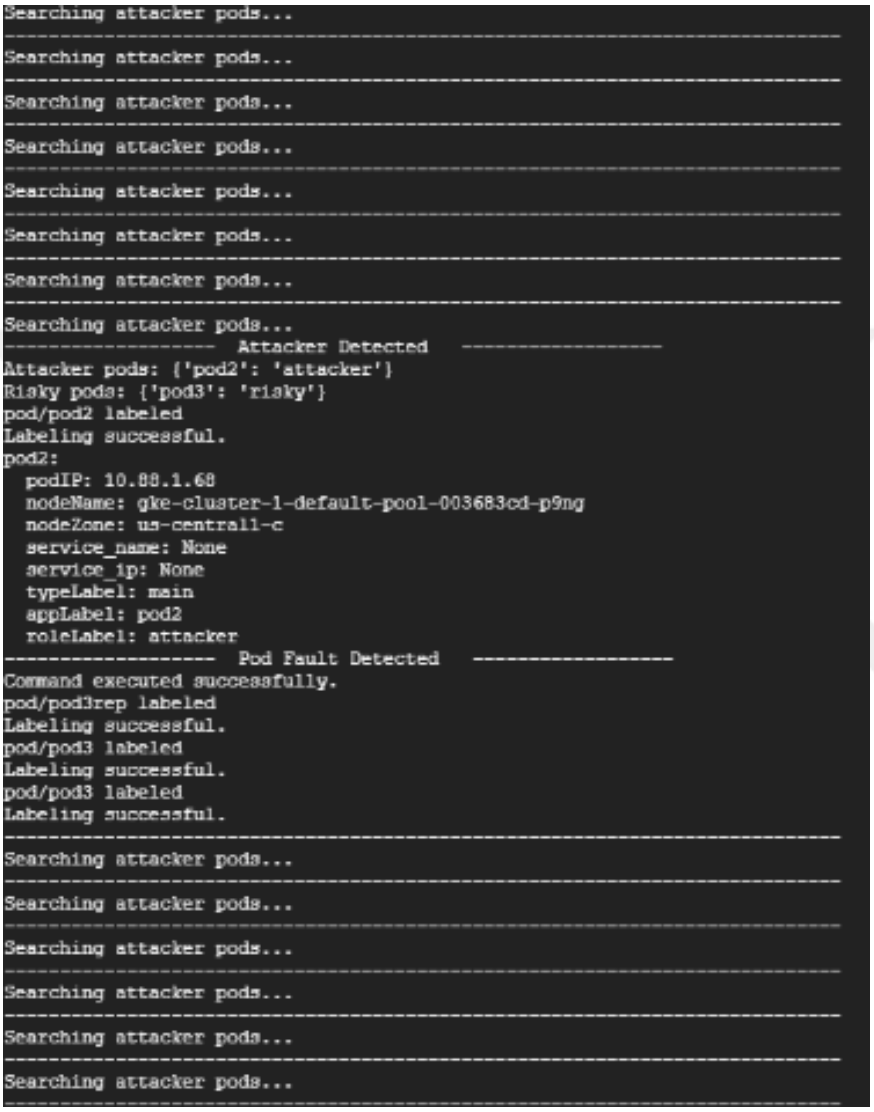


Figure 5.17: Updated HM Action

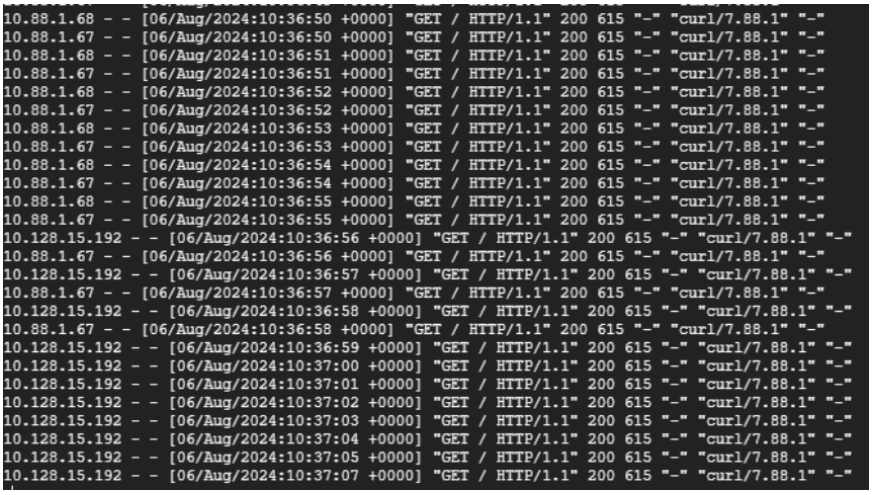


Figure 5.18: Pod3 Log

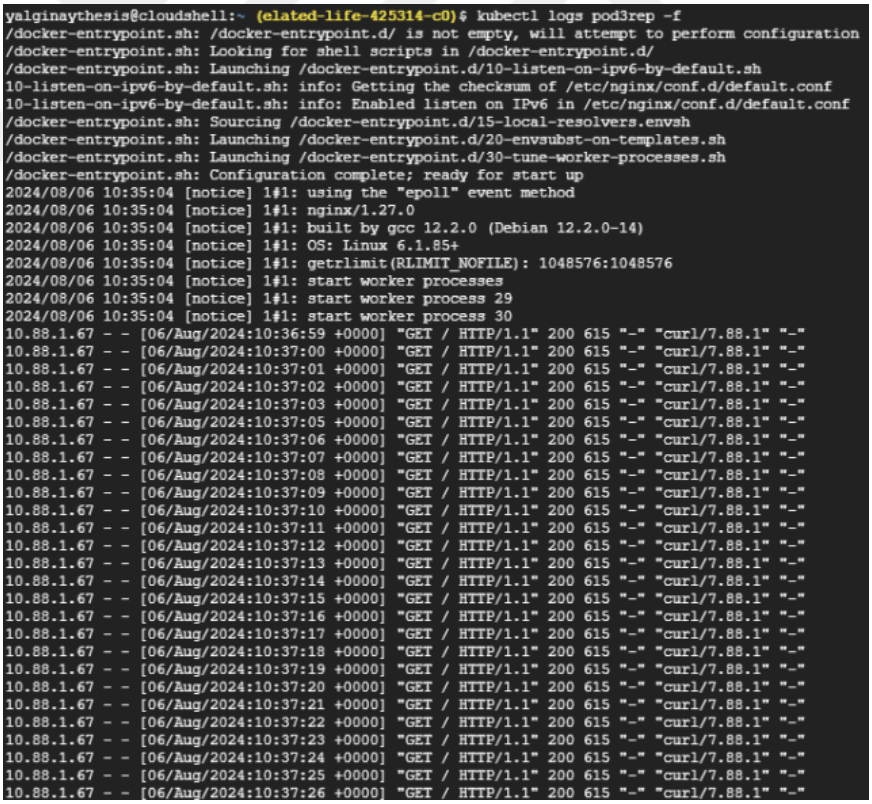


Figure 5.19: Pod3rep Log

Table 5.21: Comprehensive Network Update Analysis

		Communication Period T(s)				
		1	2	3	5	10
– sv+cl+tcp –	Total Update Time(s)	9.89	11.83	12.65	15.72	19.69
	Update Cycle Count	~9.9	~5.9	~4.2	~3.1	~2.0
—— sv+cl ——	Total Update Time(s)	10.13	11.65	11.32	14.75	20.49
	Update Cycle Count	~10.1	~5.8	~3.8	~3.0	~2.0
–nginx+tcp–	Total Update Time(s)	10	12	14	18	20
	Update Cycle Count	~10	~6	~4.7	~3.1	~2
–nginx+tcp– —— sv+cl+tcp ——	Total Update Time(s)	3.639	4.637	5.955	5.75	9.880
	Update Cycle Count	~3.6	~2.3	~1.99	~1.15	~0.9
	Total Update Time(s)	4	4	7	8	12
	Update Cycle Count	~4	~2	~2.33	~1.6	~1.2

5.6 Serial 3-Pod Topology

A comprehensive analysis of the HM module was performed in Experiments 5.5. Subsequently, the module's performance was assessed across different network topologies. Initially, a serial 3-Pod network topology, employing a microservice architecture was established as illustrated in Figure 5.20. Pre-configured custom containers were deployed within the pods of this topology. The microservice architecture was implemented by interconnecting these containers through services. In addition, backup pods were created but were isolated from communication through the implementation of network policies. The backup pods were maintained in a standby state, ready to replace the risky pod upon detection of any risky pod within the system. To provide the communication between pods, HTTP POST requests were utilized. The logs captured the timestamps of messages received and sent by the Python-based containers. The microservices in this scenario have the simple task of adding up numbers. Each pod in the network is employed to increment the number in the incoming request with the value contained in its name.

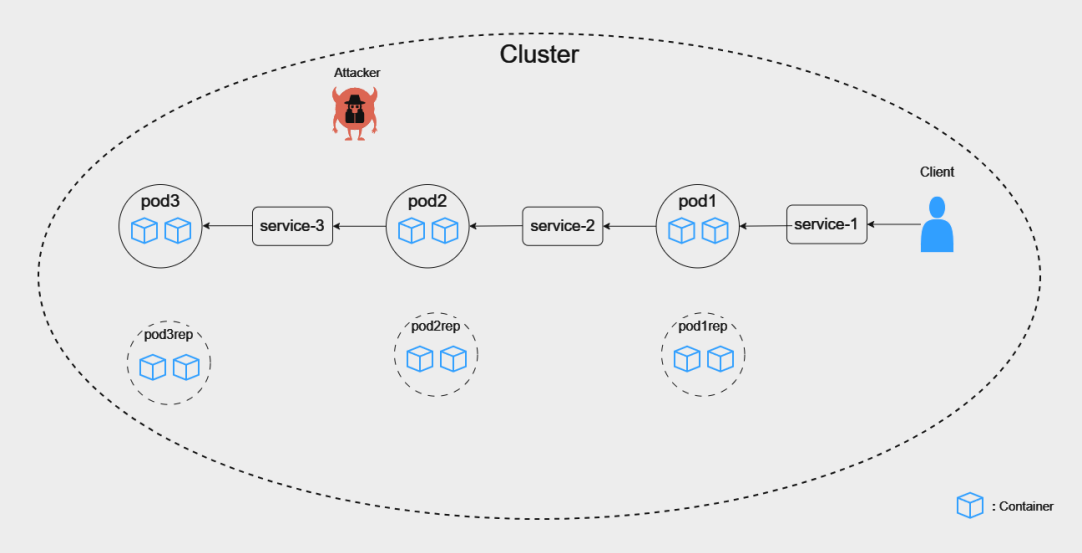


Figure 5.20: Serial 3-Pod Topology

Regarding the operational principle of the topology, the value sent by the client first arrives at pod1. In pod1, the received number is incremented by 1 and then forwarded to pod2 where it is incremented by 2 and subsequently sent to pod3. The value processed by pod3 is incremented by 3 and then passed back through pod2 and pod1,

ultimately reaching the client. To illustrate the scenario with an example, the client sends the value 1 to pod1. In pod1, the value is incremented by 1 to become 2 which is then sent to pod2. In pod2, the value is further incremented by 2 resulting in 4 and is sent to pod3. Pod3 adds 3 to the value, yielding 7, which then responds back to pod2 and pod1 respectively before returning to the client. Thus, in response to the client's initial input of 1, a final value of 7 is returned. To maintain continuous communication, requests are sent periodically using a bash script located in the client pod. Additionally, the client's request and response times are recorded in the client pod's log.

Following the establishment of the topology, an attacker pod was introduced to simulate an adversarial scenario. This attacker pod initiated a DoS attack by generating an excessive number of requests within a given time frame, targeting the designated pod. Subsequent stages of the experiment involved analyzing scenarios in which the attacker pod sequentially targeted pod3, pod2 and pod1. Figure 5.21 illustrates the IP addresses of the pods within the cluster, in addition to the IP addresses of the node and services.

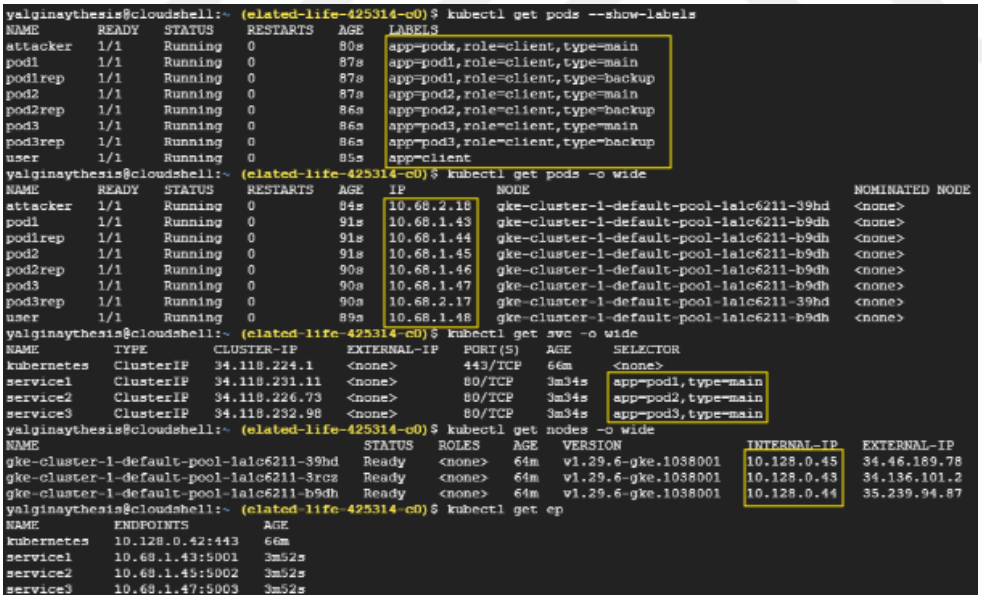


Figure 5.21: Cluster IP Addresses

The attacker pod was identifiable by the name **attacker**. As the app label was not relevant, it was assigned the placeholder **podx**. Additionally, the attacker's IP address

was recorded as 10.68.2.18. The client pod, along with the associated username, was presented in Figure 5.21. The client IP address was seen as 10.68.1.48. The figure also displayed the nodes hosting the pods along with their corresponding IP addresses.

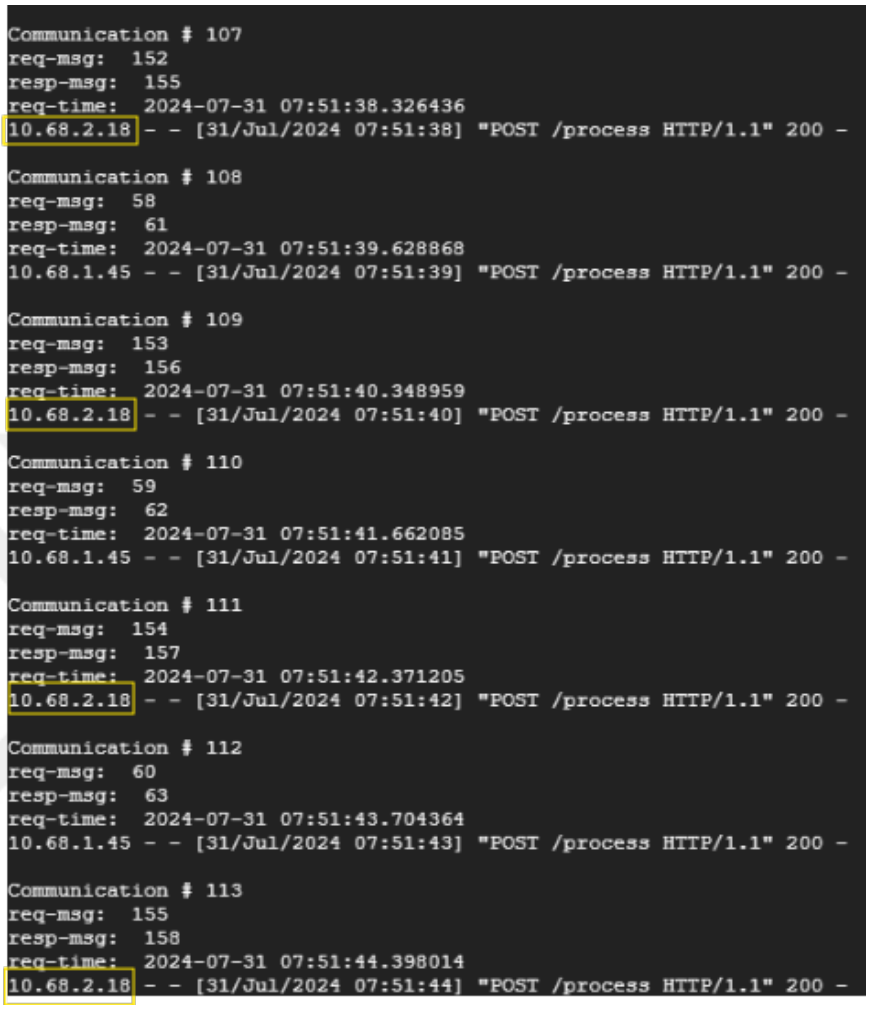


Figure 5.22: Pod3 Log with Attacker Pod

Figure 5.22 demonstrates that pod3 consistently received requests from both pod2 and the attacker's IP addresses, as intended. To distinguish the incoming request values, the attacker initiated messages starting at 100, while the client began at 1.

The attacker pod was assigned a role label of the attacker, while pod3 was labeled as risky, thereby simulating a scenario in which an error could be identified. In this context, it was confirmed that HM successfully detected the attacker and risky as illustrated in Figure 5.23. It is important to note that since this experiment was conducted before the HM module update, the HM utilized was an old version. Thus, it

should be considered that the old HM process periodicity is 1 second.



Figure 5.23: HM Action

It is known from Figure 5.21 that the IP address 10.68.1.45 belongs to pod2. It is confirmed from Figure 5.24 that requests from pod2 now go to pod3rep instead of pod3. This confirms that HM updated the network successfully.

To analyze the network's behavior before the detection of the attacker pod, 50 communication samples were collected in a scenario where the network configuration remained unchanged. The average response times of the client, pod1 and pod2 were measured based on these samples. The recorded values are presented in Table 5.22.

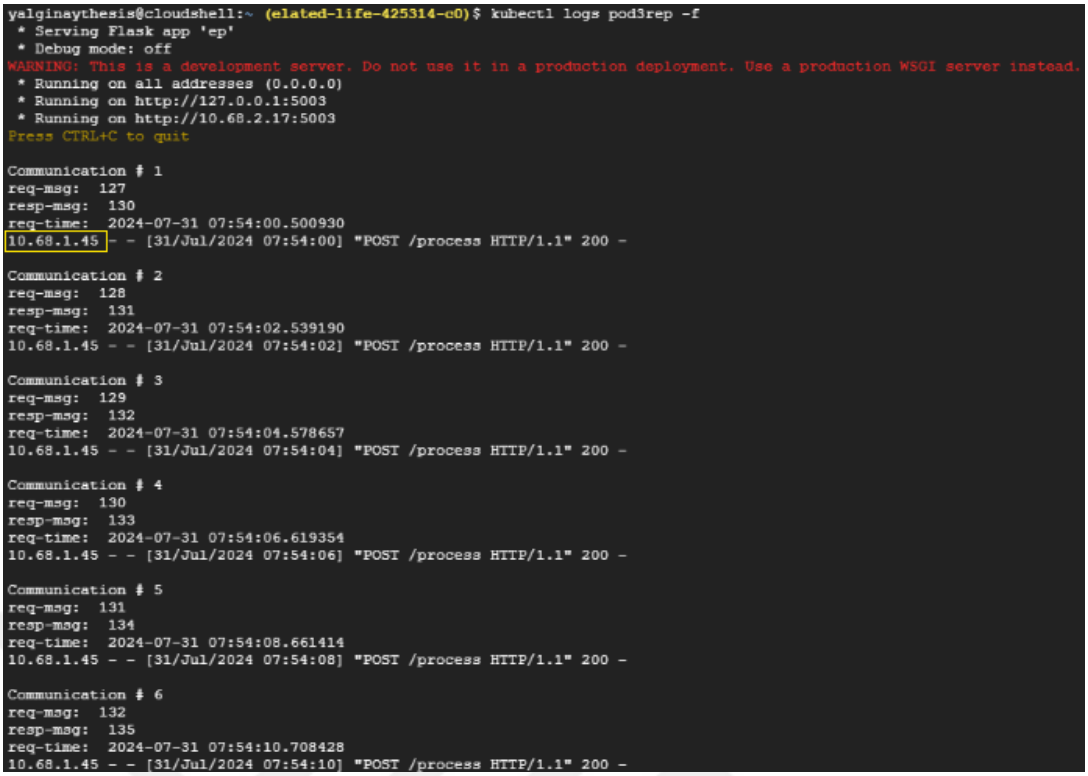


Figure 5.24: Pod3rep Log

Table 5.22: Average Response Times Before Network Update

Client Response Duration (ms)	Pod1 Response Duration (ms)	Pod2 Response Duration (ms)
45.46	39.891	9.731

The analysis focused on how the response time of the client pod was affected during network updates. To achieve this, the timestamp of the last message received by pod3 from pod2 and the first message received by pod3rep from pod2 were recorded. Additionally, the log of the client pod was examined to capture the time values for these two events, as well as two additional timestamps immediately before and after them. Based on the recording Table 5.23 was filled.

Then, to investigate the response time analysis of client, pod1 and pod2 Table 5.24 was created. The column with the value of communication sample 3 corresponds to the time of the last message that pod3 received from pod2, while the time of the first message that pod3rep received from pod2 corresponds to communication sample 4.

Table 5.23: Client Pod Timestamps During Network Update - Pod3 Fail Case

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)
1	07:53:54.309	07:53:54.343
2	07:53:56.350	07:53:56.437
3	07:53:58.443	07:53:58.472
4	07:54:00.478	07:54:00.511
5	07:54:02.516	07:54:02.548
6	07:54:04.554	07:54:04.592

This analysis aimed to determine whether there was a variation in the response times of the client pod, pod1 and pod2 during the network update and to assess any changes in response times immediately before and after the update. It should be noted that in this scenario tcpdumper containers were not added to the pods.

At first glance at Table 5.24, it is naturally noticed that the response durations decrease when the client pod1 and pod2 are viewed in order, as the client waits for all pods to send a response. In addition, no increase in delay is observed while the network is updated.

Table 5.24: Pod Response Times During Network Update - Pod3 Fail Case

	Communication Sample Number					
	1	2	3	4	5	Average
Client Response Duration (ms)	34	87	29	33	32	38
Pod1 Response Duration (ms)	14.101	18.253	12.243	15.555	13.286	15.560
Pod2 Response Duration (ms)	7.376	7.973	5.340	8.888	6.502	7.459

To enhance the statistical reliability of the results, the client response time analysis in cases where errors were detected was conducted five times. The response time was measured by examining the last message received before the network update and the first message received afterward. The arithmetic average of these two values is available in Table 5.25. The final column contains the average of the values obtained across the five experimental trials.

Table 5.25: Client Response Time Analysis (without tcpdumper, Pod3 Failed)

	Number of Iterations					
	1	2	3	4	5	Average
Client Response Duration (ms)	31.0	47.0	35.5	36.0	39.5	37.8

The analysis reveals that client response time is not effected significantly. The client response time, recorded as 45.46 ms in the scenario with an active attacker pod, decreased to 37.8 ms during the update. Thus, no additional latency was observed during this period.

Subsequently, in the scenario where tcpdumper was deployed, the tcpdumper container was added to the pods to assess whether the update performed by HM introduced any changes. An additional five iterations of client response time measurements were conducted and no significant variation in the average response time was detected. This confirmed that the presence of the tcpdumper container did not cause any delay in client response times. The results of the client response time analysis with tcpdumper are presented in Table 5.26.

Table 5.26: Client Response Time Analysis (with tcpdumper, Pod3 Failed)

	Number of Iterations					
	1	2	3	4	5	Average
Client Response Duration (ms)	35.5	44.5	40.0	30.5	32.5	36.6

It was determined that the use of the tcpdumper container had no impact on client

response times. Thus, another experiment was conducted in which the client request period was modified. This and subsequent experiments were performed with the tcp-dumper container included. The periodicity was reduced from 2 seconds to 1 second and the HM process period was similarly shortened from 2 seconds to 1 second. The experiment was repeated five times, following the same procedure as in previous trials. No change was seen in client response time during the update. After 1 second, the logs sent by pod2 were redirected from pod3 to pod3rep. Analysis of the average client response time revealed an increase of 6-7 ms compared to earlier experiments. The results of this experiment show that the client response time increases as the communication period decreases. It is consistent with the results we obtained in Table 5.21. The results are presented in Table 5.27.

Table 5.27: Average Client Response Time Analysis - Pod3 Fail Case

	Number of Iterations					
	1	2	3	4	5	Average
Client Response Duration (ms)	31.5	45.5	50.0	33.0	56.5	43.4

After conducting a thorough analysis of the scenario where the attacker targeted pod3, the cases where the attacker attacked other pods were subsequently investigated. Specifically, the cases involving attacks on pod2 and pod1 were examined in sequence. The focus of these analyses was to assess how the client response time was affected. In the case of the attack on pod2, the time at which the HM updated the network was identified using the pod2 and pod2rep logs. Six samples were then taken from the update sequence to construct Table 5.28, with the last message received by pod2 and the first message received by pod2rep corresponding to the 3rd and 4th samples, respectively. This experiment was repeated five times and the client response times were recorded in Table 5.29. No significant change was seen in client response time during the update. Compared to the previous experiment, the average response time decreased from 43.4 ms to 34.3 ms.

No significant change in client response time was observed in cases where pod3 and pod2 failed. So, the following analysis of client response times was examined when

Table 5.28: Client Pod Timestamps During Network Update - Pod2 Fail Case

	Client Request Time (hh:mm:ss.ms)	Client Response Time (hh:mm:ss.ms)
1	12:10:59.894	12:10:59.924
2	12:11:00.930	12:11:00.957
3	12:11:01.963	12:11:01.995
4	12:11:03.001	12:11:03.036
5	12:11:04.043	12:11:04.081
6	12:11:05.087	12:11:05.119

Table 5.29: Average Client Response Time Analysis - Pod2 Fail Case

	Number of Iterations					
	1	2	3	4	5	Average
Client Response Duration (ms)	33.5	40.0	32.0	32.0	34.0	34.3

pod1 was under attack. To evaluate the case, the client's timestamps during the network update and the average response times from five separate experiments were recorded in Tables 5.30 and 5.31.

The log files used for analysis in the event of pod1's failure were presented in Figures 5.25 and 5.26. During the experiment, the time when pod1rep received the first message, as well as the corresponding client log, was recorded simultaneously. Upon reviewing the response value, it is evident that the returned value was indeed 67, confirming the proper functioning of the architecture. Additionally, the first message received by pod1rep is understood from Figure 5.26.

The results indicate that among the three pod failure scenarios, the highest delay occurred when pod3 failed according to Table 5.32. In contrast, failures of pod1 and pod2 yielded less latency in response times. Based on the results of this experiment, it

Table 5.30: Client Pod Timestamps During Network Update - Pod1 Fail Case

	Client Request Time (hh:mm:ss.ms)	Client Response Time (hh:mm:ss.ms)
1	12:36:13.477	12:36:13.516
2	12:36:14.522	12:36:14.550
3	12:36:15.556	12:36:15.585
4	12:36:16.591	12:36:16.631
5	12:36:17.637	12:36:17.669
6	12:36:18.675	12:36:18.705

Table 5.31: Average Client Response Time Analysis - Pod1 Fail Case

	Number of Iterations					
	1	2	3	4	5	Average
Client Response Duration (ms)	34.5	39.5	38.0	41.0	34.5	37.6

is challenging to determine the specific delays caused by the failure of individual pods within the topology, due to its relatively short configuration. Therefore, in Experiment 7, a more detailed analysis was conducted using a serial topology composed of 10 pods.

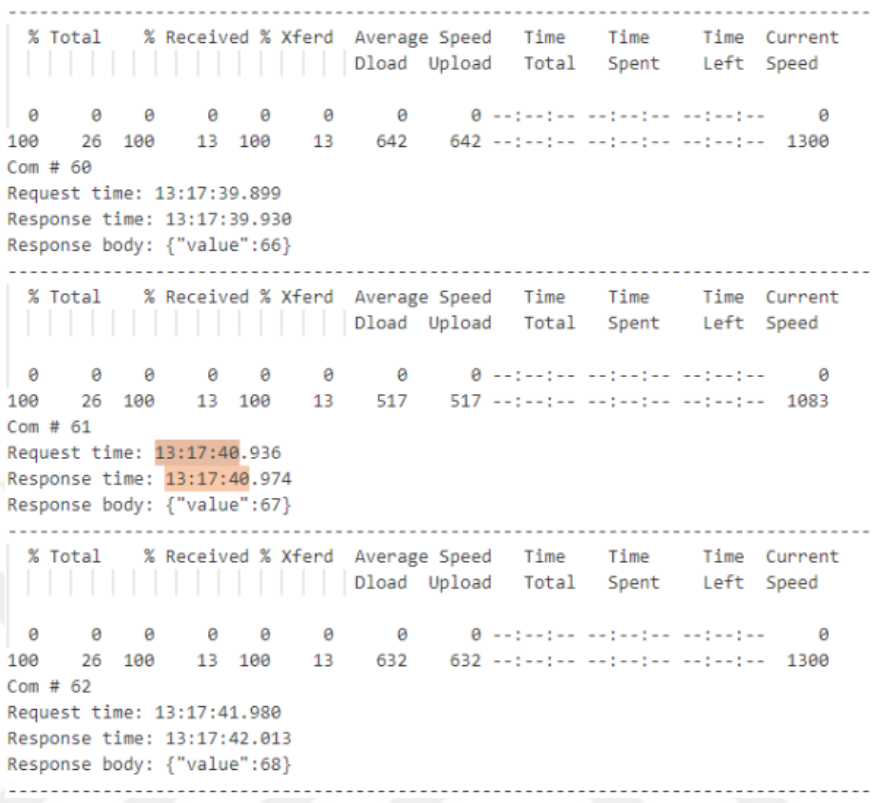


Figure 5.25: Client Log

Table 5.32: Average Client Response Time Analysis

	Failed Pods		
	pod1	pod2	pod3
Average Client Response Duration (ms)	37.6	34.3	43.4

```
[33mPress CTRL+C to quit[0m

Com # 1
req-msg: 61
resp-msg: 67
req-time: 2024-07-31 13:17:40.952212
resp-time: 2024-07-31 13:17:40.968354
resp-dur: 0:00:00.016142
10.68.1.62 - - [31/Jul/2024 13:17:40] "POST /process HTTP/1.1" 200 -

Com # 2
req-msg: 62
resp-msg: 68
req-time: 2024-07-31 13:17:41.994210
resp-time: 2024-07-31 13:17:42.007929
resp-dur: 0:00:00.013719
10.68.1.62 - - [31/Jul/2024 13:17:42] "POST /process HTTP/1.1" 200 -

Com # 3
req-msg: 63
resp-msg: 69
req-time: 2024-07-31 13:17:43.030854
resp-time: 2024-07-31 13:17:43.044330
resp-dur: 0:00:00.013476
10.68.1.62 - - [31/Jul/2024 13:17:43] "POST /process HTTP/1.1" 200 -

Com # 4
10.68.1.62 - - [31/Jul/2024 13:17:44] "POST /process HTTP/1.1" 200 -
req-msg: 64
resp-msg: 70
```

Figure 5.26: Pod1rep Log

5.7 Serial 10-Pod Topology

In this experiment, a 10-pod serial topology was constructed by expanding the 3-pod serially connected topology from the previous experiment. The objective was to determine which pod failure occurring at the beginning, middle or end resulted in the greatest delay. Each pod was configured to send a request to the adjacent pod's service and perform an addition operation using the number associated with its pod name, as illustrated in Figure 5.27.

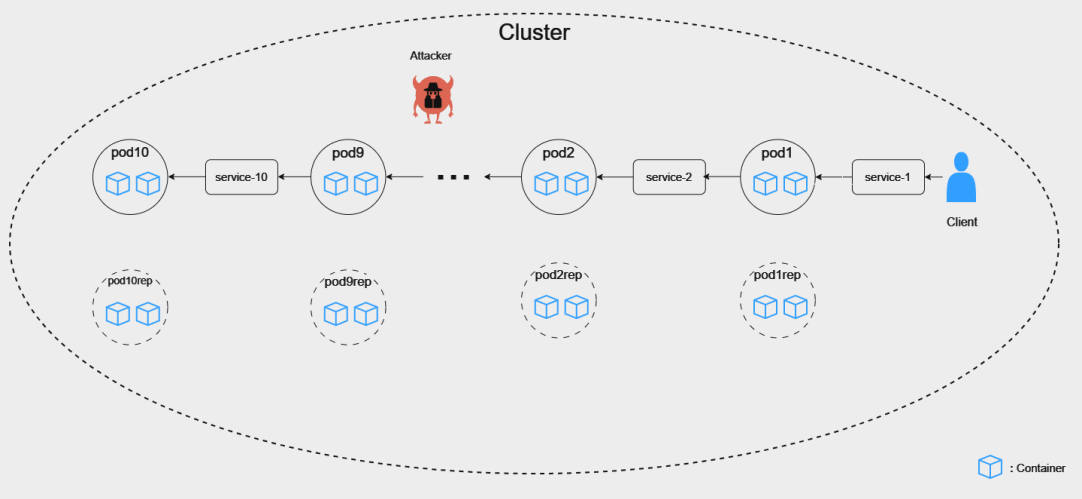


Figure 5.27: Serial 10-Pods Topology

Communication between the pods was facilitated by custom Python containers which interacted via HTTP POST requests. The client periodically sent a value to pod1, where it was incremented by 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 sequentially, before being returned to the client along the same reverse path from pod10. For instance, when the value 1 was sent the expected return value was 56 (1 + 1 + 2 + ... + 10 = 56). Figure 5.28 confirms that this expectation was met as the return value in the first communication sample is indeed 56. This indicates the successful implementation of the topology. Figure 5.28 further shows that periodically transmitted values such as 2, 3 and others were returned incremented by 1.

To evaluate the client response time while the attacker pod was active, 50 communication samples were recorded. This data was collected to enable a comparison of client response times before and after the update. Response time analysis was conducted by

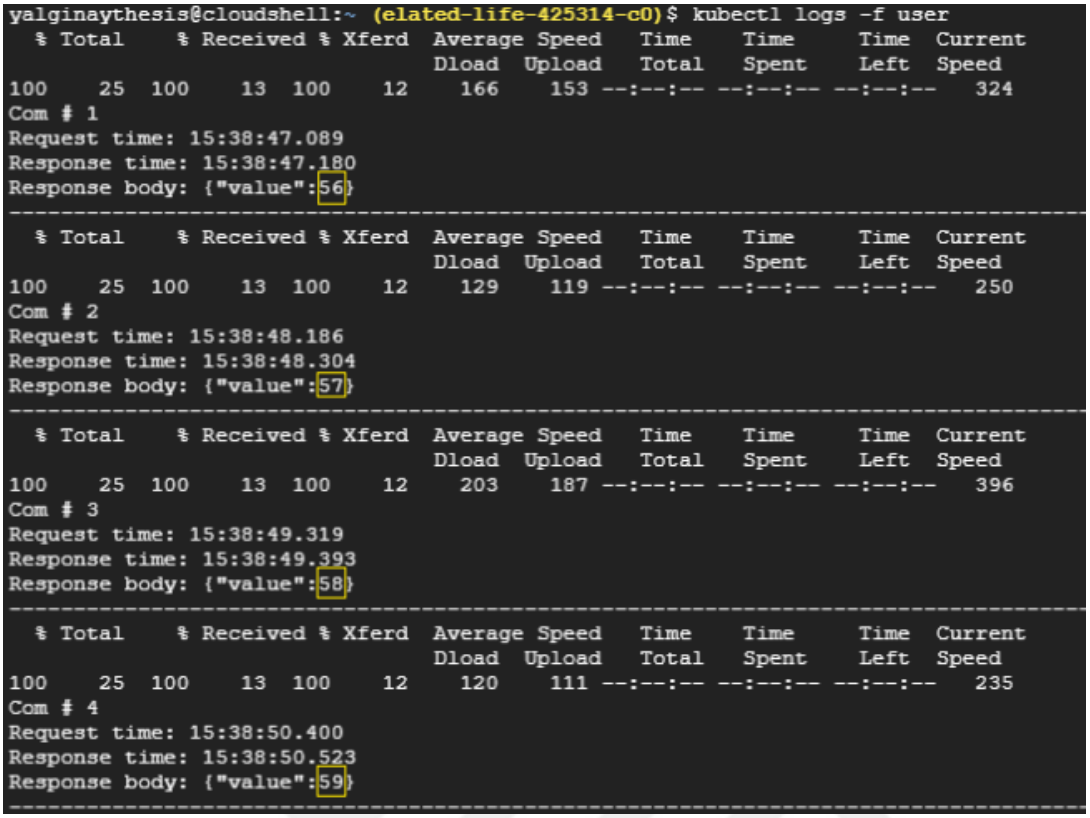


Figure 5.28: Responses of the Client Requests

recording 50 communication samples for all pods involved in the sequential processing of requests, except for the pod10. Since no response was expected from the 10th pod, no data was recorded for it. The average response times from 50 communication samples for the remaining pods are presented in Table 5.33. As indicated in the table, the response times decreased progressively from the client to pod 10. This was an expected result because as we get closer to the endpoint the number of pods the request visits decreases. Consequently, it was verified that the pod names corresponded accurately to their respective network locations.

In this experiment, the older version of the HM module was utilized, as there were no plans to update the HM at the time. In the subsequent stages of the experiment, pods 1, 5 and 10 were targeted by attacks in sequence. The impact of the HM module's updates in response to these attack scenarios was evaluated by observing changes in client response times.

The first analysis focused on an attack targeting pod1. In this instance, the HM mod-

Table 5.33: Client and Pods Response Duration

Pod Names	Response Duration (ms)
client	68.8
pod1	66.31
pod2	61.48
pod3	54.20
pod4	46.18
pod5	37.77
pod6	30.69
pod7	22.98
pod8	15.33
pod9	7.22

ule successfully identified the **attacker** and **risky** pods. For further details on HM detection, refer to Figure 5.29.

The update time was identified through the examination and analysis of the logs from pod1 and pod1rep. Upon reviewing Figure 5.30, it was observed that in communication sample 198, the response value returned was 195. Given that client requests increment by one, the expected response value should have been 196. However, as shown in Figure 5.31, the first message received was indeed 196. An inspection of the timestamps confirmed that no delay and packet loss occurred during this transition. Thus, it can be concluded that the network update was urgently performed.

Upon reviewing Table 5.34, No significant change was seen in client response time during the update. Communication persisted for a duration of 1 second. The experiment involving the failure of pod1 was conducted three times and the average value

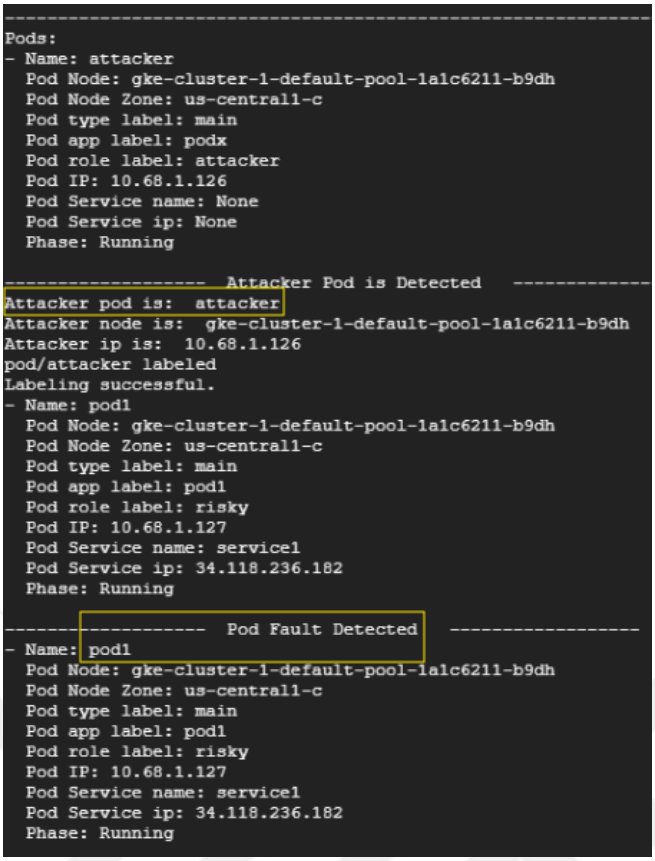


Figure 5.29: HM Action

was subsequently computed. The results of the experiment are presented in Table 5.35.

In the subsequent phase, the scenario involving the failure of pod5 was examined. This experiment focused on analyzing the situation where the attacker pod targeted pod5. During the network update, timestamps for client requests and responses, as well as the duration of responses, were documented in Table 5.36. Additionally, response times and the average response time from three repetitions of the experiment were recorded in Table 5.37. Upon examining Table 5.36, it was noted that client response time does not change throughout the update process. Communication was maintained for a period of 1 second. In the scenario where pod5 failed, the client response time increased by 3 milliseconds compared to the client response time in the scenario where pod1 failed. This increase was not considered to be substantial.

In continuation of the analysis of client response times under pod1 and pod5 fail-

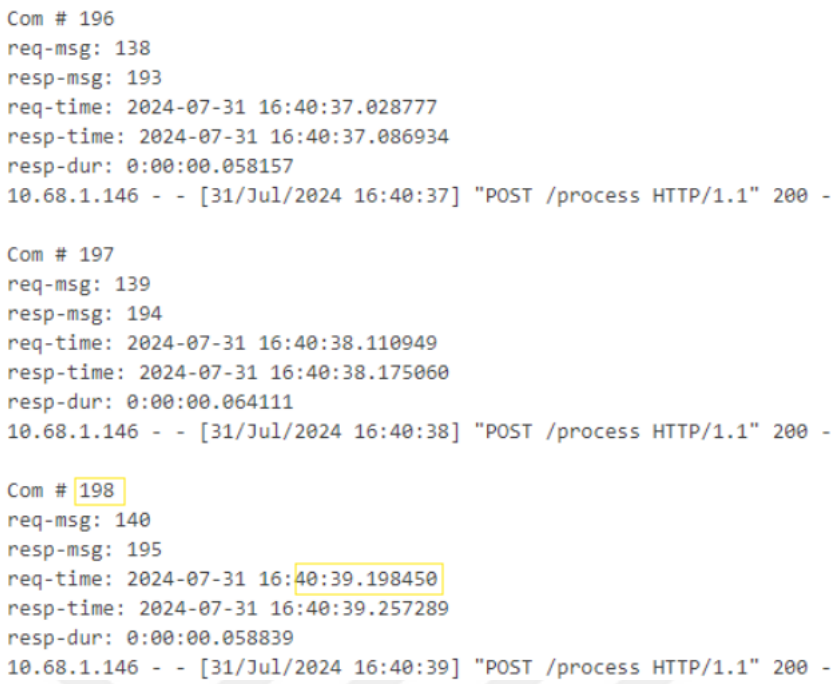


Figure 5.30: Pod1 Log

ure conditions, an additional investigation was carried out for the failure of pod10. The outcomes of these analyses are illustrated in Tables 5.38 and 5.39. As with the previous findings, no delays in client response times were observed and the network operated as expected.

Consequently, an analysis of client response durations in the 10-pod serial topology where pods 1, 5 and 10 failed cases revealed 95.167 ms, 98.0 ms and 89.5 ms respectively. The recordings can be seen in Table 5.40. The variation in latency was minimal within the 1-second communication period. The observed fluctuations in latency can be attributed to instantaneous measurements taken during the network update process. According to the inference made from these results, an update made in the middle of the network causes max latency. A difference of 8.5 ms was observed between an update made in the middle and an update made at the end. Although this error does not seem significant in a 1 second period communication, it is still a finding that should be paid attention to. As a result, except in cases where latency is very critical, which pod fails during the network update does not significantly affect the client latency. However, it should not be ignored that in cases where latency is very critical, the failure of the pod in the middle creates max latency. This suggests that the

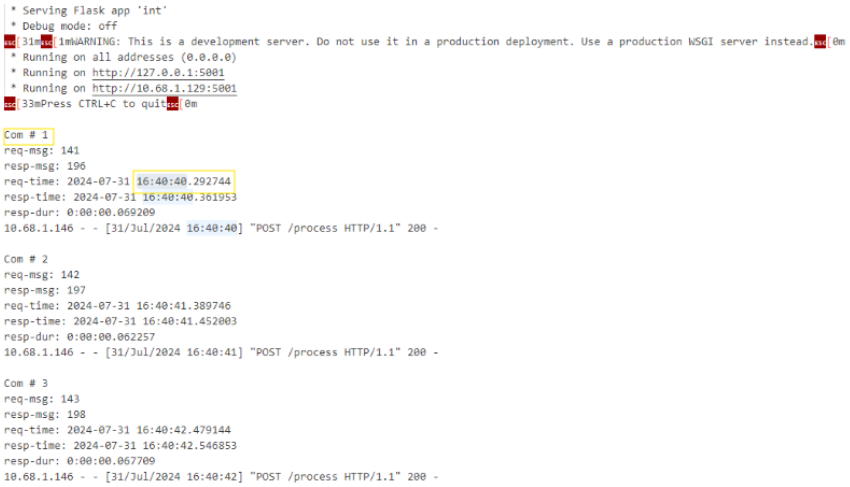


Figure 5.31: Pod1rep log

HM module operates with a uniform update time across all pods, rather than adjusting based on the specific pod affected. This characteristic is crucial in practical scenarios, as attackers may target any pod indiscriminately. If the update mechanism were pod-dependent, the HM module's performance could be compromised, affecting its robustness and stability.

Table 5.34: Client Pod Timestamps During Network Update (Pod1 failed case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	16:40:37.016	16:40:37.092	76
2	16:40:38.098	16:40:38.180	102
3	16:40:39.186	16:40:39.262	76
4	16:40:40.268	16:40:40.370	102
5	16:40:41.378	16:40:41.457	79
6	16:40:42.463	16:40:42.553	90

Table 5.35: Average Client Response Time Analysis - Pod1 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	89.0	98.0	98.5	95.167

Table 5.36: Client Pod Timestamps During Network Update - Pod5 fail case

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	17:04:22.945	17:04:23.096	151
2	17:04:24.109	17:04:24.206	97
3	17:04:25.212	17:04:25.326	114
4	17:04:26.333	17:04:26.444	111
5	17:04:27.450	17:04:27.546	96
6	17:04:28.552	17:04:28.635	83

Table 5.37: Average Client Response Time Analysis- Pod5 fail case

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	112.5	92.0	89.5	98.0

Table 5.38: Client Pod Timestamps During Network Update - Pod10 fail case

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	17:21:52.947	17:21:53.034	87
2	17:21:54.039	17:21:54.119	80
3	17:21:55.125	17:21:55.204	79
4	17:21:56.210	17:21:56.306	96
5	17:21:57.312	17:21:57.402	90
6	17:21:58.408	17:21:58.511	103

Table 5.39: Average Client Response Time Analysis - Pod10 fail case

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	87.5	89.0	92.0	89.5

Table 5.40: Average Client Response Time Analysis Based On Failed Pod

	Failed Pods		
	pod1	pod5	pod10
Average Client Response Duration (ms)	95.167	98.0	89.5

5.8 7-Pod Topology

In this experiment, a more complex topology was established, differing from previous experiments as illustrated in Figure 5.32. Communication between the pods within the cluster was established through HTTP POST requests. The client periodically sent requests to pod7 with each subsequent request incremented by a value of 1. The client's request initially reached pod7 (all requests were routed through services, with each pod represented as an endpoint by a service). Subsequently, pod7 forwarded the request to pod3 and pod6 without any calculation. As depicted in Figure 5.32, the request sent to pod3 is sequentially forwarded to pod2 and pod1, while the request sent to pod6 reaches pod5 and pod4 in the same manner. Upon arriving at the respective endpoints, the requests were aggregated with the pod's numeric identifier. These accumulated values follow their original paths back to the client, being summed with the pod numbers they encounter along the way. The requests from pod3 and pod6 were connected at pod7, where they were further aggregated with pod7's value. Finally, pod7 sent a response back to the client and these values were retrieved from the client log. Additionally, the logs of the pods provided information such as the communication sample number, request time, response time and response duration.

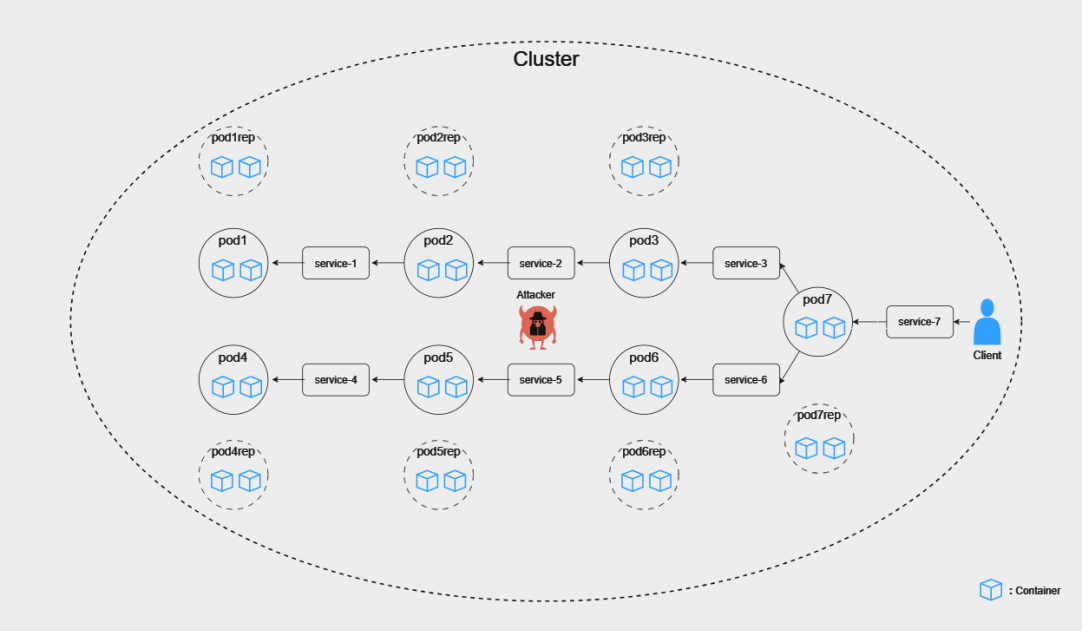


Figure 5.32: 7-Pod Network Topology

The network's operational logic was exemplified through a specific scenario in which the client sends a value of 1 to pod7. This value is transmitted to pod1 and pod4 without modification. The following sequence of events describes the processing of the responses received from pod1 and pod4. The requested value 1 is calculated as shown in Equation 5.1.

$$ClientResponseValue(inp = 1) = (1+1+2+3)+(1+4+5+6)+7 = 30 \quad (5.1)$$

When the input value is set to 2, the output increases by 2 compared to the previous state. This occurs because, even with a 1-unit increase in the input, the increment is applied twice. This phenomenon is illustrated in Equation 5.2. The findings indicate that for every 1 unit increase in the request value, the response value rises by 2 units. This calculation is validated in Figure 5.33, where the response values were recorded as 30, 32, 34, 36 and 38 respectively. These values continued to grow as the client submitted additional requests.

$$ClientResponseValue(inp = 2) = (2+1+2+3)+(2+4+5+6)+7 = 32 \quad (5.2)$$

The containers within the pods were specifically updated for this experiment. Whereas the previously utilized containers handled incoming requests, in this experiment, they processed responses. A distinct structure was implemented for pod7 which in addition to gathering incoming requests, also performs an operation corresponding to the number in its name. As pod7 serves as the connection point for two responses, its analysis will examine how the developed model operates in an unprecedented scenario in the previous experiments. Furthermore, replicas of the seven pods in the network were created initially and their communication was restricted by a network policy.

In the absence of an attacker, response times were recorded individually for the client and its associated pods. As detailed in Table 5.41, the response times ranked from highest to lowest were observed for the client, pod7, pod3, pod6, pod2 and pod5 respectively. Upon reviewing Figure 5.32, it was anticipated that response times would

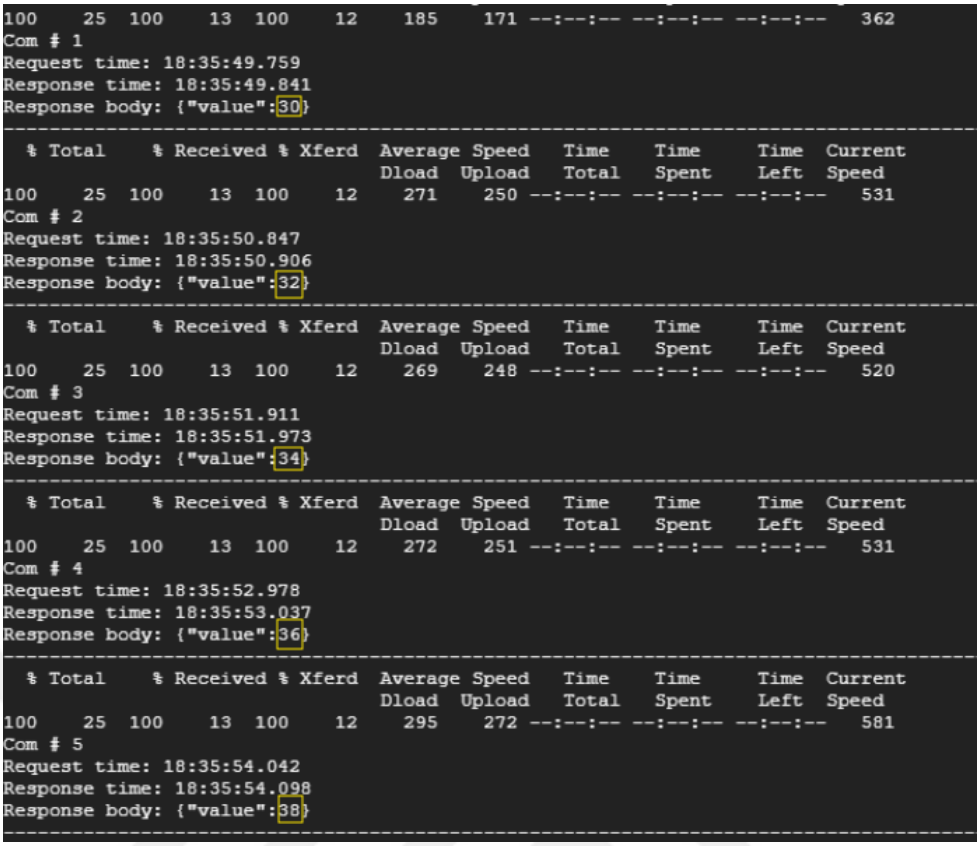


Figure 5.33: Client Responses

decrease as the endpoints were approached. This expected outcome confirmed that the network was implemented correctly.

In the experiment, scenarios in which the attacker pod targeted pod1, pod5, pod3 and pod7 were systematically examined. During the risky pod selection process, particular attention was given to choosing pods that occupied distinct positions within the network topology. The analysis focused on evaluating the variations in client response times under conditions where different network positions experienced failures. At the time of the experiment, the HM had not been updated yet. Therefore, the attacker and risky pod detection was performed with the old version of the HM.

Firstly, the pod1 fail case was investigated. The scenario in which HM detects this error upon the failure of pod1 is illustrated in Figure 5.34. As shown in figure, attacker and risky pods are found successfully.

The network update timing was determined by identifying the time between the last

Table 5.41: Pods Average Response Duration (50 Communication Sample)

Pod Names	Response Duration (ms)
client	66.18
pod7	46.24
pod3	15.89
pod6	14.59
pod2	7.38
pod5	7.07

message received by pod1 and the first message received by pod1rep. The client's log was analyzed during this interval. The client's request time, response time and response durations were recorded in Table 5.42. The timestamp associated with communication sample 3 indicates when pod1 received the final message from pod2, while communication sample 4 marks when pod1rep received the first message from pod2. Communication samples 1 and 2 correspond to the two instances before the update and samples 5 and 6 represent the two instances following the update. As a result of repeating the experiment three times, the average client response times during updates were recorded. These values were documented by taking the average of communication samples 3 and 4. The 3 different trial results are seen in Table 5.43.

According to Table 5.43, no communication latency was seen during the update. The client latency, which was initially 66.18, was recorded as 72.2 during the update of 3 different trials. Based on the results, there is no significant increase was observed.

After analyzing the situation where pod1 failed, the same analysis was repeated for pod5. In the situation where pod5 failed, the time when the network was updated was determined by looking at the pod5 and pod5rep logs. The client log corresponding to this time was recorded. This record is seen in Figure 5.35.

The relevant timestamps were recorded in Table 5.44 so that the values during the

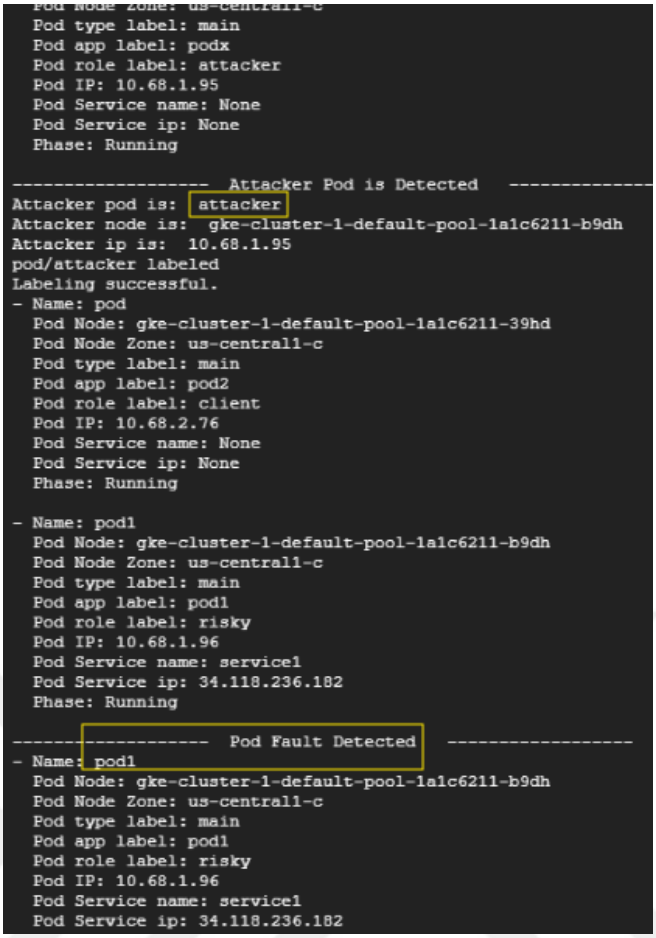


Figure 5.34: HM Action

update correspond to communication samples 3 and 4. The response duration average of the 3rd and 4th communication samples was taken and added also. In addition, the client response durations obtained by repeating the experiment 3 times were also recorded in Table 5.45.

In the event of Pod5 failure, no latency in communication was detected. Communication persisted at 1 second. The client response time, initially recorded at 66.18 seconds, was measured at 66.30 seconds during the update process. No significant delay was observed.

The same analysis was conducted for pod3. In the event of pod3's failure, the network update time was identified by examining the logs of pod3 and its replica pod3rep. Then, the corresponding client log for this time was documented. The relevant timestamps were recorded in Table 5.46 to ensure that the values during the update corre-

Table 5.42: Client Pod Timestamps During Network Update - Pod1 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	18:42:50.417	18:42:50.486	69
2	18:42:51.492	18:42:51.548	56
3	18:42:52.554	18:42:52.612	58
4	18:42:53.618	18:42:53.688	70
5	18:42:54.694	18:42:54.767	73
6	18:42:55.778	18:42:55.838	60

Table 5.43: Average Client Response Time Analysis (Pod1 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	64.0	92.5	60.0	72.2

spond to communication samples 3 and 4. The average response duration for these 3rd and 4th communication samples was calculated and included in Table 5.47, alongside the client response durations obtained from repeating the experiment 3 times.

In the event of a Pod3 failure, no communication latency was detected. Communication proceeded with a consistent 1-second period. The client response time, initially recorded at 66.18 milliseconds, was measured at 62.30 milliseconds during the update. While this represents a minor reduction in latency, the change was not substantial enough to be considered significant.

Finally, the situation where pod7 failed was analyzed. Unlike the failure situations of other pods, the critical point here was that pod7 was connected to 2 separate pods. How this situation affected the client response time at the time of the failure was analyzed. Figure 5.36, with 10.96.2.111 representing the IP address of the attacker

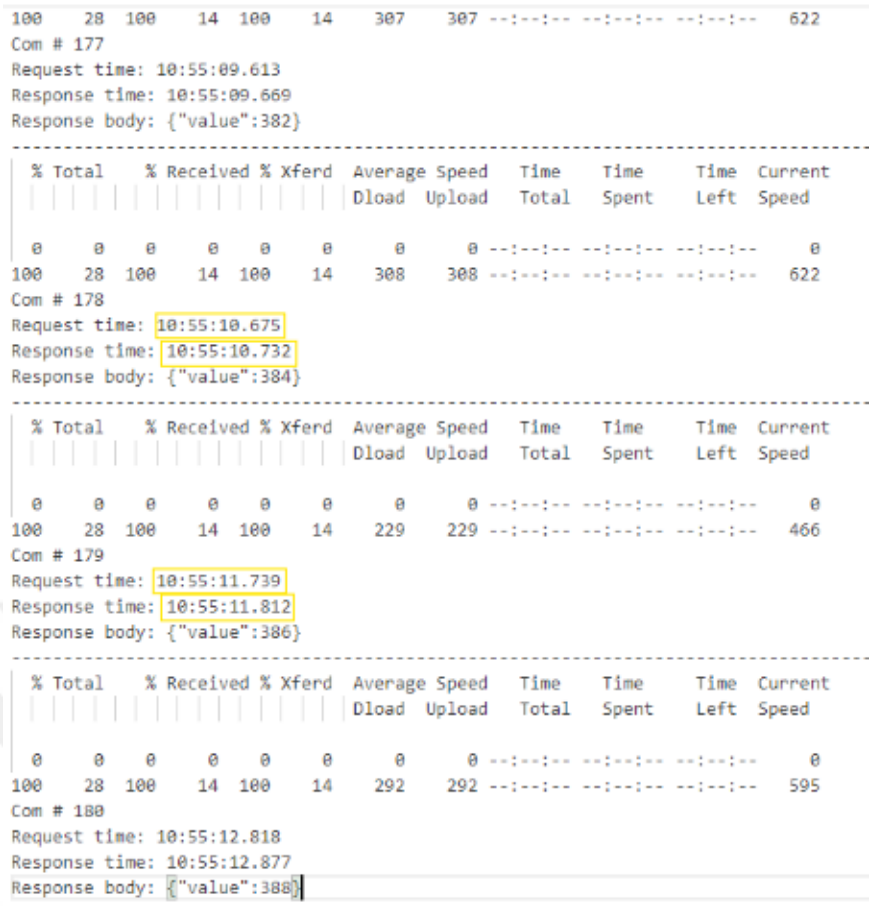


Figure 5.35: Client Responses (Pod1 Fail Case)

pod and 10.96.2.125 representing the IP address of pod7. To differentiate the client and attacker requests, different messages were sent as seen in req-msg and resp-msg.

To understand the variations in client response time during the update, the logs of pod7 and its backup pod7rep were thoroughly examined. The timestamps indicating when pod7 last received a message from the client and when pod7rep received a message were recorded as 11:27:39 and 11:27:40, respectively, as shown in Figures 5.37 and 5.38. The corresponding client log for these messages was also analyzed, with the relevant logs presented in Figure 5.39.

Subsequently, to conduct a detailed analysis of the client response time, these recorded values were incorporated into Table 5.48. The analysis was performed similarly to previous experiments, averaging the two response times recorded during the update. Table 5.49 was then generated by documenting the client response durations across

Table 5.44: Client Pod Timestamps During Network Update(Pod5 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	10:55:08.545	10:55:08.607	62
2	10:55:09.613	10:55:09.669	56
3	10:55:10.675	10:55:10.732	57
4	10:55:11.739	10:55:11.812	73
5	10:55:12.818	10:55:12.877	59
6	10:55:13.883	10:55:13.942	59

Table 5.45: Average Client Response Time Analysis (Pod5 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	65.0	65.5	68.5	66.3

three repetitions of the experiment. To mitigate potential deviations, 3 trials were performed. The final column of the table contains the average of these values.

The client response time, initially recorded as 66.18, was observed as detailed in Table 5.50 for scenarios involving failing pods. When the communication period was set to 1 second, variations of ±4 ms were noted, though this variance was not considered indicative of significant latency. This experiment sufficiently demonstrates that the network functions effectively in a different topology and confirms that network updates do not introduce notable latency in scenarios with a single client. Nevertheless, the next experiment was conducted to assess how latency is affected when the number of clients increases and an attacker executes a real DoS attack.

Table 5.46: Client Pod Timestamps During Network Update (Pod3 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	11:14:18.907	11:14:18.969	62
2	11:14:19.974	11:14:20.036	62
3	11:14:21.042	11:14:21.096	54
4	11:14:22.102	11:14:22.171	69
5	11:14:23.177	11:14:23.237	60
6	11:14:24.245	11:14:24.350	105

Table 5.47: Average Client Response Time Analysis (Pod3 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	61.5	63.5	61.0	62.0

Table 5.48: Client Pod Timestamps During Network Update(Pod7 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	11:27:37.159	11:27:37.249	90
2	11:27:38.261	11:27:38.319	58
3	11:27:39.326	11:27:39.391	65
4	11:27:40.398	11:27:40.466	68
5	11:27:41.473	11:27:41.535	62
6	11:27:42.540	11:27:42.598	58

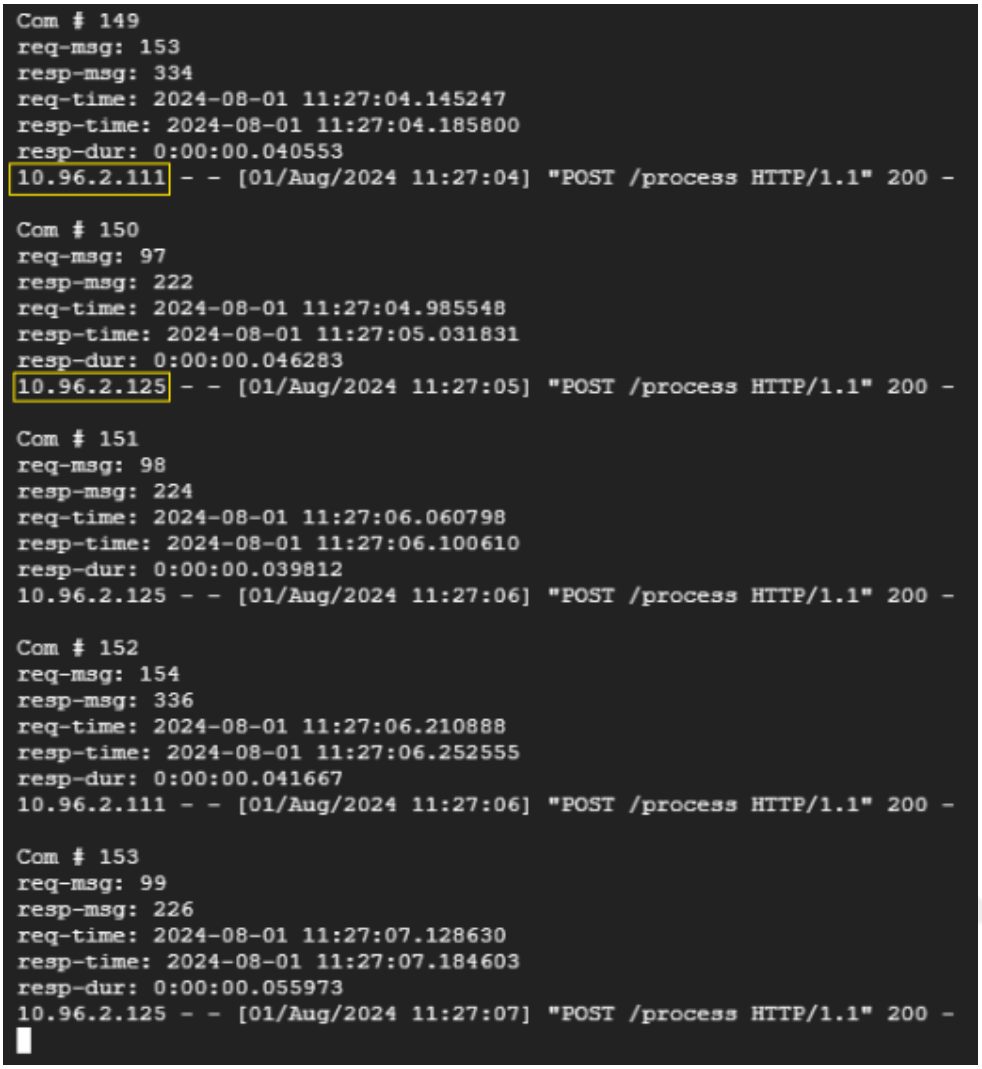


Figure 5.36: Client Responses (Pod7 Fail Case)

Table 5.49: Average Client Response Time Analysis (Pod7 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	66.5	83.5	60.0	70.0

```
resp-msg: 280
req-time: 2024-08-01 11:27:36.099809
resp-time: 2024-08-01 11:27:36.139042
resp-dur: 0:00:00.039233
10.96.2.125 - - [01/Aug/2024 11:27:36] "POST /process HTTP/1.1" 200 -

Com # 182
req-msg: 127
resp-msg: 282
req-time: 2024-08-01 11:27:37.170543
resp-time: 2024-08-01 11:27:37.221022
resp-dur: 0:00:00.050479
10.96.2.125 - - [01/Aug/2024 11:27:37] "POST /process HTTP/1.1" 200 -

Com # 183
req-msg: 128
resp-msg: 284
req-time: 2024-08-01 11:27:38.273371
resp-time: 2024-08-01 11:27:38.313698
resp-dur: 0:00:00.040327
10.96.2.125 - - [01/Aug/2024 11:27:38] "POST /process HTTP/1.1" 200 -

Com # 184
req-msg: 129
resp-msg: 286
req-time: 2024-08-01 11:27:39.340296
resp-time: 2024-08-01 11:27:39.386601
resp-dur: 0:00:00.046305
10.96.2.125 - - [01/Aug/2024 11:27:39] "POST /process HTTP/1.1" 200 -
```

Figure 5.37: Pod7 Last Log Before Network Update

Table 5.50: Average Client Response Time Analysis

	Failed Pods			
	pod1	pod5	pod3	pod7
Average Client Response Duration (ms)	72.2	66.3	62.0	70.0

```
 * Serving Flask app 'dual_int'
 * Debug mode: off
[31m[1mWARNING: This is a development server. Do not use it in a product
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5007
 * Running on http://10.96.2.124:5007
[33mPress CTRL+C to quit[0m

Com # 1
req-msg: 130
resp-msg: 288
req-time: 2024-08-01 11:27:40.413678
resp-time: 2024-08-01 11:27:40.461043
resp-dur: 0:00:00.047365
10.96.2.125 - - [01/Aug/2024 11:27:40] "POST /process HTTP/1.1" 200 -

Com # 2
req-msg: 131
resp-msg: 290
req-time: 2024-08-01 11:27:41.486125
resp-time: 2024-08-01 11:27:41.526853
resp-dur: 0:00:00.040728
10.96.2.125 - - [01/Aug/2024 11:27:41] "POST /process HTTP/1.1" 200 -

Com # 3
req-msg: 132
resp-msg: 292
req-time: 2024-08-01 11:27:42.551984
resp-time: 2024-08-01 11:27:42.589906
resp-dur: 0:00:00.037922
10.96.2.125 - - [01/Aug/2024 11:27:42] "POST /process HTTP/1.1" 200 -

Com # 4
req-msg: 133
resp-msg: 294
req-time: 2024-08-01 11:27:43.617713
```

Figure 5.38: Pod7rep First Log After Network Update

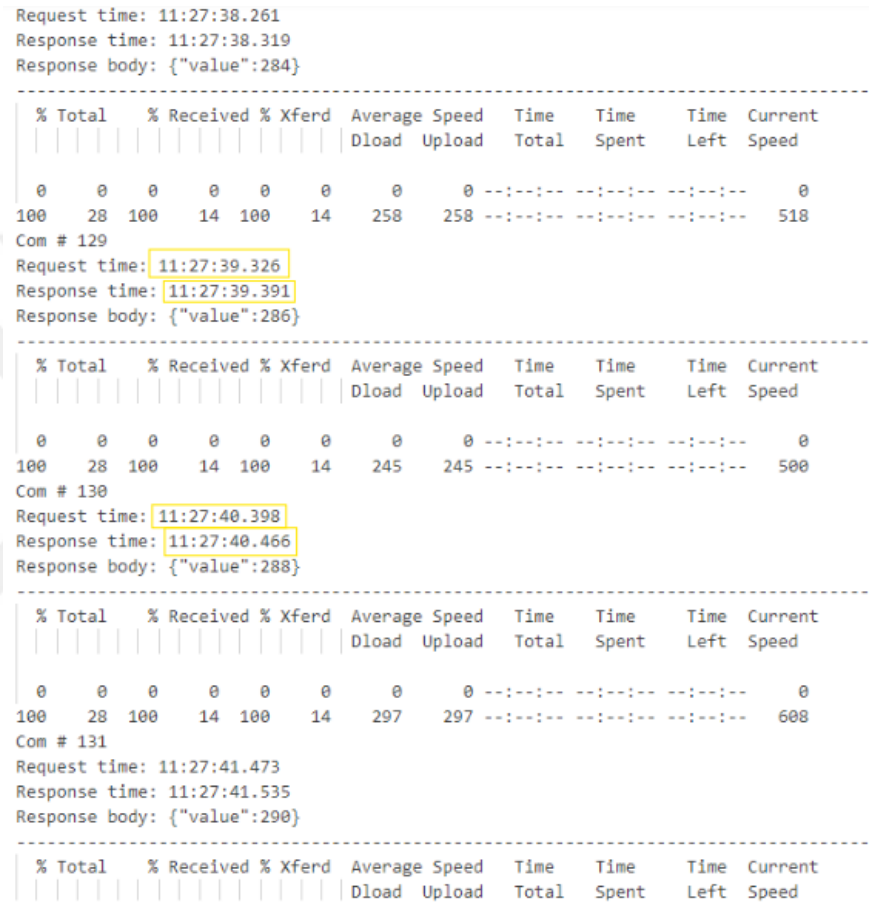


Figure 5.39: Client Log During Network Update

5.9 Multi-Client Analysis on 7-Pods Topology

In the preceding experiments, only single clients were employed. Additionally, during the network update, client response duration analyses involved collecting **one sample before** and **one sample after** updating the topology. In this experiment, the analysis was further refined by conducting scalability tests under varying client numbers: 1, 2, 5, 10, 25, 50 and 100. Furthermore, a detailed examination was carried out to assess the impact of the attacker on the network as the number of clients changed. In addition, client response duration analyses were performed before and after the attacker was detected.

The initial analysis focused on scenarios where client(s) sent requests without the presence of an attacker in the network. For each client configuration, 120 communication samples were collected and the average client response times were calculated. Subsequently, the scenario involving an attacker pod was examined, where the attacker launched a DoS attack by sending significantly more requests per unit of time to the target pod than the client did. In this scenario, 120 communication samples were taken and the average client response times were calculated. In a third scenario, 60 communication samples were taken once the attacker joined the network. After this, the role labels of the pods were changed to "attacker" and "risky," enabling the HM module to identify the attacker and risky pods. It should be considered that in this experiment updated version of HM was used. Then network was updated another 60 communication samples were recorded to analyze how the client response duration changed post-update. It was anticipated that the client response duration would decrease because the attacker had previously kept the server so occupied that a reduction in this busyness was expected after the update.

The experiment was conducted using the 7-Pods topology which was previously utilized in experiment in section 5.8. The target of the attacker was selected as pod7. In contrast to the experiment in section 5.8, the placement of pods on specific nodes was intentionally controlled as shown in 5.40. Pod1, pod3, pod4, pod6 and their respective backup pods were deployed on node A, while pod2, pod5, pod7 and their backups were deployed on node B. All client pods were initially deployed on node C to ensure a controlled allocation of resources across the nodes. However, when the

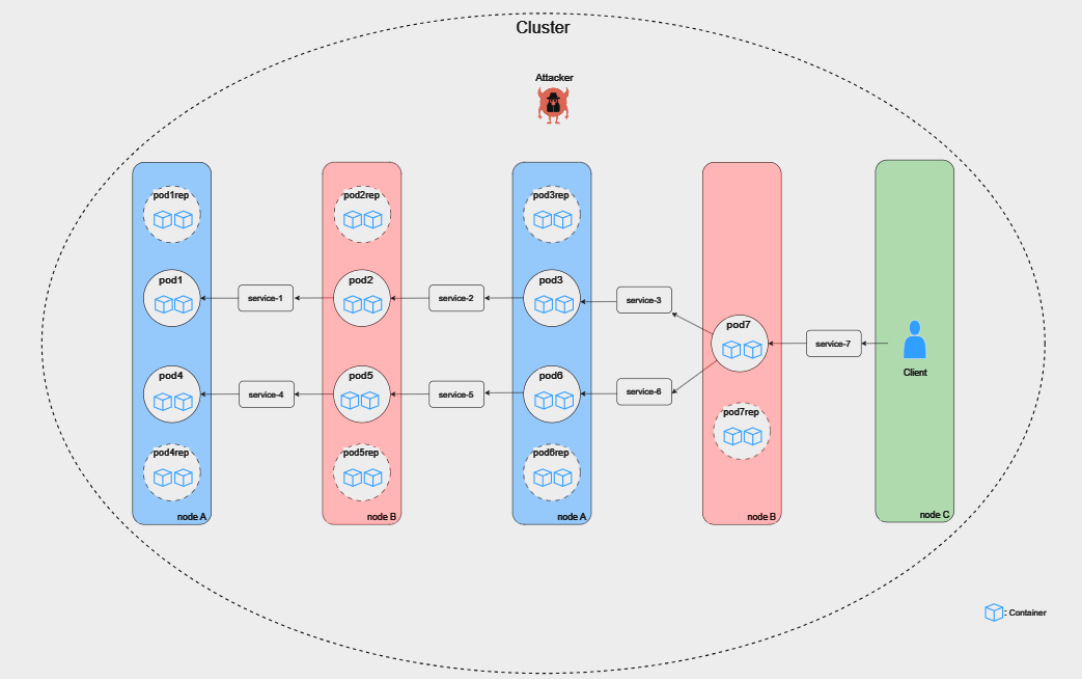


Figure 5.40: 7-Pods Network Topology with Corresponding Nodes

number of pods reaches 50 or more this distribution strategy is changed. A detailed explanation of this modification shall be provided in the relevant section. The node on which the attacker will be created has not been specifically set, because it is not desired for the attacker to be the result of a biased experiment depending on where it will attack from when it is created.

To change the number of clients, a storage configuration file was created. The number of clients could be easily scaled from this file.

The experiment commenced with a scenario involving a single client without attacker pod. The configuration of the network's pods, services, nodes and client IP's are illustrated in Figure 5.41.

The client log without attacker scenario is seen in Figure 5.42. According to the Equation 5.1, it is understood that network was established successfully.

The average client response times for the single client scenario are presented in Table 5.51. The first column displays the average response time when no attacker was present. The second column reports the average response time when the attacker pod

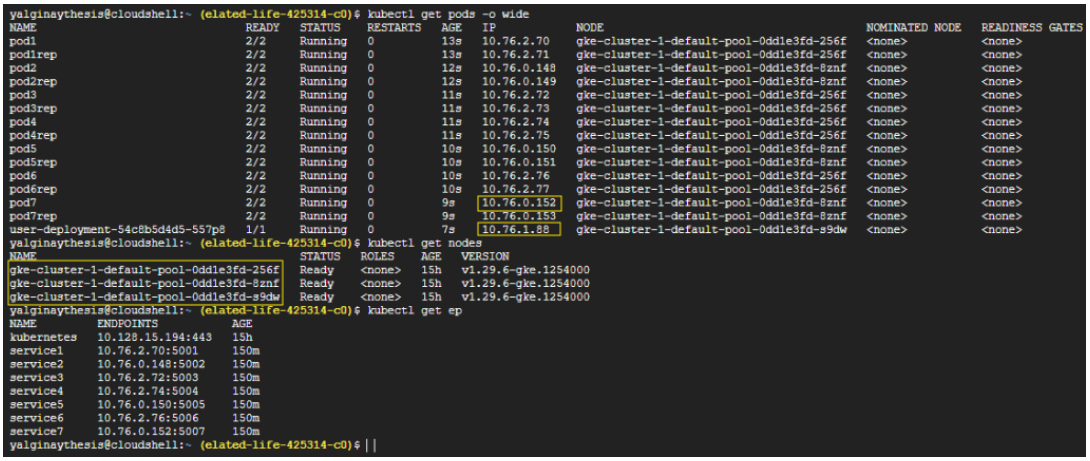


Figure 5.41: Pod, Node and Service IP's (without Attacker Scenario)

targeted pod7. In both scenarios, the average values of 120 communication samples were recorded. Given that the communication period was 1 second, the recordings spanned 2 minutes for each case. After these initial recordings, the scenario with the attacker targeting pod7 was repeated. However, in this iteration, the attacker and risky pods were detected after the 60th sample, prompting a network update. The average of the first 60 samples is shown in the third column. Following the network update, an additional 60 samples were collected and their average is presented in the fourth column. To mitigate the impact of the network update, samples from the 60th to the 80th were excluded from the calculation, with the average being derived from samples 80 through 140. This methodology was also applied in subsequent experiments where the number of clients was increased.

There is an important point to note that the clients expressed with the client-<number> in the 1st and 2nd columns are not the same clients. However, the 3rd and 4th columns correspond to the same client since they occur in the same scenario. Additionally, clients in the 3rd and 4th columns and the clients in the 1st and 2nd columns are not the same. Lastly, all the experimenters performed for 1, 2, 5, 10, 25, 50 and 100 client scenarios were performed 3 times and average values were recorded.

By examining the 3rd and 4th columns in Table 5.51, it was observed that there is a slight improvement in client response time following the HM update in the scenario involving a single client. The 1st and 2nd columns illustrate the conditions with and without a DoS attack on pod7 by the attacker pod. When the attacker pod executed a

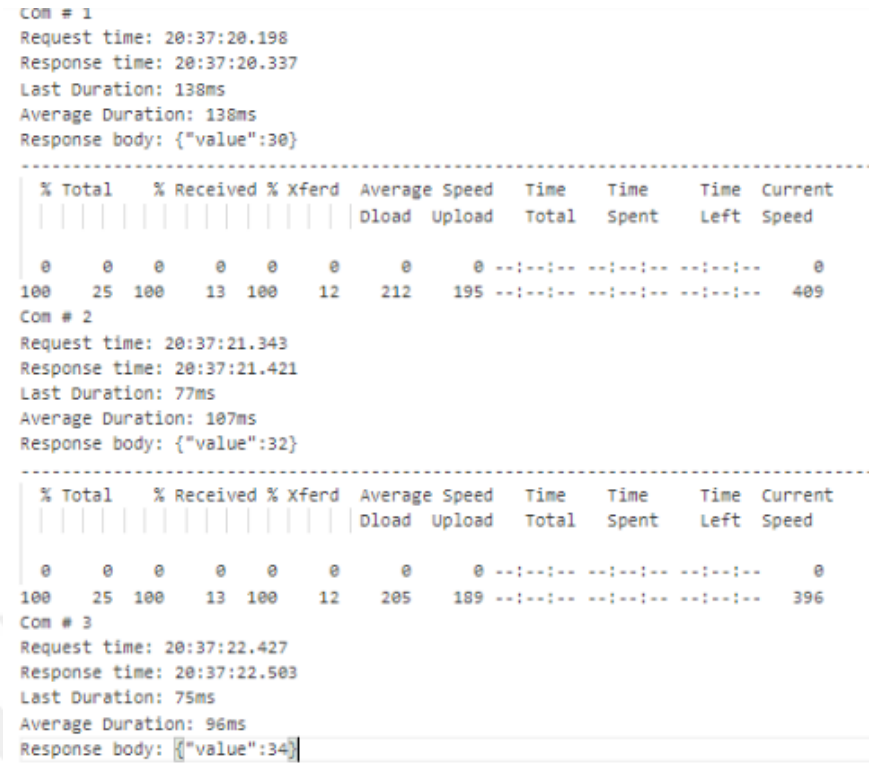


Figure 5.42: Client Log without Attacker Scenario

DoS attack, the response time for the single client increased by 48.275%. This table demonstrates that the developed model enhanced the client response time.

Subsequently, the same experiments were conducted in a scenario involving 2 clients. The individual response times for each client, as well as the average response times, are presented in Table 5.52.

As detailed in Table 5.52, an analysis of client response times was conducted. Upon examining the response times for both individual clients and their average, a 5.30% improvement was noted before and after error detection. Furthermore, a review of the 2-minute recordings revealed an average delay of 14.88% in client response time when the attacker targeted pod7. This experiment also confirmed that the model relieved the network.

In the next stage, the scenario where the number of clients is 5 was analyzed. The pods created in this scenario and the nodes on which the pods run are available in Figure 5.44. It also verifies that the nodes host the pods correctly as specified.

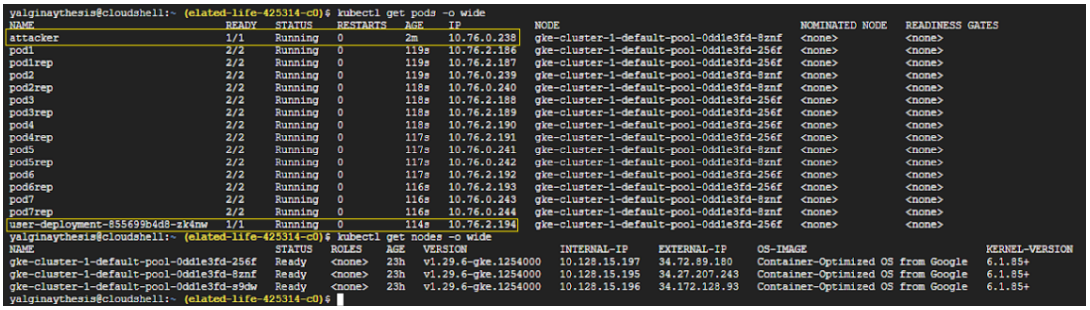


Figure 5.43: Pod, Node and Service IP's (with Attacker Scenario)

Table 5.51: 1 Client Average Response Time

	Average Client Response time (ms)			
Client	Without attacker 120 samples	With Attacker 120 samples	Before Network Update 60 samples	After Network Update 60 samples
1	58	86	84	80

The individual response times for each client, as well as the average response times, are presented in Table 5.53.

Table 5.53 shows the client response times recordings. After the attacker pod was intervened, there was a significant relief in the network. The client response time, which was 85.0 ms on average before the attacker intervened, dropped to 50.5 ms after the intervention. An average speedup of approximately 40% was observed. Apart from the average, when the pods are examined individually, it is possible to say that there is a relief in each of them. When it was looked at the 1st and 2nd columns, it is possible to say that the attacker pod slows down the client response times by an average of 33% in the 5-clients scenario.

Following the analysis of the 5-client scenario, the experiment was extended to include a 10-client scenario. Identical experiments were conducted with 10 clients in this phase. The actions taken by the HM during the error occurrence in this scenario are depicted in Figure 5.45. The **attacker** and **risky** pods were accurately identified. Additionally, the client response time analysis for the 10-client scenario is presented in Table 5.54.

Table 5.52: 2 Clients Average Response Times

	Average Client Response Time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	Before Network Update 60 samples	After Network Update 60 samples
1	61	64	74	70
2	60	75	77	73
Average	60.5	69.5	75.5	71.5

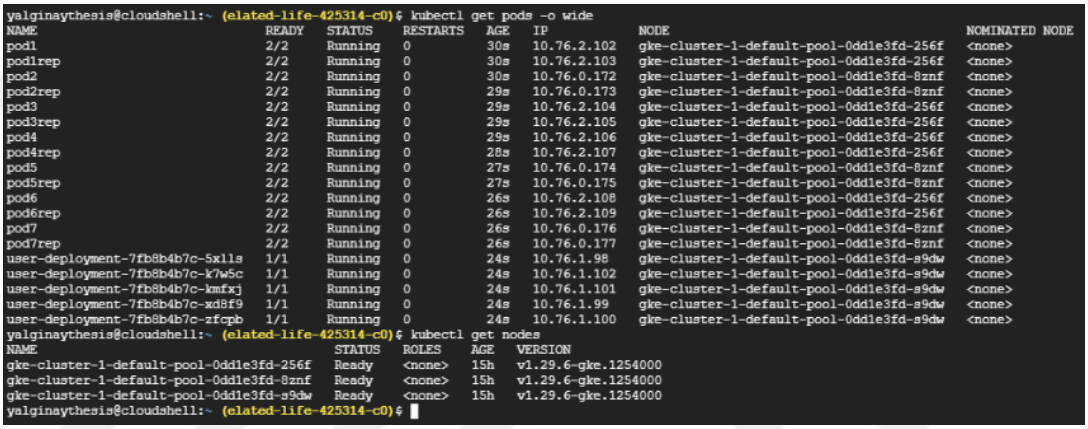


Figure 5.44: Pod and Node IPs (with Attacker 5-Clients Scenario)

As demonstrated in Table 5.54, the client response duration increased in the presence of the attacker, consistent with previous experiments. However, in this scenario, the magnitude of the increase was more pronounced than in earlier experiments. Specifically, when 120 samples were collected under the influence of the attacker, the client response time increased by 48.08%. Additionally, analysis of the third and fourth columns reveals that the most significant improvement was observed in this experiment, with the client response time decreasing by 52.04%. This indicates a substantial alleviation in network load. Apart from the average, the response time of each client decreased.

In scenarios involving 1, 2, 5 and 10 clients, the creation of pods on a single machine proceeded without issue. However, challenges arose when the number of pods increased to 25 or more. Notably, an OutOfPods error occurred when attempting to

Table 5.53: 5 Clients Average Response Times

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	Before Network Update 60 samples	After Network Update 60 samples
1	61	80	71	55
2	60	93	87	49
3	62	81	86	54
4	60	75	90	46
5	59	80	81	49
Average	60.4	81.8	85.0	50.6

create 50 and 100 pods. To address this issue, clients were distributed across three separate machines, without imposing any target node restrictions. The results for the experiments involving 25 clients are presented in Table 5.55.

Table 5.55 presents the average client response times for 120 samples in the presence of an attacker. The analysis revealed that as the number of samples increased, client response times also increased. This increase was attributed to the accumulation of clients and requests from the attacker pod over time, which led to prolonged response times. It was observed that response times for all clients, both before and after the network update, showed a reduction. However, this reduction was not as pronounced as in earlier experimental stages, amounting to an improvement of 10.86%. This was due to 25 clients sending requests to the same server within 1 second. Despite blocking the attacker, the high volume of requests persisted. The number of legitimate requests and those from the attacker were more comparable than in previous experiments, resulting in a less significant relief of approximately 50% when the attacker was disabled. Conducting this experiment with an increased period for the number of legitimate requests might have yielded greater relief, but the observed improvement still represents a notable recovery.

Upon completion of the experiment involving 25 clients, a subsequent experiment

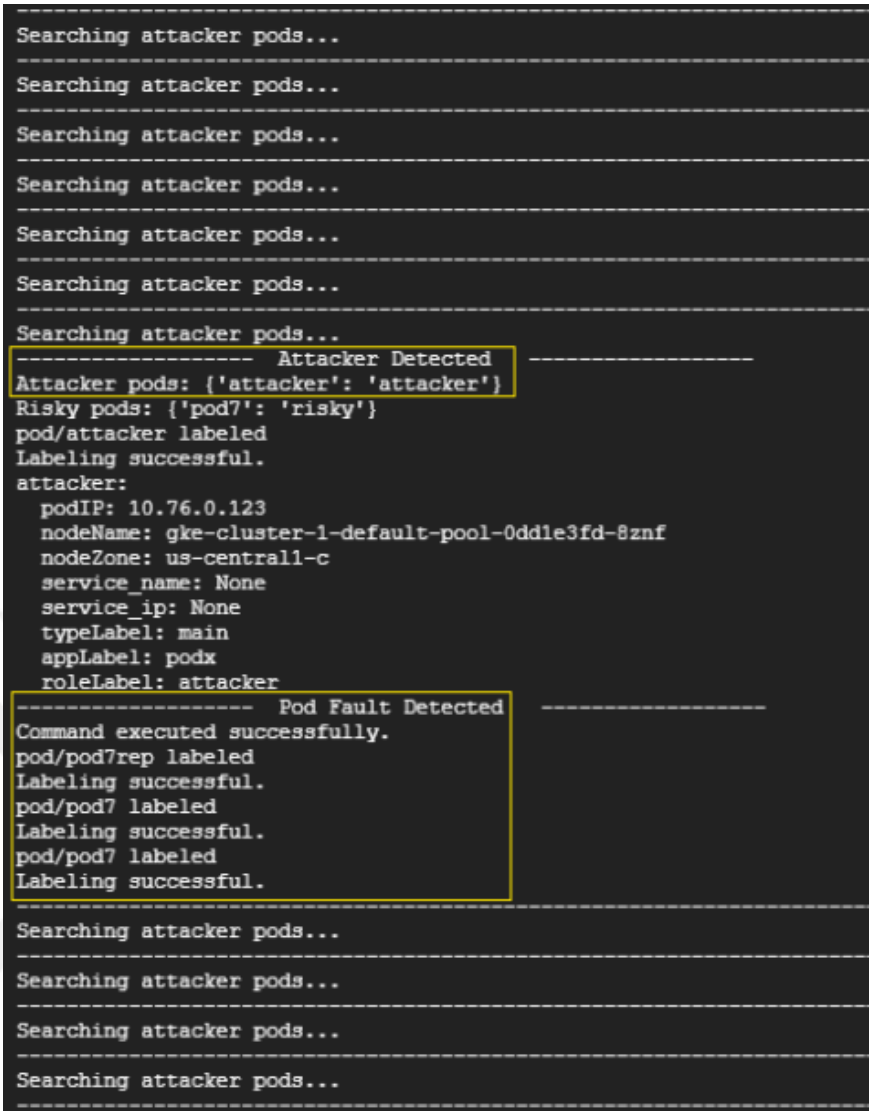


Figure 5.45: HM Action in 10-Clients Scenario

was conducted with 50 clients to further enhance the analysis. In this scenario, a previously unobserved issue was identified. Despite the inclusion of an attacker pod in both the 60 and 120 sample scenarios, client response times nearly doubled. The increase in the number of client requests, irrespective of the attacker's presence, overwhelmed the server, leading to a cascading inability to manage these requests over time, which consequently increased response times. When client response time values were recorded separately before and after sample 60, the attacker's effect could not be observed. Although the attacker's effect was blocked in the recording after sample 60, longer response times were obtained. Thus, an alternative analytical method was

Table 5.54: 10 Clients Average Response Times

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	Before Network Update 60 samples	After Network Update 60 samples
1	73	101	140	76
2	80	113	135	61
3	65	105	145	81
4	62	120	123	49
5	73	90	122	50
6	77	118	122	60
7	78	119	129	61
8	70	110	132	62
9	76	99	123	49
10	76	106	128	74
Average	73	108.1	129.9	62.3

employed instead of simply averaging network performance before and after updates. Specifically, the network was updated again at the 60th sample and communication samples were recorded at the 120$^{\mathbf{th}}$ and 140th intervals. This approach allowed for a comparison of average response times with and without the attacker when the network update occurred midway. The 140-sample analysis was conducted to demonstrate that, although cumulative response times increase over time, client response times were lower post-network intervention compared to the 120-sample scenario with the attacker. Additionally, to highlight the attacker's impact more clearly, the request sending period for the attacker was reduced to 1%. The outcomes of these experiments are detailed in Table 5.56 and Table 5.57.

Upon examining Tables 5.56 and 5.57, it is evident that the impact of the attack in the 50-pod scenario significantly increased response times, rising from 507 ms to

1443 ms. This represents a substantial delay, with response times tripling compared to expected values. Moreover, following the network update of the HM module, the average client response time improved from 1443 ms to 823 ms. Additionally, analysis of logs from 50 individual pods revealed a slight reduction in response times for each client, regardless of performance variability. Even when the sample number is 140, the network seems to be much more relaxed compared to the attacker situation. This experiment provides insight into the performance of the model developed for the 50-client scenario.

In the subsequent and final experiment, testing was conducted with a 100-client scenario, where clients were distributed across different nodes. Some of the pods can be seen in Figure 5.46.

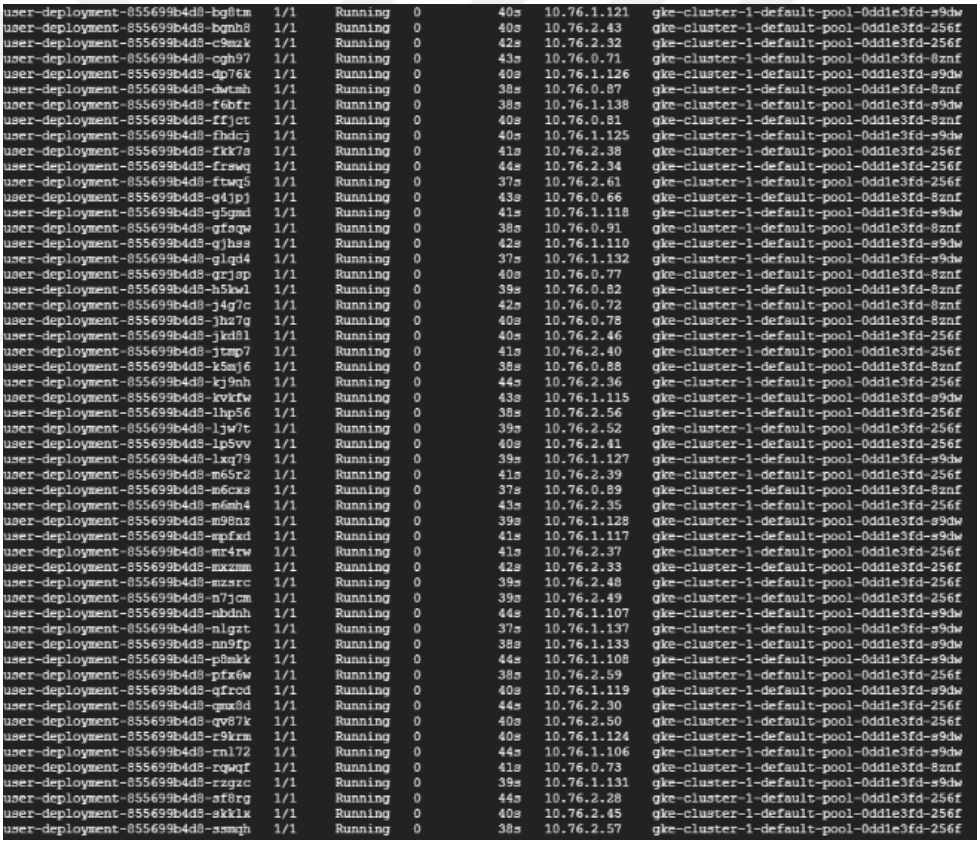


Figure 5.46: Group of Deployed Pods in The 100 Pods Cluster

The same analyses applied in the 50-client experiment were conducted for the 100-clients setup. The experimental results are detailed in Tables 5.58, 5.59, 5.60 and 5.61.

Upon reviewing Tables 5.58, 5.59, 5.60 and 5.61, it becomes apparent that the attack in the 100-clients scenario had a significant impact on response times, increasing from 6201 ms to 9469.5 ms. This indicates a considerable delay, with response times tripling compared to expected levels. However, following the network update of the HM module, the average client response time improved from 9468.5 ms to 7844.5 ms. Furthermore, an analysis of logs from 100 individual clients demonstrated a great reduction in response times for each client. Even when the sample size reached 140, the network appeared to be more stable compared to the conditions during the attack. This experiment provides valuable insights into the performance of the model developed for the 100-client scenario. As a result, although the relaxation in the network could not be observed in cases where the number of pods was low, this relaxation was observed very clearly in cases where the number of pods increased.

Table 5.55: 25 Clients Average Response Times

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	Before Network Update 60 samples	After Network Update 60 samples
1	125	1028	229	223
2	123	984	222	206
3	124	1015	227	219
4	125	978	212	176
5	128	985	245	231
6	124	1074	238	234
7	120	948	220	194
8	132	962	213	187
9	130	949	211	165
10	128	948	227	219
11	119	923	240	222
12	123	946	211	185
13	123	971	225	177
14	125	972	244	220
15	124	964	232	204
16	121	919	244	242
17	129	926	247	225
18	139	962	206	182
19	129	1019	280	124
20	130	997	243	231
21	129	942	237	233
22	135	950	227	197
23	119	1030	228	198
24	129	1062	241	215
25	122	937	220	174
Average	126.04	968.48	227.92	203.16

Table 5.56: Part 1 - 50 Clients Average Response Times (1-25-clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
1	521	1425	813	1018
2	591	1437	706	942
3	538	1417	816	1040
4	517	1394	832	1026
5	593	1465	771	954
6	531	1463	780	992
7	554	1448	789	789
8	552	1465	749	964
9	564	1429	795	1003
10	500	1478	696	905
11	496	1486	800	991
12	554	1484	845	1037
13	561	1479	773	969
14	530	1452	769	980
15	581	1451	764	973
16	522	1492	817	1027
17	504	1458	774	987
18	508	1426	768	977
19	510	1467	793	1018
20	560	1455	809	1044
21	567	1448	853	960
22	540	1473	753	1011
23	556	1443	825	982
24	548	1402	772	1015
25	556	1482	814	954

Table 5.57: Part 2 - 50 Clients Average Response Times (26-50-clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
26	490	1398	744	1012
27	514	1460	821	1005
28	573	1462	817	998
29	556	1454	786	1007
30	554	1458	814	987
31	579	1446	769	1026
32	512	1459	820	1014
33	542	1463	819	1026
34	579	1484	833	894
35	543	1488	612	1024
36	556	1461	809	936
37	559	1456	725	1000
38	543	1421	788	854
39	562	1480	569	1003
40	551	1438	794	1054
41	504	1478	863	897
42	539	1461	640	976
43	538	1466	774	987
44	549	1463	779	1039
45	456	1472	837	1038
46	584	1420	843	1013
47	592	1375	811	965
48	508	1458	762	1059
49	523	1427	861	1025
50	493	1461	833	823
Average	507	1443	823	920.5

Table 5.58: Part 1 - 100 Clients Average Response Times (1-25-clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
1	6341	9449	7767	8319
2	6177	9520	7941	8412
3	6051	9512	7522	8084
4	6182	9706	7856	8369
5	6015	9661	7027	7691
6	6361	9302	7939	8469
7	6421	9484	7706	8283
8	6240	9446	7506	8056
9	6071	9709	7337	7929
10	6470	9695	7910	8411
11	6211	9513	7790	8301
12	6406	9678	6840	7500
13	6377	9513	7885	8390
14	5623	9561	7327	7936
15	6053	9663	7115	7745
16	6340	9520	7163	7800
17	6122	9708	7619	8179
18	6546	9377	7200	7777
19	5893	9650	7701	8294
20	6357	9228	7873	8385
21	6302	9576	7864	8408
22	6128	9272	7875	8388
23	6115	9524	8003	8477
24	6441	9572	7142	7757
25	5600	9556	7390	7981

Table 5.59: Part 2 - 100 Clients Average Response Times (26-50-clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
26	6191	9539	7727	8262
27	5868	9500	7788	8343
28	6217	9559	7721	8293
29	5904	9328	7827	8360
30	6090	9703	7656	8225
31	6036	9480	7781	8298
32	6213	9673	7506	8138
33	6528	9482	6851	7513
34	5478	9744	8113	8562
35	5652	9517	7127	7750
36	5994	9561	6810	7473
37	5930	9502	8005	8474
38	6228	9627	7030	7682
39	6398	9632	7443	8032
40	6353	9410	7969	8448
41	6012	9574	7672	8215
42	6067	9482	7875	8384
43	6427	9428	7015	7681
44	6312	9668	7506	8081
45	5753	9712	8079	8551
46	5935	9653	7782	8276
47	6244	9773	7643	8187
48	6505	9518	7536	8117
49	5645	9408	7333	7955
50	6203	9578	7923	8389

Table 5.60: Part 3 - 100 Clients Average Response Times (51-75 clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
51	6221	9553	7240	7859
52	6374	9464	7984	8460
53	6338	9545	7759	8285
54	6473	9566	7931	8467
55	5486	9523	7886	8380
56	5875	9450	7198	7834
57	6242	9675	7569	8129
58	5748	9416	7532	8112
59	5674	9566	7644	8227
60	5938	9457	7883	8422
61	6172	9464	7838	8351
61	5387	9602	7622	8216
63	6044	9418	6873	7534
64	5568	9461	7767	8302
65	6537	9567	7369	7987
66	5881	9285	7329	7928
67	6358	9530	7975	8479
68	5907	9479	7767	8273
69	6201	9291	7275	7883
70	6314	9605	7940	8420
71	6116	9426	7735	8282
72	6012	9356	7969	8461
73	6401	9487	7738	8283
74	6374	9419	7434	8133
75	5873	9239	7514	8091

Table 5.61: Part 4 - 100 Clients Average Response Times (76-100 clients)

	Average Client Response time (ms)			
Clients	Without attacker 120 samples	With Attacker 120 samples	With/out Attacker 120 samples	With/out Attacker 140 samples
76	6156	9599	7519	8111
77	6548	9506	7616	8202
78	6011	9251	7352	7948
79	6330	9518	7888	8422
80	5949	9558	7480	8080
81	6469	9689	7597	8155
82	5966	9518	7614	8191
83	6046	9484	7794	8323
84	5978	9430	7757	8283
85	6109	9483	7225	7869
86	6324	9665	7588	8175
87	6126	9699	7234	7901
88	6224	9518	7940	8438
89	6414	9510	7278	7867
90	5850	9337	7555	8138
91	5842	9277	7668	8232
92	6371	9543	7385	7977
93	5979	9670	7986	8497
94	6266	9659	7448	8031
95	6280	9570	7297	7886
96	6062	9519	7625	8175
97	6065	9745	7969	8459
98	5662	9692	7511	8087
99	5698	9498	7667	8227
100	6061	9490	7922	8434
Average	6201	9469.5	7844.5	8376.5

5.10 5-Endpoints Topology

In this experiment, an alternative network topology was employed, as illustrated in Figure 5.47. Unlike previous configurations, this topology involved multiplexing the pods at the endpoints. The primary focus of the analysis was to observe how client response times were affected by network updates.

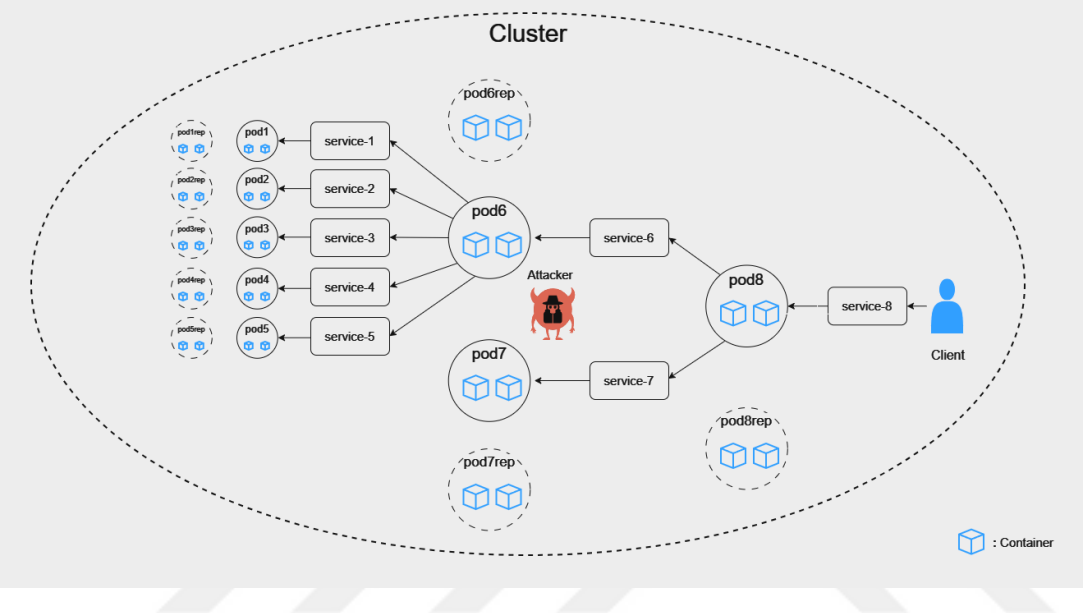


Figure 5.47: 5-Endpoints Network Topology

Regarding the network topology, a cluster consisting of 16 pods was established, comprising 8 primary pods within the main network and their respective replicas. The backup pods were positioned outside the main network and were governed by a specific network policy. In addition, a client pod named **user** was created, alongside an attacker pod labeled **attacker**. The configuration of all pods within the cluster, along with their IP addresses, is depicted in Figure 5.48.

In the context of the network's operational principle, as observed in the previous experiment, each pod appends its respective numerical identifier to the response values. Initially, the client sends a request to pod8, which subsequently forwards the request to pod6 and pod7. Pod6 then sends the request to pods 1, 2, 3, 4 and 5. After these pods process and aggregate the requests, the collective responses are returned to pod6. Pod6 adds 6 to the total sum of the received responses and responses this result back

```
yalginaythesis@cloudshell:~ (elated-life-425314-c0)$ kubectl get pods -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE
attacker   1/1     Running   0          16s   10.88.1.239   gke-cluster-1-default-pool-003683cd-p9ng
pod1       2/2     Running   0          15s   10.88.1.240   gke-cluster-1-default-pool-003683cd-p9ng
pod1rep    2/2     Running   0          15s   10.88.1.241   gke-cluster-1-default-pool-003683cd-p9ng
pod2       2/2     Running   0          15s   10.88.1.242   gke-cluster-1-default-pool-003683cd-p9ng
pod2rep    2/2     Running   0          15s   10.88.1.243   gke-cluster-1-default-pool-003683cd-p9ng
pod3       2/2     Running   0          14s   10.88.1.244   gke-cluster-1-default-pool-003683cd-p9ng
pod3rep    2/2     Running   0          14s   10.88.1.245   gke-cluster-1-default-pool-003683cd-p9ng
pod4       2/2     Running   0          14s   10.88.1.246   gke-cluster-1-default-pool-003683cd-p9ng
pod4rep    2/2     Running   0          14s   10.88.1.247   gke-cluster-1-default-pool-003683cd-p9ng
pod5       2/2     Running   0          13s   10.88.1.248   gke-cluster-1-default-pool-003683cd-p9ng
pod5rep    2/2     Running   0          13s   10.88.1.249   gke-cluster-1-default-pool-003683cd-p9ng
pod6       2/2     Running   0          13s   10.88.1.250   gke-cluster-1-default-pool-003683cd-p9ng
pod6rep    2/2     Running   0          13s   10.88.1.251   gke-cluster-1-default-pool-003683cd-p9ng
pod7       2/2     Running   0          12s   10.88.1.252   gke-cluster-1-default-pool-003683cd-p9ng
pod7rep    2/2     Running   0          12s   10.88.1.254   gke-cluster-1-default-pool-003683cd-p9ng
pod8       2/2     Running   0          12s   10.88.1.4     gke-cluster-1-default-pool-003683cd-p9ng
pod8rep    2/2     Running   0          12s   10.88.1.253   gke-cluster-1-default-pool-003683cd-p9ng
user       1/1     Running   0          10s   10.88.1.6     gke-cluster-1-default-pool-003683cd-p9ng
```

Figure 5.48: Pod IP's in Cluster

to pod8. Meanwhile, pod7 adds 7 to the request previously received from pod8 and then responds back to pod8. Pod8 further adds 8 to the accumulated responses and delivers the final result to the client. The client maintains continuous communication by periodically sending requests to pod8, incrementing the input by 1 each second. Communication is facilitated via HTTP POST requests. For instance, as explained in Equation 5.3, the response generated when the input value is 1 is computed theoretically through the operations outlined in the topology.

$$Responsevalue = (1+1)+(1+2)+(1+3)+(1+4)+(1+5)+6+(1+7)+8 = 42 \tag{5.3}$$

The theoretically calculated value was verified in Figure 5.49. The response value for the requested value of 1 was observed as 42. Additionally, in each request, the response was expected to increase by 1 due to the increase in input, but the request that reached 5 different endpoints increased by 5 more times. Therefore, the response value was expected to increase by 6 in each request. Based on this result, it was verified that the network was implemented as desired.

The analysis commenced with calculating the average response times of the pods using 50 communication samples under the influence of the attacker. The average response times for client pods 8 and 6 are presented in Table 5.62. As anticipated, the client response time for pod 8, which is the furthest from the endpoint, was the

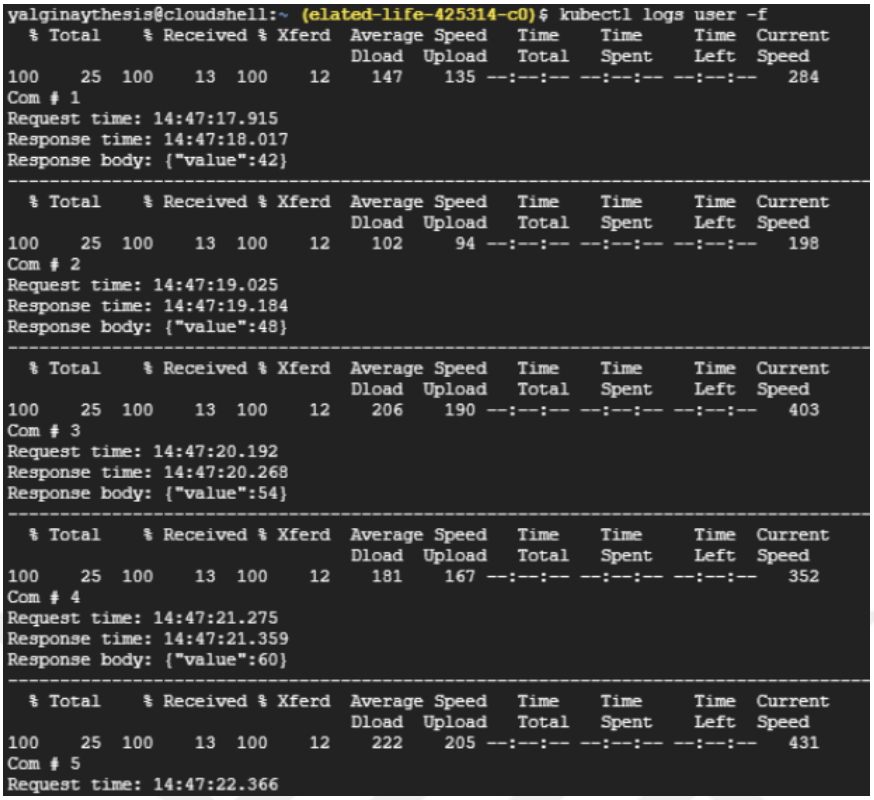


Figure 5.49: Client Responses

longest, whereas pod 6, being closest to the endpoint, exhibited the shortest response time.

Table 5.62: Client and Pods Response Duration

Pod Names	Response Duration(ms)
client	78.6
pod8	55.139
pod6	38.625

Examining Figure 5.50, it is evident that requests from IP addresses 10.88.1.239 (attacker) and 10.88.1.250 (pod6) are recorded in pod1's logs before the occurrence of the error. The attacker's communication period is 2 seconds, whereas the client's period is 1 second, resulting in a higher frequency of requests from pod6.

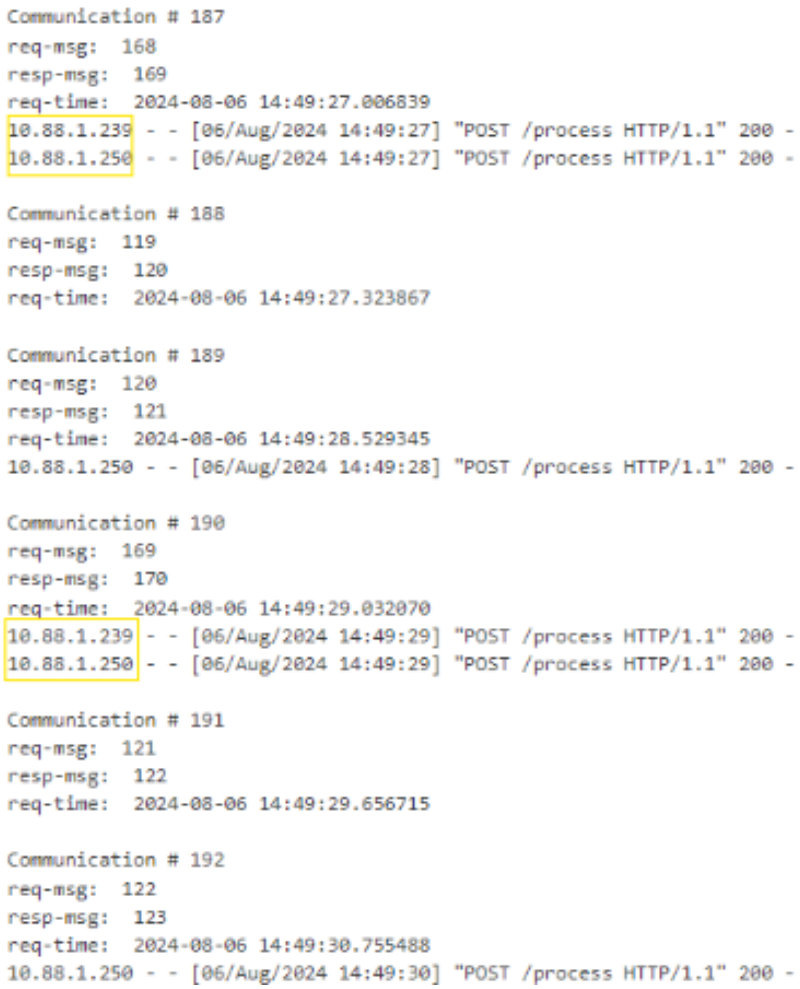


Figure 5.50: Pod1 Log

Figure 5.51 was used to determine the time when pod1rep was integrated into the network. Additionally, the final message from pod6 to pod1 was identified from pod1's logs. Based on these two timestamps, Table 5.63 was constructed. The client response times correspond to the pod1 and pod1rep logs were filled. The 3rd and 4th samples represent the last request received by pod1 from pod6 and the first request received by pod1rep from pod6, respectively. Subsequently, Table 5.64 was created by averaging these timestamps. The scenario in which the attacker targeted pod1 was repeated three times and the average of these three values was calculated. All values were then incorporated into the table.

During the update, communication maintained its regular period. A latency of 78.8 ms was measured during the update, compared to 78.6 ms before the update, indicat-



Figure 5.51: Pod1rep Log

ing no significant variation.

Subsequently, the error status of pod6 was analyzed. In this situation, the attacker pod attacked the pod6. Table 5.65 and Table 5.66 present the client response times during the network update, alongside the results from three separate experiments, including their respective averages.

The client response time exhibited a slight increase when pod6 failed, compared to the response time observed during a pod1 failure. Specifically, the response time increased by 78.8 ms, reaching 87.5 ms in this experiment. This increase is attributed to the fact that pod6 sends requests to five different pods. Notably, such an increase was not observed in prior experiments involving serial topologies, which suggests that this difference in topology is responsible for the observed increase.

Table 5.63: Client Pod Timestamps During Network Update(Pod1 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	14:47:28.810	14:47:28.874	64
2	14:47:29.880	14:47:29.948	68
3	14:47:30.955	14:47:31.034	79
4	14:47:32.040	14:47:32.122	82
5	14:47:33.133	14:47:33.233	100
6	14:47:34.240	14:47:34.314	74

Table 5.64: Average Client Response Time Analysis (Pod1 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	80.5	71.0	85.0	78.8

The failure of pod7 was subsequently analyzed. During this failure, the instances when requests from pod8 directed to pod7 were rerouted to pod7rep were documented, with observations recorded at approximately 15:27:19 and 15:27:20. The client log for these timestamps is illustrated in Figure 5.52.

To encompass the timestamps before, during and after the update, six communication samples were collected. These samples were used to populate Table 5.67 Following the repetition of these experiments, three distinct experimental results and their averages were compiled and are presented in Table 5.68.

The duration of client response, initially recorded at 78.6 milliseconds, was observed to be 73.0 milliseconds in the particular scenario. It indicates that the substitution of pod7 with pod7rep did not result in a significant change in response time.

Analysis of cases involving the failure of three distinct pods led to the following con-

Table 5.65: Client Pod Timestamps During Network Update (Pod6 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	15:16:45.438	15:16:45.516	78
2	15:16:46.522	15:16:46.598	76
3	15:16:47.605	15:16:47.683	78
4	15:16:48.691	15:16:48.774	83
5	15:16:49.781	15:16:49.854	73
6	15:16:50.861	15:16:50.928	67

Table 5.66: Average Client Response Time Analysis (Pod6 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	80.5	92.0	90.0	87.5

clusion: When a pod directs requests to multiple other pods, a minor increase in client response time may be detected during network updates. Conversely, if the failing pod only communicates with a single pod or serves as an endpoint, no significant variation in client response time is observed during network updates.

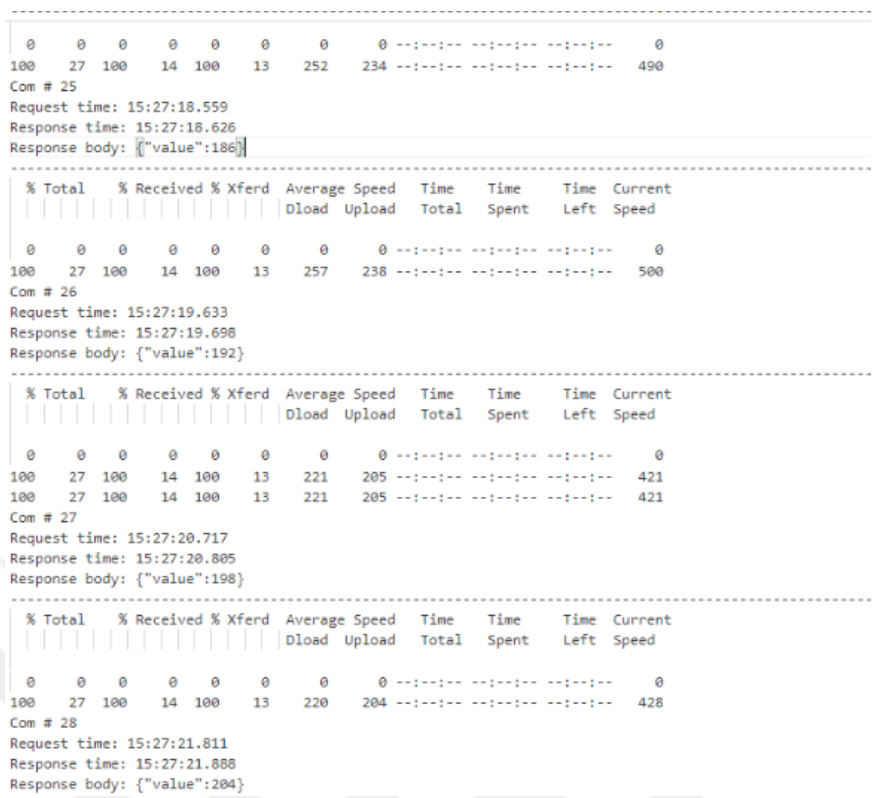


Figure 5.52: Pod1 Log

Table 5.67: Client Pod Timestamps During Network Update(Pod7 fail case)

	Request Time (hh:mm:ss.ms)	Response Time (hh:mm:ss.ms)	Response Duration (ms)
1	15:27:17.470	15:27:17.551	81
2	15:27:18.559	15:27:18.626	67
3	15:27:19.633	15:27:19.698	68
4	15:27:20.717	15:27:20.805	88
5	15:27:21.811	15:27:21.888	77
6	15:27:22.898	15:27:23.059	61

Table 5.68: Average Client Response Time Analysis (Pod7 fail case)

	Number of Iterations			
	1	2	3	Average
Client Response Duration (ms)	78.0	68.5	72.5	73.0

CHAPTER 6

CONCLUSION

This thesis introduced a novel architecture for intrusion prevention in SDCNs with several key contributions. The architecture is modular, incorporating pod labeling, sidecar containers for traffic capture (tcpdumper), a fetch log module and a HM module, which together enable automated attack detection and mitigation with minimal network performance degradation. Significant performance improvements were achieved by optimizing the fetch log module, reducing processing time by up to 50% for large-scale pod deployments. The impact of the tcpdumper container on node performance was thoroughly assessed, revealing certain operational limitations. Additionally, a comprehensive performance evaluation of the HM module in relation to container usage was conducted. Extensive experimentation demonstrated the architecture's ability to effectively mitigate attacks across various network topologies, including configurations of 3, 10 and 7 pods as well as 5-endpoint setups while maintaining network functionality with minimal impact on client response times. Scalability tests, involving up to 100 clients, showed the system's capacity to handle increased loads, with response time reductions of up to 52% observed in high-traffic attack mitigation scenarios. Additionally, the HM module was refined to focus on role labels, improving network update efficiency and reducing unnecessary processing, resulting in faster attack mitigation in large pod environments. The architecture demonstrated resilience in various pod failure scenarios, maintaining consistent performance regardless of the pod targeted.

In conclusion, the proposed architecture offers a significant advancement in securing SDCNs by delivering automated, efficient, and scalable intrusion prevention with minimal impact on legitimate network traffic. The model was validated to intro-

duce minimal latency and impose negligible overhead, while also proving effective in large-scale network environments. Additionally, collecting network traffic data directly from containers is more cost-efficient compared to gathering data from network switches, and the model is easier to implement. Given the widespread use of Kubernetes and GKE, this work can be readily adopted by others within the community.

The proposed architecture lays a solid foundation for advancing intrusion prevention mechanisms in container networks. First of all, future research should focus on developing IDS that leverage ML and DL techniques to enhance detection accuracy with minimal training data while also incorporating feedback from incident response teams to adapt more effectively to new attack methods. Secondly, optimizing the HM module is crucial for minimizing latency in complex network topologies, alongside broadening the focus beyond DoS attacks to strengthen cloud-native integration for improved overall performance. Thirdly, to better replicate real-world applications, future implementations should employ more sophisticated containers. Additionally, identifying critical pods within the network and replicating only those will enhance memory utilization and ensure more efficient resource allocation. Also, extensive resilience testing, including scenarios involving multiple or cascading attacks is essential to assess the robustness of the system. Moreover, addressing privacy concerns in log collection through privacy-preserving methods and conducting real-world deployment studies will offer practical validation. Addressing these key areas will allow the architecture to evolve into a more secure, efficient, and resilient solution for SDN-based container networks within cloud environments.

REFERENCES

[1] I. Aktolga, E. S. Kuru, Y. Sever, and P. Angin, "Ai-driven container security approaches for 5g and beyond: A survey," *ITU Journal on Future and Evolving Technologies*, vol. 4, pp. 364–382, 06 2023.

[2] C. Gupta, "Honeykube: designing a honeypot using microservices-based architecture," Master's thesis, University of Twente, 2021.

[3] C. de Weever and M. Andreou, "Zero trust network security model in containerized environments," *University of Amsterdam: Amsterdam, The Netherlands*, 2020.

[4] İ. T. Aktolga, "A study on analysis and detection of container escape vulnerabilities in docker," Master's thesis, Middle East Technical University (Turkey), 2024.

[5] M. Aly Amin, A. Harun Dogan, E. Sena Kuru, Y. Sever, and P. Angin, "Misuse detection and response for orchestrated microservices based software," in *International Conference on Advanced Information Networking and Applications*, pp. 217–226, Springer, 2024.

[6] Y. Sever, G. Ekinci, A. H. Dogan, B. Alparslan, A. S. Gurbuz, V. Jabrayilov, and P. Angin, "An empirical analysis of ids approaches in container security," in *2022 International Workshop on Secure and Reliable Microservices and Containers (SRMC)*, pp. 18–26, IEEE, 2022.

[7] Y. Sever and A. H. Dogan, "A kubernetes dataset for misuse detection," 2023.

[8] A. Demirpolat, A. K. Sarica, and P. Angin, "Protédge: a few-shot ensemble learning approach to software-defined networking-assisted edge security," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 6, p. e4138, 2021.

[9] Y. Wang, X. Chen, Q. Wang, R. Yang, and B. Xin, "Unsupervised anomaly detection for container cloud via bilstm-based variational auto-encoder," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3024–3028, IEEE, 2022.

[10] T. P. Ravi Kumar, "Securing communication and identifying threats in cloud microservices," 2023.

[11] M. Soualhia and F. Wuhib, "Automated traces-based anomaly detection and root cause analysis in cloud platforms," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 253–260, IEEE, 2022.

[12] Y. Wang, Q. Wang, X. Chen, D. Chen, X. Fang, M. Yin, and N. Zhang, "Containerguard: A real-time attack detection system in container-based big data platform," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 5, pp. 3327–3336, 2020.

[13] H. Gantikow, T. Zöhner, and C. Reich, "Container anomaly detection using neural networks analyzing system calls," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 408–412, IEEE, 2020.

[14] Z. Zou, Y. Xie, K. Huang, G. Xu, D. Feng, and D. Long, "A docker container anomaly monitoring system based on optimized isolation forest," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 134–145, 2019.

[15] P. V. Mohan, S. Dixit, A. Gyaneshwar, U. Chadha, K. Srinivasan, and J. T. Seo, "Leveraging computational intelligence techniques for defensive deception: a review, recent advances, open problems and future directions," *Sensors*, vol. 22, no. 6, p. 2194, 2022.

[16] N. Aslam, S. Srivastava, and M. Gore, "A comprehensive analysis of machine learning-and deep learning-based solutions for ddos attack detection in sdn," *Arabian Journal for Science and Engineering*, vol. 49, no. 3, pp. 3533–3573, 2024.

[17] A. V. Songa and G. R. Karri, "An integrated sdn framework for early detection of

ddos attacks in cloud computing," *Journal of Cloud Computing*, vol. 13, no. 1, p. 64, 2024.

[18] P. Rengaraju, V. R. Ramanan, and C.-H. Lung, "Detection and prevention of dos attacks in software-defined cloud networks," in *2017 IEEE Conference on Dependable and Secure Computing*, pp. 217–223, IEEE, 2017.

[19] K. Bhushan and B. B. Gupta, "Distributed denial of service (ddos) attack mitigation in software defined network (sdn)-based cloud computing environment," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, pp. 1985–1997, 2019.

[20] T. V. Phan and M. Park, "Efficient distributed denial-of-service attack defense in sdn-based cloud," *IEEE Access*, vol. 7, pp. 18701–18714, 2019.

[21] S. E. V. S. Pillai and K. Polimetla, "Mitigating ddos attacks using sdn-based network security measures," in *2024 International Conference on Integrated Circuits and Communication Systems (ICICACS)*, pp. 1–7, IEEE, 2024.

[22] D. Yin, L. Zhang, and K. Yang, "A ddos attack detection and mitigation with software-defined internet of things framework," *IEEE Access*, vol. 6, pp. 24694–24705, 2018.

[23] A. Bose, G. S. Aujla, M. Singh, N. Kumar, and H. Cao, "Blockchain as a service for software defined networks: A denial of service attack perspective," in *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pp. 901–906, IEEE, 2019.

[24] S. Pisharody, J. Natarajan, A. Chowdhary, A. Alshalan, and D. Huang, "Brew: A security policy analysis framework for distributed sdn-based cloud environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 1011–1025, 2017.

[25] M. E. Ahmed and H. Kim, "Ddos attack mitigation in internet of things using software defined networking," in *2017 IEEE Third International Conference on*

Big Data Computing Service and Applications (BigDataService), pp. 271–276, IEEE, 2017.

[26] P. Bull, R. Austin, E. Popov, M. Sharma, and R. Watson, "Flow based security for iot devices using an sdn gateway," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 157–163, IEEE, 2016.

[27] P. Zhang, F. Liu, N. Kumar, and G. S. Aujla, "Information classification strategy for blockchain-based secure sdn in iot scenario," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1081–1086, IEEE, 2020.

[28] N. Ravi and S. M. Shalinie, "Learning-driven detection and mitigation of ddos attack in iot via sdn-cloud architecture," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3559–3570, 2020.

[29] D. Gao, Z. Liu, Y. Liu, C. H. Foh, T. Zhi, and H.-C. Chao, "Defending against packet-in messages flooding attack under sdn context," *Soft Computing*, vol. 22, pp. 6797–6809, 2018.

[30] A. Chikhale and R. Khondoker, "Security analysis of sdn cloud applications," *SDN and NFV Security: Security Analysis of Software-Defined Networking and Network Function Virtualization*, pp. 19–38, 2018.

[31] A. Derhab, M. Guerroumi, A. Gumaei, L. Maglaras, M. A. Ferrag, M. Mukherjee, and F. A. Khan, "Blockchain and random subspace learning-based ids for sdn-enabled industrial iot security," *Sensors*, vol. 19, no. 14, p. 3119, 2019.

[32] J. Galeano-Brajones, J. Carmona-Murillo, J. F. Valenzuela-Valdés, and F. Luna-Valero, "Detection and mitigation of dos and ddos attacks in iot-based stateful sdn: An experimental approach," *Sensors*, vol. 20, no. 3, p. 816, 2020.

[33] L. Golightly, P. Modesti, R. Garcia, and V. Chang, "Securing distributed systems: A survey on access control techniques for cloud, blockchain, iot and sdn," *Cyber Security and Applications*, vol. 1, p. 100015, 2023.

[34] M. S. Salek, S. M. Khan, M. Rahman, H.-W. Deng, M. Islam, Z. Khan, M. Chowdhury, and M. Shue, "A review on cybersecurity of cloud computing

for supporting connected vehicle applications," *IEEE Internet of Things Journal*, vol. 9, no. 11, pp. 8250–8268, 2022.

[35] R. Zhang, C. Xu, and M. Xie, "Powering hands-on cybersecurity practices with cloud computing," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pp. 1–2, IEEE, 2019.

[36] J. Smith-perrone and J. Sims, "Securing cloud, sdn and large data network environments from emerging ddos attacks," in *2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*, pp. 466–469, IEEE, 2017.

[37] D. Javeed, T. Gao, and M. T. Khan, "Sdn-enabled hybrid dl-driven framework for the detection of emerging cyber threats in iot," *Electronics*, vol. 10, no. 8, p. 918, 2021.

[38] "Docker: Enterprise container platform." https://www.docker.com. Accessed: 2024-09-19.

[39] "Containers vs. virtual machines." https://cloud.google.com/discover/containers-vs-vms/. Accessed: 2024-09-19.

[40] "Software defined network." cloud.google.com/kubernetes-engine/docs/concepts/network-overview. Accessed: 2024-09-19.

[41] "Google kubernetes engine." https://cloud.google.com/kubernetes-engine/. Accessed: 2024-09-19.

[42] "Calico." https://cloud.google.com/blog/products/gcp/network-policy-support-for-kubernetes-with-calico/. Accessed: 2024-09-19.

[43] "Tcpdump." https://www.tcpdump.org/. Accessed: 2024-09-19.

[44] P. L. N. C. Josyula, "Schema sentinels: Empowering microservices with sidecar containers for centralized configuration validation,"

[45] "Nat." https://en.wikipedia.org/wiki/Network_address_translation. Accessed: 2024-09-22.

[46] “Nat.” https://kubernetes.io/docs/reference/kubectl/generated/kubectl_top/. Accessed: 2024-10-15.

[47] “Nat.” https://www.avg.com/en/signal/fix-high-cpu-usage. Accessed: 2024-10-15.