POLITECNICO DI TORINO

Master's Degree in ICT FOR SMART SOCIETIES

Master's Degree Thesis

Honeypot in a box: A distributed cluster network for honeypot deployment

Supervisors

Prof. Marco MELLIA

Prof. Idilio DRAGO

Candidate

Alejandro AYALA GIL

April 2024

Abstract

Honeypots are strategic tools crafted to divert potential attackers away from compromising infrastructures while simultaneously capturing their attack techniques. These sophisticated cybersecurity instruments empower experts to discern patterns that could present risks to specific infrastructures. Deploying a honeypot in a particular location may result in the repetitive collection of similar patterns. Establishing an infrastructure that enables the distribution of honeypots across diverse locations could yield distinct patterns. Despite this potential advantage, there is presently a lack of dedicated tools designed for such purposes. In cooperation with various entities, I aspire to establish a distributed network of honeypots for comprehensive research endeavors.

To initiate an in-depth examination of the advantages of containers and virtual machines, ultimately necessitating the adoption of a container orchestrator. This exploration involved a detailed comparative analysis, assessing Docker Compose, Swarm, and Kubernetes, with the latter emerging as the preferred solution due to its unparalleled scalability. To enhance the robustness of secure connections between nodes, an exhaustive exploration of VPN technologies, including OpenVPN, IPsec, and WireGuard, was undertaken. The latter was chosen for its outstanding throughput performance, solidifying its selection in the network architecture. In the quest for an optimal Kubernetes distribution, a thorough evaluation covered K8s, Minikube, Rancher, K3s, and K0s. The choice of K3s stemmed from its simplicity and robust support for edge devices, including Raspberry Pis.

Consequently, I delve into the implementation of scripts designed to facilitate the seamless installation of a cluster and the establishment of node connections through a VPN. This installation ensures the creation of a robust system that can withstand disruptions, promptly initiating recovery mechanisms in the event of a cluster node failure. Once the cluster is operational, specific manifests containing the Cowrie honeypot image are applied, allowing me to deploy these honeypots across diverse networks. Leveraging services, I enable the exposure of these honeypots in various locations, ultimately achieving our objective of distributing honeypots across different environments.

Upon establishing the K3s cluster, it becomes imperative to conduct thorough performance assessments. The benchmarks employed to evaluate the cluster encompass a spectrum of critical metrics. These include Network Latency Testing, Pod Deployment Time, Honeypot Simulation, Network Throughput, and Node Failure and Recovery. These benchmarks collectively provide comprehensive insights into the efficiency and resilience of the k3s cluster under varied conditions.

In the future, the project envisions the incorporation of monitoring tools, an expansion in the number of honeypots, and the development of intelligent mechanisms to enhance honeypot control. This forward-looking strategy aims to enhance the cluster's overall functionality and security. These planned initiatives aim to create a more sophisticated and responsive infrastructure, paving the way for continual improvements in the project's capabilities.

Acknowledgements

Esta experiencia de venir a Italia a hacer una doble titulación ha estado una experiencia inolvidable. Nadie sabe lo que es estar lejos de casa, de tu familia, de tus amigos hasta que se vive una experiencia como esta. Es una experiencia única en la que se pude decir siempre hay una primera vez para todo. Una primera vez buscando casa, primera vez haciendo mercado solo, primera vez pagando facturas de servicios, primera vez teniendo clases en inglés, primera vez comunicándote constantemente con otros en una lengua que no es la tuya. No ha sido un camino fácil, pero en medio puedo decir que hay muchas personas que debo agradecer por impulsarme a conseguir este logro.

Agradezco a Dios por haberme permitido llegar tan lejos en la vida, por darme la oportunidad de vivir experiencias en la vida que muy pocas personas han podido tener.

A mis abuelos y mi primo Juan Camilo que desde el cielo siempre están pendientes de mí y me aman con todo su corazón.

A mis padres, mi hermana quienes han luchado incontables días por verme crecer de manera profesional, y por siempre apoyarme desde lo lejos.

A mi familia por siempre estar pendiente de mí y siempre alentarme a seguir adelante con fuerza para lograr todos mis sueños.

A los amigos que deje en Colombia que también me apoyaron en este proceso, que siempre han estado pendientes de mí.

A David Contreras uno de mis mejores amigos que me ha impulsado en mi vida personal y profesional. A quien agradezco por siempre haber luchado conmigo tantas batallas durante la carrera.

A los amigos que hice aquí en Italia que me han hecho sentir siempre en casa, recordándome siempre lo orgulloso que me debo sentir de ser colombiano. En especial agradezco a Daniel Ballestero por a pesar de no conocerme, ofrecerme un lugar donde vivir en mis primeros días aquí.

A mis roomates que he tenido, prácticamente han sido hermanos para mi siempre cuidando de mí, siempre compartiendo juntos experiencias inolvidables en la casa.
A mi pareja que ha sido un gran apoyo para mi y siempre me llena de mucha felicidad compartir con ella momentos lindos.

A mi pareja que ha sido un gran apoyo para mi y siempre me llena de mucha felicidad compartir con ella momentos lindos.

A Idilio Drago e Rodolfo Vieira por me orientarem, apoiarem e estarem sempre atentos no desenvolvimento desta tese. Sou sempre grato a vocês.

A Marco Mellia per avermi dato l'opportunità di svolgere questa tesi. Per essere stato sempre attento a me durante tutto questo processo, guidandomi in ogni fase che ho sviluppato.

To the guys from the smart data center with whom I always shared fun experiences during the development of my thesis.

A todos mis compañeros de maestría que fueron parte de este proceso de casi 2 años y medio.

¡Desde el fondo de mi corazón, muchas gracias a todos!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cybersecurity is in constant growth concerning businesses worldwide. With the pervasive integration of digital infrastructure into various facets of operations, the threat landscape has expanded exponentially[1].

Consequently, organizations are investing significant resources to fortify their defenses against cyber threats. The urgency of this investment underscores the significance, as evidenced by alarming statistics, such as IBM's 2023 report revealing that the global average cost of a data breach surged to $4.45 million, marking a 15% increase over three years. Organizations leveraging security AI and automation demonstrate substantial savings, highlighting the efficacy of proactive cybersecurity measures [2].

Moreover, the proliferation of internet connectivity further amplifies the susceptibility of systems to potential threats. Projections indicate that by 2023, nearly two-thirds of the global population will have internet access, with an estimated 5.3 billion users worldwide. In such a hyperconnected environment, the likelihood of cyberattacks is virtually inevitable. Consequently, the focus shifts from whether an organization will experience an attack to how well-prepared they are to mitigate and respond to such threats[3].

One innovative approach that has emerged to bolster cybersecurity defenses is the utilization of honeypots. Honeypots strategically position decoy systems within an infrastructure to divert and capture potential attackers, allowing organizations to gather valuable insights into their adversary's tactics and techniques. The distribution of honeypots across diverse geographical locations enables cybersecurity experts to discern new patterns based on geographic factors.

The research presented start from a project undertaken by the Smart Data division of Politecnico aimed at developing a distributed honeypot system. This system seeks to proliferate honeypots across diverse locations, facilitating the collection of varied and contextually relevant data for the analysis of attack strategies. However, the endeavor poses multifaceted challenges, encompassing technological complexities and the need for collaboration with third-party entities.

This thesis address these challenges by delving into the development of a distributed honeypot infrastructure, laying the groundwork for accommodating the requirements of future stakeholders. The research will encompass a comparative analysis of technologies pertinent to deploying such a system, culminating in the realization of a prototype capable of distributing honeypots and capturing data.

Subsequently, I will conduct a comprehensive benchmarking exercise to evaluate the performance and efficacy of the deployed infrastructure. In particular, with the experiments conducted, I reached average values of 17 ms and 95 Mbits/s.

In essence, this thesis is a pioneering endeavor to advance the capabilities of cybersecurity defense mechanisms through the strategic deployment of distributed honeypots. By elucidating the intricacies of implementation and performance assessment, this research aims to contribute significantly to the collective efforts aimed at fortifying digital ecosystems against evolving cyber threats.

Chapter 2

Objectives and Related Work

2.1 Objectives

The goal is to distribute honeypots to analyze patterns that may emerge in different scenes. However, the project goes beyond the concept of distributed honeypots. It encompasses various objectives that pose challenges when tackled individually. The supplementary goals are outlined below:

- Collect relevant data
- Store non-classified data
- Enhance honeypots to withstand severe attacks
- Develop a monitoring interface
- Ensure security for entities
- Control honeypot behavior during attacks
- Train models based on collected data
- Enable deployment on devices with limited resources

This thesis presents the implementation of a prototype that serves as the foundation for achieving the mentioned objectives. At the same time, some of these goals are addressed in this thesis, such as **storing data without considering relevancy or confidentiality, ensuring security for entities, and enabling deployment on devices with limited resources.**

2.2 Related Work

In my initial exploration for this thesis, I delved into the intricacies of a project known as T-Pot[4], developed by Telekom T-Pot is an all-encompassing multi-honeypot platform. This innovative system coordinates multiple honeypots using containers, complemented by monitoring tools for seamless data visualization.

For optimal performance, T-Pot mandates a device with a robust configuration, necessitating a minimum of 8-16GB RAM and 128GB SSD. This platform does not cater to devices with limited capabilities.

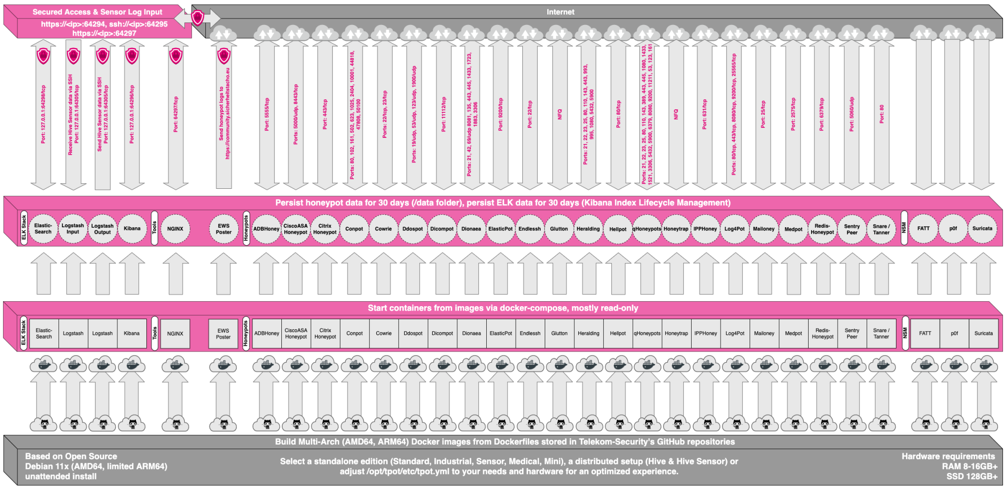


Figure 2.1: T-pot Architecture.
Source: Architecture Server and Agents. [4]

Figure 2.1 illustrates the architecture of TPOT. Docker containers form the backbone of this infrastructure, housing a variety of honeypots that expose specific ports to the internet. Additionally, some containers host monitoring tools and Elastic Search for effective data visualization, fortified with security measures.

One distinctive feature of T-Pot is its data collection mechanism. The installed T-Pot image transmits the gathered data to a centralized repository managed by Telekom. Note that if an individual opts to install multiple T-Pot ISOs, there is no streamlined approach to consolidate information from these instances.

A downside aspect of T-Pot's functionality revolves around using Docker Compose. This tool is instrumental in deploying the containers; however, it lacks resilience when containers go offline due to low resources or system issues, as they do not automatically restart. This characteristic merits careful consideration for maintaining uninterrupted functionality.

In my pursuit, T-Pot is a starting point showing the power of containers and honeypots. My project aims to transcend its current limitations:

- Container reliability
- Contemplation of low-resource devices

While T-Pot is proficient in certain aspects, several features essential to my endeavor are not within its framework. Nevertheless, I recognize the value of the tools employed in T-Pot because I plan to incorporate some of them into my project.

Chapter 3

Background

3.1 Virtualization

Virtualization emerges as a foundational concept, constituting one of the principal pillars. In this thesis, I confront two primary technologies: virtual machines and containers. Both of these technologies serve the purpose of resource encapsulation, offering an optimal means to establish a stable foundation for tasks involving the deployment of honeypots. Additionally, the functionality of snapshots and rollback features provide the capability to recover information from previous moments before system corruption occurs.

Virtual machines are computational entities that replicate an operating system environment, eliminating the need for a physical computer to execute programs and deploy applications[5]. It enables users to host a virtualized instance of one operating system, such as having a Windows 11 operating system and running Ubuntu 20.04. Well-known tools like VirtualBox and VMware facilitate the creation and management of virtual machines. Also, virtual machines can seamlessly integrate into a cloud provider's infrastructure. Moreover, virtual machines offer a layer of isolation from actual devices, adding an extra hurdle for potential hackers.

Containers are encapsulated units comprising an application along with its dependencies. Unlike virtual machines, containers leverage the host operating system's kernel, making them lightweight and efficient[6]. Essentially, containers operate as background processes, dynamically utilizing resources without requiring predefined allocations. This attribute allows numerous honeypots or applications to operate concurrently without imposing excessive resource demands on the host machines.

Delving deeper into the distinction between containers and virtual machines, Figure 3.1 illustrates their structures. One notable difference lies in the number of applications a container can execute with identical resources compared with a

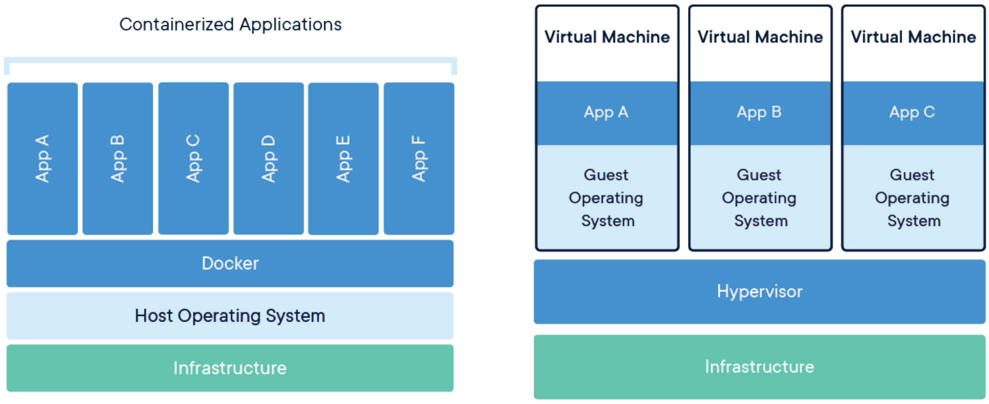


Figure 3.1: Containers vs Virtual Machines.
Source: Comparing Containers and Virtual Machines. [6]

virtual machine. Containers employ images to specify the required packages and resources for an application. Inside Docker Hub[7], we can encounter millions of images developed and tested. One particular example is the Cowrie[8] honeypot, constructed using Python. In a future chapter, I will describe how these technologies play a role in the implementation of the prototype.

3.2 Containers Orchestration

In the preceding section, I highlighted the advantages of utilizing containers. However, deploying containers necessitates the use of specific tools. Docker orchestrators come into play, enabling the simultaneous deployment of multiple containers.

Among these orchestrators, Docker Compose is one of the most popular and the default choice with Docker. Docker Compose simplifies the deployment of multi-container applications on Docker. It operates by interpreting a YAML file, adhering to the Compose file format, where users define the configuration of the application developers wish to deploy[9]. This file outlines the specifications for one or more containers, streamlining the setup and management of complex application environments. Applications like T-Pot leverage Docker Compose to deploy various components, including honeypots and monitoring tools. However, it lacks a mechanism to restore containers that may have been compromised or pose malicious intent.

Docker Swarm, a docker orchestrator developed by Docker, features an architecture comprising managers and worker nodes. Docker Swarm has a straightforward installation, is lightweight, and seamlessly integrates into the Docker environment [10]. It offers built-in load-balancing capabilities and employs an intelligent node selection process for container deployment. Nevertheless, It has limitations in scaling for larger applications, less community support, and the absence of cloud provider integration. Figure 3.2 describes how the architecture is composed in detail.

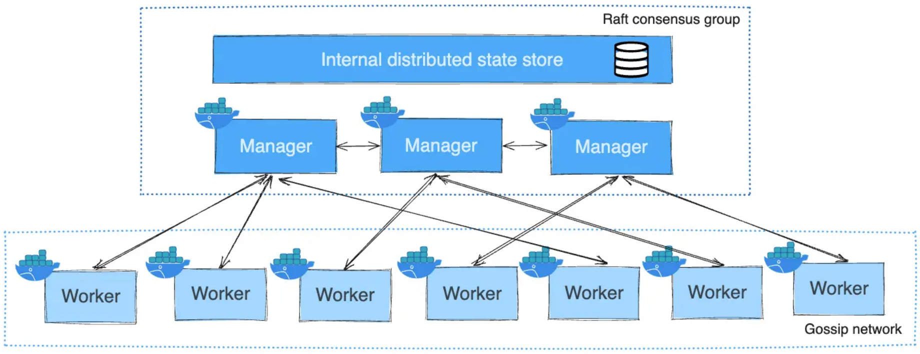


Figure 3.2: Docker Swarm Architecture.
Source: How nodes work. [11]

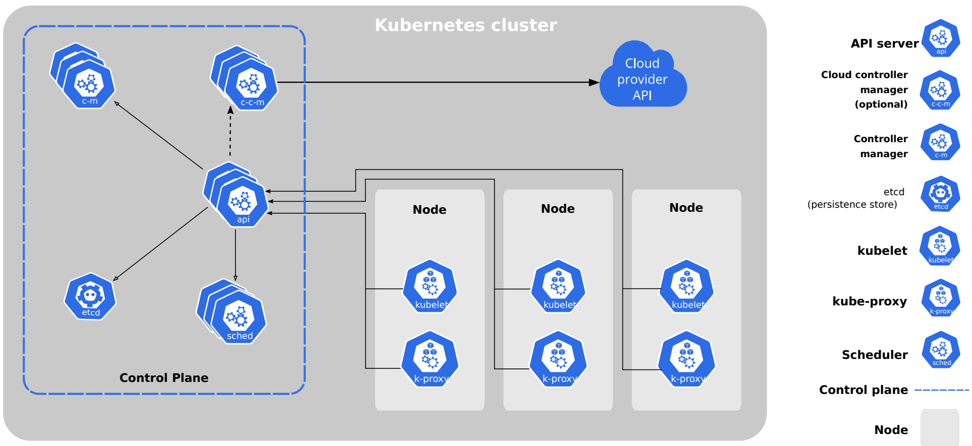


Figure 3.3: Kubernetes Architecture based on components.
Source: Kubernetes Components. [12]

Kubernetes, designed by Google, was born as an alternative to the docker orchestrators. Its expansive ecosystem, widespread adoption, and robust support from cloud providers distinguish Kubernetes as a compelling option. A downside is the learning curve associated with Kubernetes can be challenging for newcomers due to its complexity and the array of concepts and elements[13]. Kubernetes presents a modular and intricate architecture comprising various essential components, including the API server, etcd (a key-value store), scheduler, controller manager, and more[14], in Figure 3.3, you can observe the different components inside the architecture. A central master node empowers you to control the infrastructure using kubectl. With kubectl, you can seamlessly deploy applications, monitor, manage cluster resources, and review logs.

3.3 Network Security

When discussing the distribution of honeypots, it's essential to acknowledge that information exchange over a network is inherent. Additionally, to minimize the potential vulnerability to malicious attacks during this communication process, the thesis proposes implementing a Virtual Private Network (VPN).

A VPN operates by rerouting your internet traffic through a distant server while encrypting it. Ordinarily, when you attempt to access a website, your internet service provider (ISP) receives the request and guides you to your desired destination. However, upon connecting to a VPN, your internet traffic is directed through a remote server before reaching its intended destination.

The encryption established by a VPN shields your data from potential eavesdroppers, significantly bolstering your online security and reducing your digital footprint. As a result, your ISP cannot compile and sell your browsing history to third parties.

Furthermore, a VPN masks your IP address, replacing it with one belonging to the VPN server you are using. It adds an extra layer of security and enhances your online anonymity by concealing your geographical location. As a result, your browsing activities remain undisclosed, ensuring heightened privacy without revealing your city or country of origin.

Considering various open-source VPNs, I came across a comprehensive comparison provided by IVPN [15]. IKEv2 is an excellent choice due to its speed, security, and reliability. Notably, unlike OpenVPN, IKEv2 typically doesn't require additional software installation, making it the quickest to set up in most cases.

However, if your threat model involves sophisticated adversaries, it's worth considering OpenVPN due to concerns raised in leaked NSA presentations. OpenVPN remains an excellent choice across all platforms, boasting remarkable speed, security, and reliability.

Another noteworthy option is WireGuard, which excels particularly in high-speed scenarios. Promising enhanced security and faster speeds compared to existing solutions, WireGuard has gained traction since its integration into the Linux Kernel (v5.6) and subsequent release of v1.0. Given these advancements, WireGuard is now considered suitable for widespread use.

Chapter 4

Methodology

In this chapter, I will delineate the methodology employed in my thesis. Herein lies a comprehensive exposition of the decision-making processes I engaged in, elucidating how each chosen technology contributes to a prototype aligned with the objectives I have set forth.

4.1 System Architecture

Now that I have identified and selected the technologies to achieve my objectives, I will describe the system I intend to implement. Figure 4.1 illustrates the architecture, comprising three layers: physical, cluster, and service. Each layer encompasses various components, which I will elaborate on in the following chapter.

Firstly, at the physical layer, we have the foundational elements mentioned earlier in this chapter, including all virtualization setups and hosts involved in the system, interconnected via the internet. Specifically, I have three virtual machines: two running on my local computer and one operating within the Politecnico cluster.

Moving on to the cluster layer consists of the nodes constituting the cluster itself. Here, two key technologies, namely K3s and WireGuard, play crucial roles. Initially, I establish a VPN connection and then integrate each node into the cluster. Each worker node manages distinct workloads and may assume specific roles within the cluster. Furthermore, within the prototype's construction, one node will function as the master, while the remaining nodes will operate as workers. To streamline this process, I will develop scripts facilitating seamless peer-to-peer interactions.

Finally, the service layer focuses on tasks such as distributing honeypots, acquiring images, allocating storage memory, managing node loads, and directing traffic within the cluster. Specifically, I plan to deploy two Cowrie honeypots: one within a virtual machine on my local system, exposed within the same environment, and another on a virtual machine at Politecnico, exposed on a separate virtual machine

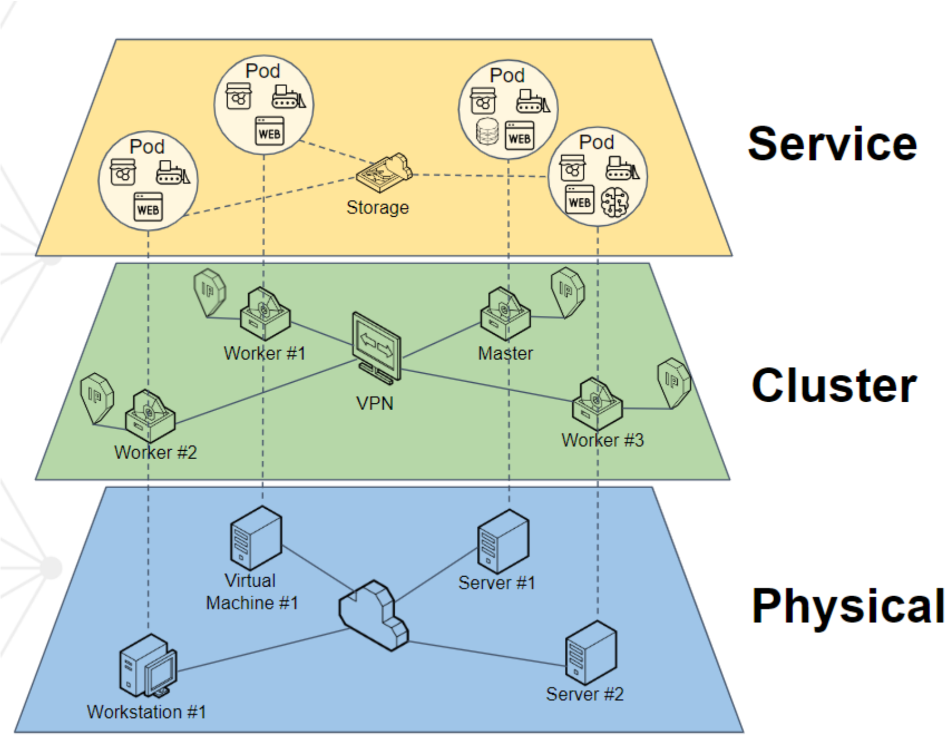


Figure 4.1: System architecture composed of three different layers.

on my computer. The honeypots will store the collected data on the hosting node disk using volumes.

Upon completing the prototype, it becomes imperative to conduct benchmarks to assess the system's efficiency across various metrics. I will elaborate on the testing procedures and present the results in another chapter.

4.2 Virtual Machines

In the preceding chapter, I delved into the concept of virtualization, elucidating the distinctions between virtual machines and containers. In the initial stages of my thesis, I leveraged VirtualBox as a testing platform for related work, including TPOT. Given my aim to deploy honeypots across diverse locations, I found it expedient to establish separate playgrounds in different geographical areas. Fortunately, the Smart Data Center at Politecnico generously provided me access to a virtual machine within their cluster, while I also utilized my personal computer for experimentation. Given that the tools essential to my thesis operate within a Linux kernel, whereas my primary operating system is Windows, I needed to use virtual machines. Moreover, the inherent advantages of resource management and snapshot functionality significantly facilitated my exploration.

At the start, I encountered a significant challenge regarding bridge adapter networking in VirtualBox. When connecting to a corporate internet network, a mandatory authentication step necessitates each physical machine to possess a unique IP address. Consequently, I explored two potential solutions to address this issue. The first involves utilizing the NAT network, wherein the virtual machine shares the same network interface as the physical host. Alternatively, I considered connecting to my cellular network, where firewall restrictions are absent. The second alternative emerged as my preferred choice because it enables me to simulate real-world scenarios more accurately. By connecting to my cellular network, the virtual machines emulate distinct hosts rather than simply a computer sending traffic to itself with the same IP address.

The virtual machines typically used for my experiments were configured with one core, 1024 MB of RAM, and shared 10GB of disk space with my physical host. They ran Ubuntu Server 22.04 as the operating system. Opting for Ubuntu Server, which lacks a graphical interface, ensured lightweight functionality while encompassing all the essential features necessary for my thesis work.

4.3 Containers Orchestration

Early in my thesis, I considered leveraging containers for deploying honeypots. Containers offer significant advantages, notably their minimal resource requirements compared to virtual machines. However, I grappled with uncertainties regarding how to manage multiple containers reliably.

In my search for a reliable container deployment, I explored three technologies: Docker Compose, Docker Swarm, and Kubernetes. I dismissed Docker Compose from my options due to its inability to ensure container reliability. Specifically, if a failure occurs after deploying the containers, Docker Compose cannot restart them, thus compromising the reliability of the setup. Despite Docker Swarm's simplicity, I opted against this option primarily due to its limitations in scaling for larger applications, less community support, and the absence of cloud provider integration, unlike Kubernetes, which enjoys broader industry backing and seamless integration with various cloud platforms. Despite acknowledging the potentially steep learning curve, I opted for Kubernetes due to its comprehensive features and broad industry acceptance.

After selecting Kubernetes as the container orchestrator, the next crucial decision is to choose the most suitable distribution. It's important to note that you similarly manage each distribution. In exploring various distributions, I considered the following options: Rancher, K8s, Minikube, K0s, and K3s. I ruled out the first two options as they are primarily designed for cloud services, making them unsuitable for lightweight devices[16][17]. Minikube presented a limitation with their single-node nature, preventing the deployment of honeypots across multiple locations[18]. Upon closer examination, I found that K0s and K3s share significant similarities in their composition. Ultimately, I chose K3s because of its robust community support. K3s, released a year earlier than K0s, gained widespread adoption and user familiarity, making it the preferred choice[19].

4.4 K3s

Having opted for K3s as the Kubernetes distribution due to its lightweight nature and scalability, let's delve into its offerings and the system composition. The installation process is remarkably straightforward; a simple command initiates the process.

```
curl -sfL https://get.k3s.io | sh -
```

Within moments, utilizing sudo K3s, I gained immediate interaction with a single node, completing the setup in approximately 30 seconds.

Expanding the cluster by adding nodes is equally uncomplicated. Ensuring accessibility of the master node and utilizing the token located at:

```
/var/lib/rancher/k3s/server/node-token
```

Connecting the nodes involves executing the command:

```
curl -sfL https://get.k3s.io | K3S_URL=https:/url K3S_TOKEN=
    mynodetoken sh -
```

Setting up a cluster with K3s is effortless, and the uninstallation process mirrors this simplicity.

```
# Master node
/usr/local/bin/k3s-uninstall.sh
# Agent node
/usr/local/bin/k3s-agent-uninstall.sh
```

Flannel facilitates networking within the cluster, a lightweight layer tree provider that implements a container network interface. Flannel boasts multiple plugins, enabling the customization of security for network traffic. Examples of such plugins include WireGuard and IPsec, both serving as VPN providers. Also, incorporating –flannel-iface into the installation command allows setting the network interface used by the node.

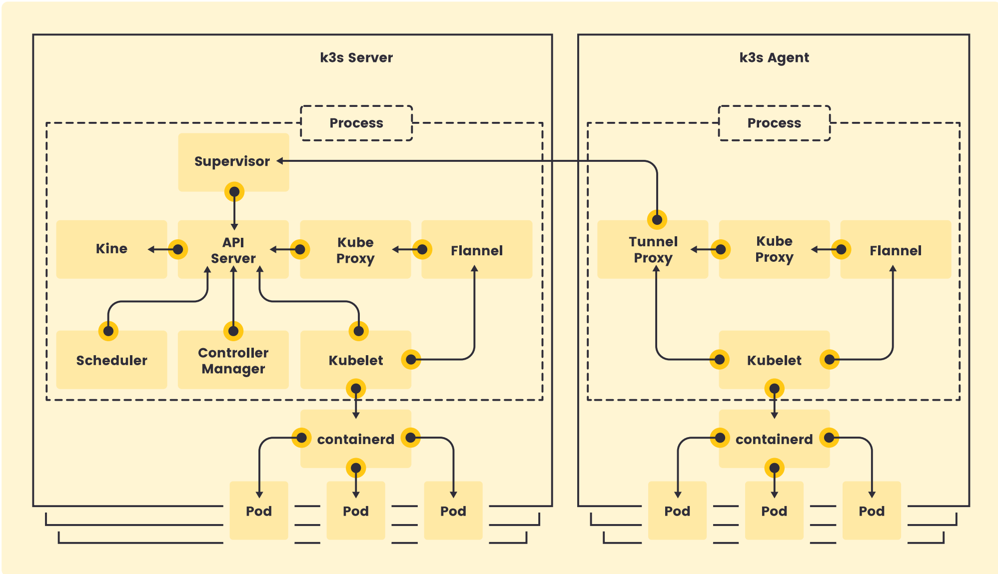


Figure 4.2: K3s Architecture.
Source: Architecture Server and Agents. [20]

Figure 4.2 illustrates the K3s architecture, delineating the distribution of roles among servers and agents. K3 architecture allows the master node to assume the role of an agent node. Within the agent nodes, components include Kubelet, responsible for ensuring container reliability, and Kube Proxy, facilitating communication within the cluster. On the server side, essential components comprise the API server for cluster management, the controller manager overseeing the desired state, the scheduler assigning workloads to nodes, and kine serving as an API to a database functioning as the equivalent of etcd in a standard Kubernetes distribution. This database stores events and states critical for cluster functioning. Finally, guaranteeing communication between the server and the agents involves the tunnel proxy of the agent reaching the supervisor in a unidirectional manner. After establishing the connection, the communication evolves into a bidirectional exchange.

4.5 Wireguard

As outlined in the project objectives, ensuring secure communication between entities is a requirement. Addressing this concern involves implementing a VPN layer between the nodes. Notably, K3s facilitates such integration through plugins, with options like Wireguard and IPsec. It's important to note that each plugin must be installed before its inclusion in the cluster, presenting a constraint in the process.

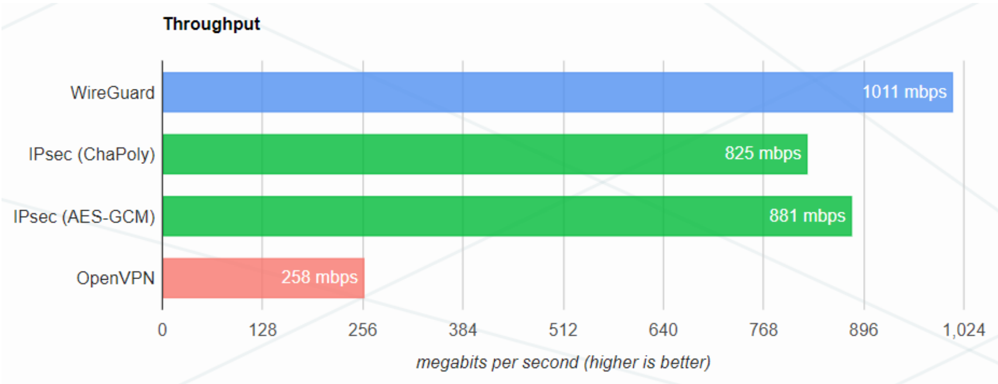


Figure 4.3: Throught comparison between different VPN's.
Source: Benchmarking Results. [21]

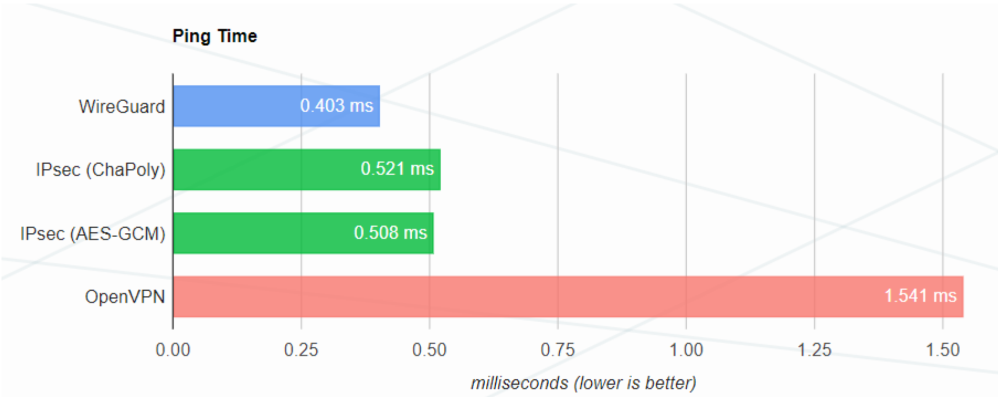


Figure 4.4: Ping Time comparison between different VPN's.
Source: Benchmarking Results. [21]

Let's delve into a comparative analysis of IPsec and Wireguard. According to performance benchmarks on the Wireguard website, both exhibit similar capabilities in latency and bandwidth tests. Figure 4.3 visually demonstrates Wireguard's superior bandwidth performance compared to different configurations of IPsec and OpenVPN. Moreover, Figure 4.4 highlights Wireguard's dominance in latency performance over alternative options. Based on these results, Wireguard emerges as the preferred VPN solution for this cluster.

Turning our attention to Wireguard, the technology prides itself on simplicity compared to other VPNs. All communication between peers is encrypted, contributing to a secure environment. Additionally, cybersecurity experts can easily audit Wireguard, aligning with best security practices. Notably, Wireguard's emphasis on high performance positions it as a fitting choice for embedded devices, aligning with the specific requirements of this project, which aims to run efficiently on low-capability devices.

Now, let's explore how Wireguard works. In its fundamental operation, Wireguard integrates a network interface into the host, typically denoted as wg0, or even multiple interfaces like wg1, wg2, etc. The architecture of Wireguard relies on two pairs of keys, a public key and a private key. When establishing a connection between a server and a client, Wireguard associates the IP address of the endpoint with its corresponding public key. In the process, Wireguard identifies the port configured on the other peer, allowing it to send an encrypted message. At the client end, Wireguard inspects whether the incoming IP address is permitted; if not, Wireguard promptly drops the message, enhancing the security of the communication channel.

To initiate Wireguard setup, the process varies depending on the operating system. I jump to this step replacing IPs and keys by placeholder in the examples. Specifically, for Linux-based systems, executing the following command suffices:

```
sudo apt install wireguard
```

After installing Wireguard, the following steps include generating keys and configuring the network interface. To create the public and private keys, utilize the following command:

```
wg genkey | tee privatekey | wg pubkey > publickey
```

To configure the network interface, create a file at the following path:

```
/etc/wireguard/wg0.conf
```

The following file details the server configuration:

```
[Interface]
Address = [ServerIP]
PrivateKey = [ServerPrivateKey]
ListenPort = 51820

[Peer]
```

```
PublicKey = [Peer1PublicKey]
AllowedIPs = [Peer1IP]

[Peer]
PublicKey = [Peer2PublicKey]
AllowedIPs = [Peer2IP]
```

For a client, the configuration is as follows:

```
[Interface]
Address = [Peer1IP]
PrivateKey = [Peer1PrivateKey]

[Peer]
PublicKey = [ServerPublicKey]
Endpoint = some.domain.com:51820
AllowedIPs = [Peer1IP]
```

4.6 Workloads, services and networking

Considering the configuration of a K3s cluster and the integration of WireGuard, let's delve into the realm of Kubernetes and explore its associated tools. In particular, to incorporate any resource into a cluster, a manifest is required. This manifest, typically in YAML format, comprehensively outlines the various parameters a resource may possess, similar to a JSON file. It's crucial to recognize that only the master node or a node with appropriate permissions can manipulate the cluster. All the manifest can be applied to the cluster using the following kubectl command:

```
kubectl apply -f manifest.yaml
```

Given the extensive array of components within Kubernetes, I'll focus solely on those pertinent to the prototype: Pods, Deployments, Services, and network policies.

4.6.1 Pod

Upon my initial encounter with pods, I mistakenly viewed them as containers. However, pods are more intricately categorized, encompassing those with a single container and those housing multiple containers. Indeed, Kubernetes allocates a cluster IP to each pod and hosts it on a specific node within the cluster. By utilizing labels, one can correlate nodes with other resources in the cluster, such as services and network policies. I'll delve into this further when explaining the prototype.

Consider the example of an nginx application:

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

Inside the manifest, observe the pod's distinctive name, which must be unique among other pods in the cluster. It is imperative to list the containers along with their respective names, images, and required ports. Employing the command below, I gain an overview of all pods in the cluster.

```
kubectl get pods -o wide
```

Figure 4.5: List of a pod with its parameters.

Notably, in Figure 4.5, an IP address and a cluster are assigned to this pod. Now, let's consider a pod housing multiple containers. A manifest for such a scenario may appear as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
  - name: busybox-container
    image: busybox:latest
```

```
command: ['sh', '-c', 'while true; do echo Hello from BusyBox; sleep 10; done']
```

When executing the command to list pods, the "ready" option shows as 2/2, indicating the successful deployment of both containers. An issue I encountered involved two containers necessitating the same port within their images; in such cases, the deployment only deploys one of the two containers.

4.6.2 Deployment

Pods offer a convenient approach to developing various applications. Kubernetes operates like Docker Compose in that case. However, utilizing Kubernetes deployments is essential when fortifying honeypots against severe attacks. Deployments serve as mechanisms for managing high demand and ensuring the stability of pods. Essentially, deployments operate similarly to pods, with the distinction in the replica set, which determines the number of replicas to add for load-balancing purposes. Below is an example of a deployment manifest in Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Employing the command below, I list the different deployments. A simple command suffices, much like listing pods:

```
kubectl get deployments -o wide
```

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                 SELECTOR
cowrie-2           2/2     2            2           26d   cowrie       cowrie/cowrie:latest   app=cowrie-2
cowrie-1           0/1     1            0           26d   cowrie       cowrie/cowrie:latest   app=cowrie-1
nginx-deployment   3/3     3            3           13s   nginx        nginx:latest           app=nginx
```

Figure 4.6: List of deployments with its parameters.

Figure 4.6 displays the command logs on the screen, showcasing the similarity it shares with pods. However, instead of the containers being ready, the pods indicate readiness.

4.6.3 Volumes

I've delved into the functionalities of pods and deployments, indispensable tools within the Kubernetes environment. However, one crucial aspect yet to be addressed is how Kubernetes manages data storage and persistence. Here is where volumes step in as a solution. Volumes are classified into two parts, each serving distinct purposes: persistent volumes and persistent volume claims.

Persistent volumes essentially act as physical storage spaces within your cluster, facilitating the linkage to one or multiple applications. For instance, if you aim to store data on a node, you can establish a connection between a pod's folder and a folder on the actual machine. In the event of pod failure, persistent volumes ensure seamless data retention, maintaining continuity between the preceding and newly created pods.

On the other hand, persistent volume claims are responsible for identifying the appropriate persistent volume within the cluster for the application that needs storage information. For example, if in the cluster are persistent volumes available in capacities of 1MB, 2MB, and 1GB, and your application requires a maximum of 1GB, the associated persistent volume claim of 1GB will seek out a suitable persistent volume with matching capabilities and allocate storage accordingly.

Here's an illustrative example of a persistent volume and a persistent volume claim within a manifest:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
```

```
  storageClassName: manual
  hostPath:
    path: /data/example

---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: manual
```

The deployment manifest must include a persistent volume claim, to store the data. Here's an example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
        - name: example-volume
          mountPath: /data
      volumes:
      - name: example-volume
        persistentVolumeClaim:
          claimName: example-pvc
```

Utilizing these tools enables me to manage the storage of honeypots in a physical location. In Kubernetes, there are various storage options, the most prevalent being local or network file system storage.

4.6.4 Services

So far, I've discussed three components in Kubernetes: pods, deployments, and volumes. However, I haven't addressed how to make applications accessible. Where a tool in Kubernetes, called a service, comes into play. A service can expose one or multiple pods to a network, making them accessible to use, in my case, including potential cyber attackers.

For this application, I'll focus on a specific option within services called External IPs. External IPs allow applications to be exposed using the IP addresses of one or more cluster nodes. The following manifest explains in detail:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: your-image:latest
        ports:
        - containerPort: 80
        volumeMounts:
        - name: data-volume
          mountPath: /data
  volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: my-pvc
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: my-app-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  externalIPs:
    - 198.51.100.32
```

In this manifest, you can see that the deployment and the service match by a label. You can also define the protocol, the port on the node where you want to run the application, and which nodes will host that port. In this case, the target port specifies the port the node will use.

One particularly notable aspect of Kubernetes is its ability to deploy a pod on one node and expose it to another. This flexibility makes it powerful for supporting lightweight devices. In some scenarios, such as running multiple honeypots, you could direct all the workload to a specific node and redirect attacker access from other nodes with lesser capabilities. In the prototype, I will describe in detail this case.

4.6.5 Network policies

An aspect that warrants attention is Kubernete's incorporation of namespaces, a tool enabling the segregation of applications within distinct spaces. In practical terms, this means that disparate groups of developers operating within the same cluster can operate in separate namespaces without necessitating direct interaction. However, this involves network policies to regulate traffic throughout the cluster, thus confining interactions. By default, communication is unrestricted across the cluster, permitting any component to communicate with another. If network policies are not defined, for instance, when deploying two pods, they can transfer data between each other.

Network policies dictate the ingress and egress traffic for pods. They determine which entities can interact with each other through three distinct identifiers: pods, namespaces, and IP blocks. The first two utilize Kubernetes selectors, like labels, while IP blocks employ CIDR ranges to specify the permissible traffic flow. Below is an example manifest showcasing all the mentioned components:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: example-network-policy
spec:
  podSelector:
    matchLabels:
      app: example-app
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    - namespaceSelector:
        matchLabels:
          environment: production
    - ipBlock:
        cidr: 192.168.0.0/24
  egress:
  - to:
    - podSelector:
        matchLabels:
          role: backend
    - namespaceSelector:
        matchLabels:
          environment: staging
    - ipBlock:
        cidr: 10.0.0.0/16
```

In the previous example, the podSelector ensures that pods are authorized to communicate. Meanwhile, the namespaceSelector designates the desired namespace. Finally, the CIDR range specifies the IP ranges not restricted from interacting with those pods.

Chapter 5

Prototype

5.1 Physical Layer

Initiating the building of a cluster involves acquiring the necessary nodes. As mentioned previously, the Smart Data Center at Politecnico provided one virtual machine while I implemented the other two on my personal computer. Access to the Politecnico virtual machine was granted via SSH, allowing remote connection from my computer using an SSH certificate. The setup for the other two machines proceeded as follows:

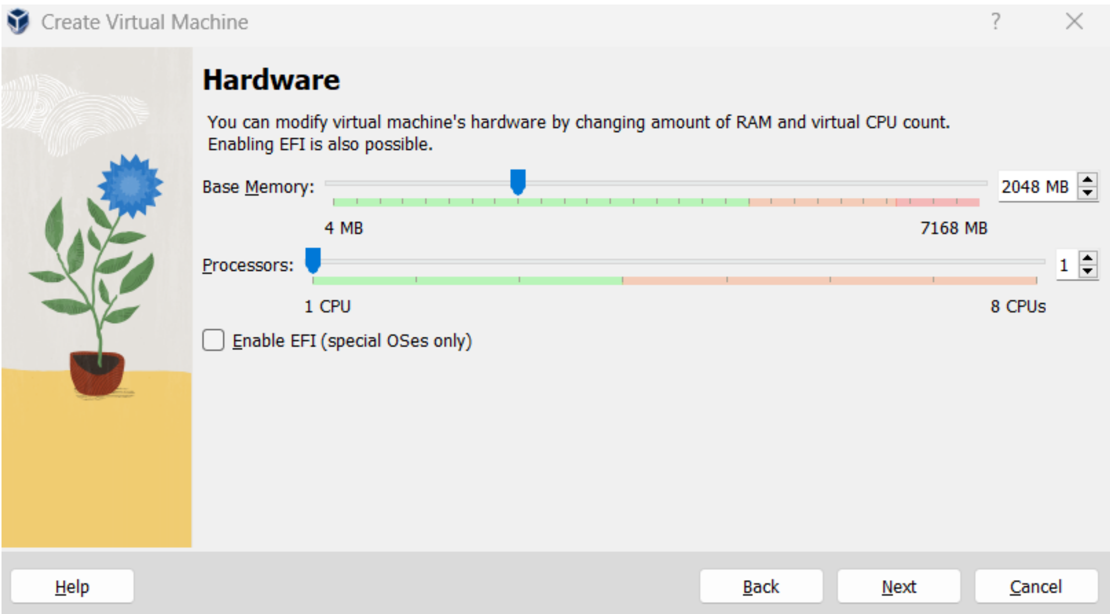


Figure 5.1: Panel of hardware specification for a virtual machine in VirtualBox.

To begin, VirtualBox must be installed and available in various versions compatible with different operating systems. The next step involves downloading the Ubuntu Server 22.04 disk image. When I selected "New" in the VirtualBox interface, it prompted me to specify a name and select the ISO image. Subsequently, I defined the credentials for the virtual machine and configured its hardware capabilities based on my host's specifications, as depicted in Figure 5.1. Following this, I allocated the desired amount of memory from my local computer to the virtual hard disk. Finally, the setup process concluded by launching the virtual machine in a new tab.

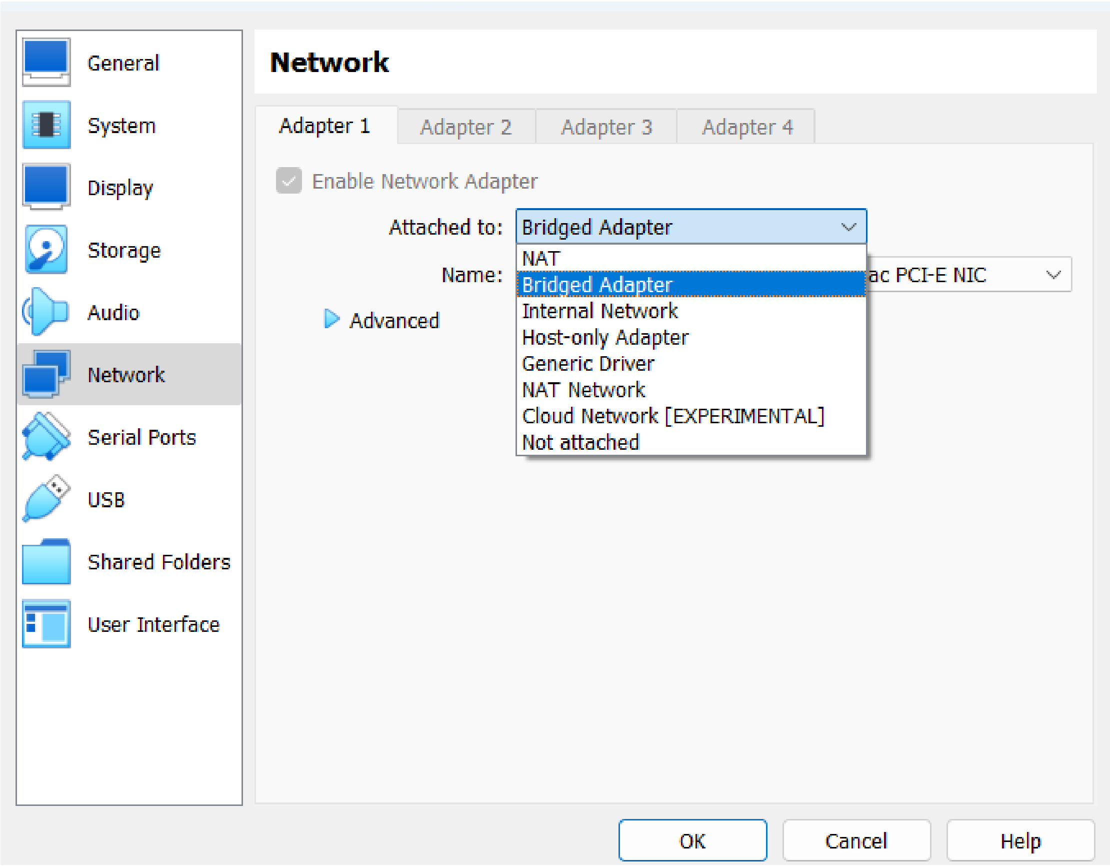


Figure 5.2: Panel of network options for a virtual machine in VirtualBox.

Once the virtual machine starts, it will prompt you to select the operating system installation. Upon proceeding, it will inquire about your preferred language. You can opt for the minimized version of Ubuntu Server, which omits many packages to conserve resources. After confirming, it will prompt for interface settings, typically set to default. When asked to utilize the entire disk, you should accept, as this allocates the previously selected virtual disk. Once all prerequisites are accepted, it will request a username and password to enter the machine. Optionally, you can enable OpenSSH for remote terminal access from your computer. Subsequently, it will inquire about popular packages, with Docker needed for integration with Kubernetes infrastructure. After installation and reboot, it defaults to using NAT networking for installation. If you desire an alternate IP bridge adapter mode, you should configure it accordingly. Refer to Figure 5.2 for the available option to ensure that your bridge adapter is the internet card operating.

5.2 Cluster Layer

As per the methodology, this layer involves using two primary technologies: K3s and WireGuard. Setting up WireGuard is a prerequisite for K3s, necessitating its configuration as the initial step. The configuration varies between the master and worker nodes in several aspects. I will delve into each script I've designed, elucidating how they enhance reliability and ease of implementation.

```
#!/bin/bash

# Set the WireGuard IP address
WG_IP="10.0.0.1"

# Set the internet interface
iface="enp0s3"

# Install WireGuard
sudo apt-get install -y wireguard

# Generate WireGuard keys
wg genkey | tee privatekey | wg pubkey > publickey

# Create WireGuard configuration file
sudo bash -c "cat > /etc/wireguard/wg0.conf << EOF
[Interface]

# The IP address of this host in the wireguard tunnels
Address = $WG_IP

# Every Raspberry Pi connects via UDP to this port. Your Cloud VM
    must be reachable on this port via UDP from the internet.
ListenPort = 51820

# Set the private key to the value of the privatekey file generated
    by the previous command
PrivateKey = $(cat privatekey)

```

```
PostUp   = iptables -A FORWARD -i %i -j ACCEPT; iptables -A FORWARD -o %i -j ACCEPT; iptables -t nat -A POSTROUTING -o $iface -j MASQUERADE
PostDown = iptables -D FORWARD -i %i -j ACCEPT; iptables -D FORWARD -o %i -j ACCEPT; iptables -t nat -D POSTROUTING -o $iface -j MASQUERADE
EOF"

# Create WireGuard Up service file
sudo bash -c "cat > /etc/systemd/system/wireguard-up.service << EOF
[Unit]
Description=WireGuard Up Service
After=network.target

[Service]
Type=simpled
ExecStart=/usr/bin/wg-quick up wg0

[Install]
WantedBy=default.target
EOF"

# Reload systemd and start WireGuard service
sudo systemctl daemon-reload
sudo systemctl enable wireguard-up.service
sudo systemctl start wireguard-up.service

# Install K3s with WireGuard
curl -sfL https://get.k3s.io | K3S_NODE_NAME=master sh -

# Check for an empty line and remove it before appending the echo message
sed -i '/^$/d' /etc/systemd/system/k3s.service

# Configure flannel in node
echo "ExecStart=/usr/local/bin/k3s server --advertise-address $WG_IP --flannel-iface=wg0" | sudo tee -a /etc/systemd/system/k3s.service

# Restart k3s service
sudo systemctl daemon-reload
sudo systemctl restart k3s.service
```

Let's commence with the script for configuring the master node. As is shown, two variables are required: the IP address of the WireGuard interface and the internet interface. The script proceeds with the installation of WireGuard, followed by the generation of public and private keys. Subsequently, a new interface file is crafted for the WireGuard interface, necessitating the addition of the IP address, port, and private key. Additionally, two lines for 'postup' and 'postdown' are inserted to facilitate traffic forwarding through the Linux system serving as a router or gateway.

One issue encountered was the failure of the system to reinstate the interface upon reboot. I deployed a systemd service to automate configuring the WireGuard interface during the start of the system. This service must be enabled and initiated. With WireGuard fully configured, the installation of K3s follows, accomplished with a simple command. Within the 'K3S_NODE_NAME' parameter, I specify the node's name. After the K3s installation completion, the WireGuard interface is incorporated into the K3s service via a command, ensuring it's placed correctly within the configuration file. Afterward, I restarted the K3s service with the WireGuard interface integrated.

```
#!/bin/bash

# Set Wireguard peer Public Key
WG_PK="K30I8eIxuBL3OA43Xl34x0Tc60wqyDBx4msVm8VLkAE="

# Set Wireguard peer IP address
WG_IP="10.0.0.2"

# Add peer into network
sudo wg set wg0 peer $WG_PK allowed-ips $WG_IP/32
sudo ip -4 route add $WG_IP/32 dev wg0

# Save configuration
sudo wg-quick save wg0
```

Now that WireGuard and k3s are operational on the master node, it's currently the sole node in the cluster, meaning all workload remains centralized. However, there's a solution to this by adding peers to the WireGuard connection. The process involves obtaining the peers's public key and IP address, then utilizing the WireGuard 'add peer' command and enabling the route to that peer. For preservation, these peers should be saved into the configuration, as demonstrated in the final line of code. Ideally, the master node should assign the IP address to the peer to avoid duplication within the cluster.

```
#!/bin/bash

# Set the WireGuard IP address
WG_IP="10.0.0.2"

# Install WireGuard
sudo apt-get install -y wireguard
```

```

# Generate WireGuard keys
wg genkey | tee privatekey | wg pubkey > publickey

# Create WireGuard configuration file
sudo bash -c "cat > /etc/wireguard/wg0.conf << EOF
[Interface]
Address = $WG_IP/24
PrivateKey = $(cat privatekey)

[Peer]
PublicKey = <Publickey_Master>
Endpoint = <domain>:<port>
AllowedIPs = 10.0.0.1/32
EOF"

# Create WireGuard Up service file
sudo bash -c "cat > /etc/systemd/system/wireguard-up.service << EOF
[Unit]
Description=WireGuard Up Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/wg-quick up wg0

[Install]
WantedBy=default.target
EOF"

# Reload systemd and start WireGuard service
sudo systemctl daemon-reload
sudo systemctl enable wireguard-up.service
sudo systemctl start wireguard-up.service

unset WG_IP
```

After establishing the master node, it's time to configure the worker nodes. While all worker nodes undergo the same configuration process, they must input different variables as parameters. Configuring a worker node involves setting up WireGuard and configuring K3s. Upon completing the WireGuard phase, the master node must add the worker node as a peer into the WireGuard network to ensure complete k3s integration.

As evident from the previous code, similar to the master node, the initial step is to install WireGuard. Following this, I create a new interface with the assigned WireGuard IP, a selection best made by the master node, which possesses knowledge of the IP for each peer in the cluster.

Additionally, I must add the private key. The master node is added as a peer in the configuration, requiring the master's public key and endpoint (the domain or IP through which one can reach the master node, along with its corresponding port). The 'allowedips' line establishes which peers can communicate with the worker node. Similar to the master node, it's necessary to configure the WireGuard service to initialize the WireGuard interface every time the system reboots. Once this process is complete, the worker node's public key must be sent to the master node to be added, following the same procedure previously.

```
#!/bin/bash

# Set the WireGuard IP address
WG_IP="10.0.0.2"

# Install K3s with WireGuard
curl -sfL https://get.k3s.io | K3S_URL=https://10.0.0.1:6443
    K3S_TOKEN=<master_node_token> K3S_NODE_NAME=name-node sh -

# Check for an empty line and remove it before appending the echo
    message
sed -i '/^$/d' /etc/systemd/system/k3s-agent.service

# Configure flannel in node
echo "ExecStart=/usr/local/bin/k3s agent --node-ip $WG_IP --flannel-
    iface=wg0" | sudo tee -a /etc/systemd/system/k3s-agent.service

# Restart k3s service
sudo systemctl daemon-reload
sudo systemctl restart k3s-agent.service

# Unset the WireGuard IP address variable
unset WG_IP
```

Once the WireGuard setup is complete, you can verify its functionality by using the ping command to reach the IP of the master node. Next, we detail the setup procedure for K3s on the agent in the following code snippet. Firstly, it's essential to specify the WireGuard IP of the worker node, followed by executing the node installation using the subsequent line of code. Let's break this down further: Initially, the K3S_URL must contain the IP address of the master node on the WireGuard interface, utilizing port 6443 (the designated port for K3s). The K3S_TOKEN needs to be provided by the master node, as described in the methodology. You can find this token in the following path:

```
/var/lib/rancher/k3s/server/node-token
```

When specifying the K3S_NODE_NAME, each node should have a distinct name. Otherwise, the master won't be able to recognize it within the cluster. The subsequent lines configure the WireGuard interface as the cluster interface and restart the K3s service. Notably, the service name for the worker node differs from that of the master node. To check the status of each node, run the following command:

```
kubectl get nodes
```

```
NAME          STATUS     ROLES                  AGE    VERSION
server9       NotReady   <none>                 40d    v1.28.6+k3s2
dpi           NotReady   <none>                 118d   v1.27.7+k3s2
server2       NotReady   <none>                 111d   v1.27.7+k3s2
server1       Ready      <none>                 96d    v1.27.7+k3s2
vm1-bigdata   Ready      control-plane,master   118d   v1.27.7+k3s2
```

Figure 5.3: Display of the nodes in the cluster.

It will display which nodes are available and which are not. Figure 5.3 illustrates all the relevant information regarding the nodes in the cluster. The master node is responsible for rescheduling workloads to available nodes with resources.

5.3 Service Layer

Now that the cluster layer is complete, I have a Kubernetes cluster capable of running any desired application. With an infinite array of deployment possibilities and millions of images on Docker Hub, I'll limit this section to a simple prototype showcasing the system's capability to distribute honeypots.

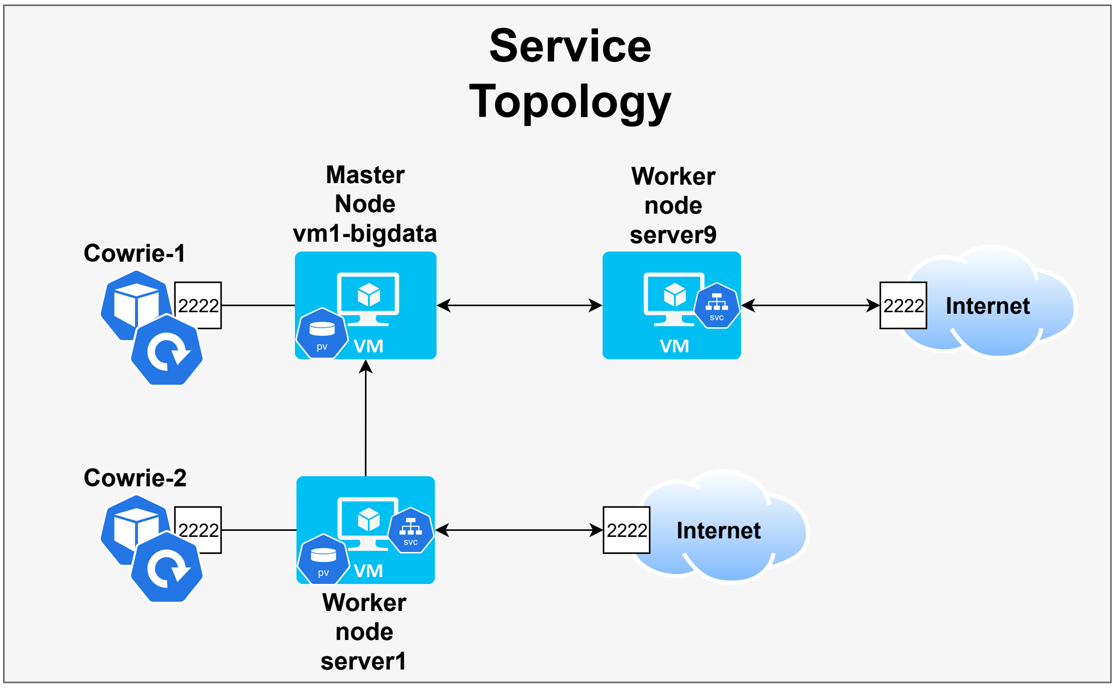


Figure 5.4: Detailed layer of service of the prototype.

In Figure 5.4, I outline the prototype for building and testing at the service layer. It consists of three nodes: one master and two worker nodes, all virtual machines running Ubuntu. The master node will host a Cowrie honeypot deployment, along with worker node server1. Each node will locally store the data from these honeypots. However, as mentioned in the methodology, there's a method for exposing these services on a different node, even if the pods are not in the node.

Regarding the persistence volumes and persistent volume claims. It's necessary to create a persistence volume on each node where the honeypot will deploy for local storage on the node. Below are the manifests for creating these storage resources on the master node and the worker node, respectively:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: cowrie-pv-1
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
```

```
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/"
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - vm1-bigdata
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cowrie-pvc-1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

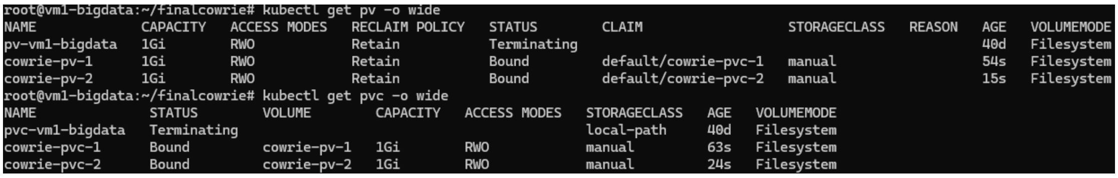
```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: cowrie-pv-2
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/"
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
```

```
          - server1

---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cowrie-pvc-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

It is important to note that the persistence volume folder needs to have write permissions; otherwise, the honeypot will not be able to write to it. If you wish to experiment without permission, you can utilize the ¨*tmp* folder. You can list the different persistence volumes and persistence volumes claims respectively with the following lines:

```
kubectl get pv -o wide
```

```
kubectl get pvc -o wide
```



```
root@vm1-bigdata:~/finalcowrie# kubectl get pv -o wide
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS        CLAIM                   STORAGECLASS   REASON   AGE   VOLUMEMODE
pv-vm1-bigdata   1Gi        RWO            Retain           Terminating                                                 40d   Filesystem
cowrie-pv-1      1Gi        RWO            Retain           Bound         default/cowrie-pvc-1   manual                  54s   Filesystem
cowrie-pv-2      1Gi        RWO            Retain           Bound         default/cowrie-pvc-2   manual                  15s   Filesystem
root@vm1-bigdata:~/finalcowrie# kubectl get pvc -o wide
NAME              STATUS        VOLUME        CAPACITY   ACCESS MODES   STORAGECLASS   AGE   VOLUMEMODE
pvc-vm1-bigdata   Terminating                                           local-path     40d   Filesystem
cowrie-pvc-1      Bound         cowrie-pv-1   1Gi        RWO            manual         63s   Filesystem
cowrie-pvc-2      Bound         cowrie-pv-2   1Gi        RWO            manual         24s   Filesystem
```

Figure 5.5: List of persistence volumes and persistence volumes claims.

Figure 5.5 illustrates persistence volumes and persistence volume claims inside the cluster. After defining the persistence volumes and persistence volume claims, it's time to implement the deployments of the honeypots. Each deployment needs to link to a persistence volume claim to store the data captured by the honeypot. The honeypot chosen is Cowrie, which emulates a UNIX operating system, making it perfect for capturing brute-force attacks. Both deployments will run just one replica. Below are the manifests for both deployments:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cowrie-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cowrie-1
  template:
    metadata:
      labels:
        app: cowrie-1
    spec:
      volumes:
      - name: cowrie-data
        persistentVolumeClaim:
          claimName: cowrie-pvc-1
      containers:
      - name: cowrie
        image: cowrie/cowrie:latest
        volumeMounts:
        - name: cowrie-data
          mountPath: "/cowrie/cowrie-git/var/log/cowrie"
      nodeName: vm1-bigdata
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cowrie-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cowrie-2
  template:
    metadata:
      labels:
        app: cowrie-2
    spec:
      volumes:
      - name: cowrie-data
        persistentVolumeClaim:
          claimName: cowrie-pvc-2
      containers:
      - name: cowrie
```

```
        image: cowrie/cowrie:latest
        volumeMounts:
        - name: cowrie-data
          mountPath: "/cowrie/cowrie-git/var/log/cowrie"
      nodeName: server1
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
cowrie-1	1/1	1	1	24m	cowrie	cowrie/cowrie:latest	app=cowrie-1
cowrie-2	1/1	1	1	24m	cowrie	cowrie/cowrie:latest	app=cowrie-2

Figure 5.6: List of deployments.

Figure 5.6 displays the list of Cowrie deployments. The only remaining components are the services and network policies. While network policies aren't necessary for testing purposes, they are essential in a production environment. The services for these honeypots are exposed using the external IP, equivalent to the IP of the node to be exposed. Below are the manifests for these services:

```
apiVersion: v1
kind: Service
metadata:
  name: cowrie-1
spec:
  selector:
    app: cowrie-1
  ports:
    - protocol: TCP
      port: 2222
      targetPort: 2222
  externalIPs:
    - 192.168.1.175
```

```
apiVersion: v1
kind: Service
metadata:
  name: cowrie-2
spec:
  selector:
    app: cowrie-2
  ports:
    - protocol: TCP
      port: 2222
      targetPort: 2222
  externalIPs:
```

```
    - 192.168.1.70
```

Figure 5.7 illustrates the list of different services in the cluster after executing the following command:

```
kubectl get services -o wide
```

```
kubernetes   ClusterIP   10.43.0.1       <none>          443/TCP    118d   <none>
cowrie-1     ClusterIP   10.43.146.236   192.168.1.175   2222/TCP   31s    app=cowrie-1
cowrie-2     ClusterIP   10.43.110.150   192.168.1.70    2222/TCP   29s    app=cowrie-2
```

Figure 5.7: List of services.

With all these manifests applied to the cluster, the service layer for the prototype is now complete. In the next chapter, I will conduct a test to show the attacker's perspective and how the node saves the data.

Chapter 6

Benchmarks

After implementing the prototype, it's relevant to conduct measurements to assess the performance behavior of the system. Therefore, I will perform five different tests. Four tests will focus on statistical analysis after measuring a significant sample size. In contrast, the fifth test will demonstrate the behavior of the prototype in the service layer (from the attacker's perspective and data storage).

For the statistical tests, I will utilize the mean (described by equation 6.1), the standard deviation (followed by equation 6.2), and the confidence interval considering a confidence level of 90 percent (indicating a 10 percent chance of being wrong, as shown by equation 6.3). In those equations, x_i represents each measurement, N is the number of measurements, $\bar{x}$ is the mean, σ is the standard deviation, and z is the confidence level value (1.645).

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{6.1}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}} \tag{6.2}$$

$$\mathrm{CI} = \bar{x} \pm z \frac{\sigma}{\sqrt{N}} \tag{6.3}$$

6.1 Network Latency

I aimed to assess network latency to predict the time required for future actions. To achieve this, I systematically tested various configurations among nodes and pods. Utilizing the ping tool, depicted in Figure 6.1, I measured delays. Ping's efficacy in determining host availability, owing to its transmission of small packets, rendered it ideal for gauging network delays.

```
root@vm1-bigdata:~# ping -c 8 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=9.25 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=10.0 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=11.4 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=9.31 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=9.35 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=10.8 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=9.15 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=9.83 ms

--- 10.0.0.3 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7010ms
rtt min/avg/max/mdev = 9.153/9.889/11.358/0.764 ms
```

Figure 6.1: Ping command example.

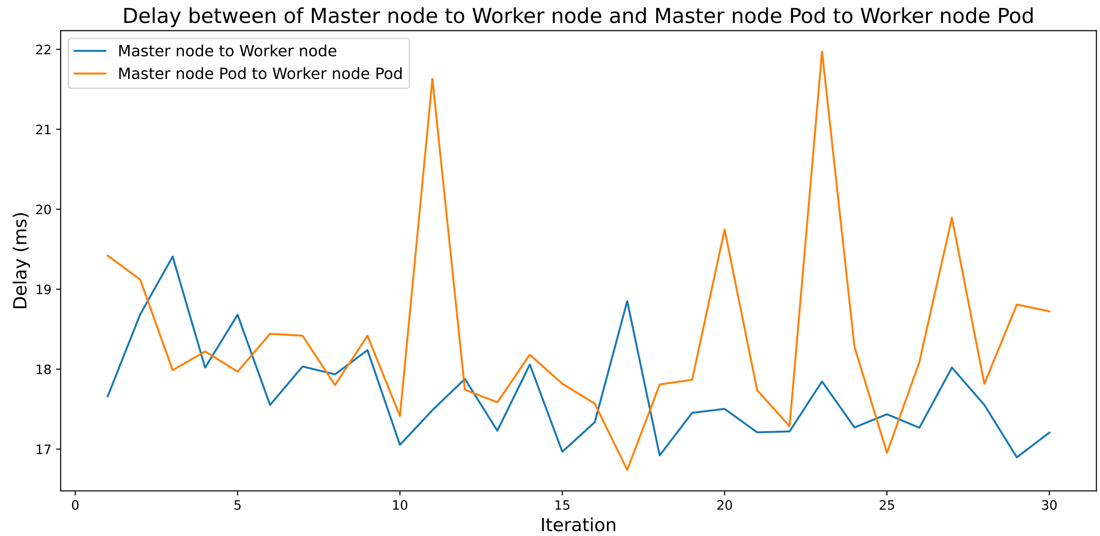


Figure 6.2: Delay comparison between node to node and pod to pod.

The experimentation encompassed measuring delays between the master node and working node, between the master node and a pod running within it, between two pods within the master node, and between a pod in the master node and another in the working node.

I notice that the two results within a node yield values based on the hardware limitations on site. In Figure 6.2, I compare the discrepancy in delay between a

Peers	Mean (ms)	Std (ms)	90% CI (ms)
Master node - Worker node	17.69	0.603	[17.51, 17.87]
Master node - Master node pod	0.069	0.017	[0.064, 0.075]
Master node pod - Master node pod	0.082	0.019	[0.077, 0.088]
Master node pod - Worker node pod	18.38	1.16	[18.03, 18.73]

Table 6.1: Statistical results of network delay for each possible peer.

master node and a worker node, as well as between two pods within those nodes. After conducting 30 sample runs, each comprising ten traces, I derived statistical insights for each configuration. Table 6.1 presents the obtained results.

6.2 Network Bandwidth

I replicated the experiments conducted in the preceding section here, focusing on measuring bandwidth instead of latency. For this purpose, I employed iperf3, a tool that sets up a server-client configuration to initiate data transfer, thereby assessing network capabilities. As illustrated in Figure 6.3, this tool typically conducts a 10-second measurement.

```
root@vm1-bigdata:~# iperf3 -c 10.0.0.3
Connecting to host 10.0.0.3, port 5201
[  5] local 10.0.0.1 port 41086 connected to 10.0.0.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   556 KBytes  4.55 Mbits/sec   28   12.0 KBytes
[  5]   1.00-2.00   sec  3.06 MBytes  25.7 Mbits/sec    0   65.5 KBytes
[  5]   2.00-3.00   sec  4.83 MBytes  40.5 Mbits/sec    2    106 KBytes
[  5]   3.00-4.00   sec  6.93 MBytes  58.1 Mbits/sec    0    146 KBytes
[  5]   4.00-5.00   sec  5.52 MBytes  46.3 Mbits/sec    0    171 KBytes
[  5]   5.00-6.00   sec  7.36 MBytes  61.7 Mbits/sec    0    198 KBytes
[  5]   6.00-7.00   sec  13.6 MBytes   114 Mbits/sec    0    243 KBytes
[  5]   7.00-8.00   sec  9.07 MBytes  76.1 Mbits/sec    0    269 KBytes
[  5]   8.00-9.00   sec  8.83 MBytes  74.1 Mbits/sec    0    293 KBytes
[  5]   9.00-10.00  sec  11.5 MBytes  96.2 Mbits/sec    0    319 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  71.2 MBytes  59.7 Mbits/sec   30             sender
[  5]   0.00-10.01  sec  70.6 MBytes  59.2 Mbits/sec                  receiver
```

Figure 6.3: Iperf command example.

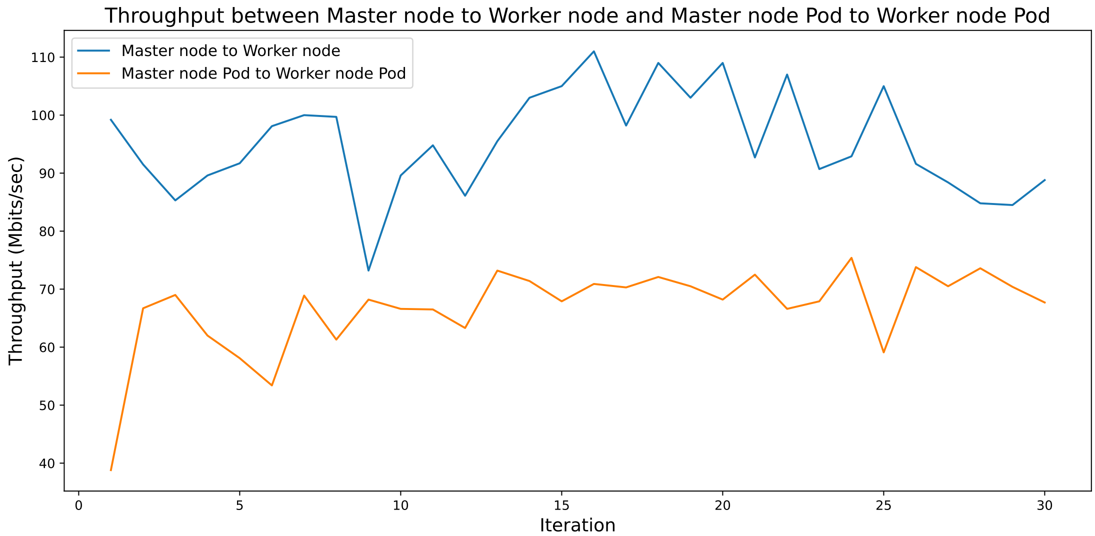


Figure 6.4: Throughput comparison between node to node and pod to pod.

Peers	Mean (Mbits/s)	Std (Mbits/s)	90% CI (Mbits/s)
Master node - Worker node	95.29	8.69	[92.68, 97.90]
Master node - Master node pod	13493.3	11398.6	[13151, 13835.6]
Master node pod - Master node pod	11367.7	1210.78	[11004.1, 11731.3]
Master node pod - Worker node pod	66.82	7.16	[64.67, 68.97]

Table 6.2: Statistical results of network bandwidth for each possible peer.

I essentially replicated the comparison from the previous section, this time focusing on the throughput of the network. As depicted in Figure 6.4, the results were significantly divergent, possibly owing to the capabilities inherent in a node compared to the relatively simpler resources inside a pod. Analysis of the results presented in Table 6.2 reveals that, in this instance, the bandwidth between nodes surpasses that between pods. Additionally, other probes indicate that internal processes operate at speeds reaching gigabits per second.

6.3 Pod Deployment Time

In the pod deployment time test, I create 30 consecutive pods using a cowrie image, considering both the time taken for the creation and the time until the pod is ready, and then calculate the difference. Table 6.3 presents the statistical values obtained. This experiment was conducted separately for the master node and a working node.

Node	Mean (ms)	Standart deviation (ms)	90% Confident Interval (ms)
Master	1812	329	[1271, 2353]
Worker	2903	768	[1640, 4166]

Table 6.3: Statistical results of pod deployment time.

The results indicate that the pod creation time ranges between one and four seconds. That's due to the pre-stored image that the pod requires from the cluster's registry. It is noteworthy that the values differ between the master node and the working node, which could be due to the time taken for the master to notify the Kubelet of the working node, as it needs to traverse the network.

6.4 Node Failure and Recovery

For the test of node failure and recovery in the K3s architecture, I needed to understand the underlying mechanism. Rather than immediately notifying the system of a node failure, the master node initiates periodic heartbeat signals. To accurately assess node recovery time, it was necessary to wait for the node to report its status as offline before initiating the reboot process, ensuring a measurable interval between states.

The measurement script employed for this purpose followed a sequence: it initiated a reboot command via SSH to the node and then commenced time tracking. Upon the transition from an offline to an online state, the script recorded the end time, allowing for the calculation of the recovery duration.

I replicated this experiment 30 times, rebooting the virtual machine on each occasion. The results presented in Table 6.4 indicate variability in the data, likely influenced by occasional jobs running before some reboots.

Mean (s)	Standart deviation (s)	90% Confident Interval (s)
56.83	14.45	[40.11, 70.56]

Table 6.4: Statistical results of node recovery time.

6.5 Data collection

In the latest test I conducted, I aimed to showcase the prototype defined in the service layer discussed in the previous chapter. Figure 6.5 illustrates a terminal on the left side, displaying the file containing all the commands and passwords proposed by the attacker. On the right side of the screen, you can observe the actions I performed, highlighting the striking resemblance between the Cowrie honeypot and a Unix system.

Figure 6.5: Data storage vs attacker action.

Chapter 7

Conclusion

In conclusion, this thesis has successfully achieved its primary objective of distributing honeypots across various locations. Additionally, it has realized several goals broadening the project's success.

The initial phase involved meticulous selection of technologies suitable for running honeypots, with virtual machines and containers emerging as the primary candidates. Subsequently, I decided to employ Kubernetes for orchestrating and ensuring control over the honeypots due to its scalability and robust community support.

Further refinement led to the adoption of K3s, a lightweight Kubernetes distribution capable of facilitating multi-node deployment across multiple locations. This decision was pivotal in ensuring the efficient utilization of resources and the seamless integration of honeypots into diverse environments.

Moreover, I implemented a Wireguard deployment as the VPN solution, instrumental in achieving secure communication across nodes, with benchmarking results highlighting its superior performance compared to alternative options.

After the decision-making, I developed a three-layered prototype encompassing physical, cluster, and service layers. The physical layer laid the groundwork by establishing virtual machine nodes. In the cluster layer, I employed algorithms to ensure the reliability and resilience of the network, even in the face of system failures.

The service layer showcased the design of honeypots exposed across different locations, facilitated by deployments, volumes, and services orchestrated through Kubernetes.

Rigorous testing of the prototype yielded promising results, with data transmission rates between nodes averaging between 92.6 to 97.9 Mbits/s and minimal latency ranging from 17.5 to 17.9 ms. Furthermore, the prototype demonstrated satisfactory performance in pod generation and node recovery, with pods being instantiated within 1 to 4 seconds and node recovery times ranging from 40 to

70 seconds, notwithstanding the inherent delay in node disconnection notification within the K3s cluster.

In essence, this thesis achieves its primary objective of deploying distributed honeypots and establishes a foundation for future advancements in cybersecurity infrastructure. Dispersing honeypots across various locations enables the capture of diverse patterns that may not be observable within a single geographic area. The insights garnered from this research are invaluable contributions to the continual evolution of cybersecurity practices. They emphasize the significance of innovative solutions in fortifying digital ecosystems against emerging threats by offering a nuanced understanding of malicious activities across different environments.

7.1 Future Work

While this thesis lays the groundwork for distributed honeypot deployment and demonstrates its feasibility, several avenues for further research and development remain unexplored. The following tasks represent potential areas for future work:

- **Aggregate New Honeypot Images:** Continuously expand the repository of honeypot images to encompass a broader range of deceptive services and emulate diverse targets, enhancing detection capabilities across different threat landscapes.
- **Establish Secure Connection Between Clusters:** Develop robust protocols and mechanisms for securely interconnecting distributed honeypot clusters, ensuring the confidentiality, integrity, and availability of communication channels to prevent unauthorized access and data breaches.
- **Implement Classification for Non-sensitive Data Capture:** Integrate machine learning algorithms or rule-based classifiers to discern between sensitive and non-sensitive data captured by honeypots, enabling the selective retention and analysis of information while adhering to privacy and regulatory requirements.
- **Train Models Based on Collected Data:** Leverage the wealth of data collected by distributed honeypots to train machine learning models for anomaly detection, intrusion detection, and threat intelligence, empowering organizations to identify and mitigate emerging cyber threats.
- **Create Smart Control of Honeypots Based on External Data:** Develop dynamic control mechanisms that adjust the behavior and configuration of honeypots in real time based on contextual information gathered from external sources, such as threat intelligence feeds, network telemetry, and incident response systems.

- **Design Interface for Traffic Monitoring:** Design and implement a user-friendly interface for centralized monitoring and visualization of network traffic, allowing security analysts to gain insights into malicious activities, detect patterns, and respond promptly to evolving threats across distributed honeypot deployments.

By addressing these future tasks, researchers can further advance the capabilities and effectiveness of distributed honeypot infrastructures, bolstering cybersecurity defenses and enhancing resilience against evolving cyber threats.

Bibliography

[1] B. Aiyer, J. Caso, P. Russell, and M. Sorel. «New survey reveals $2 trillion market opportunity for cybersecurity technology and service providers». In: (Oct. 2022). URL: https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/cybersecurity/new-survey-reveals-2-trillion-dollar-market-opportunity-for-cybersecurity-technology-and-service-providers (cit. on p. 1).

[2] *Cost of a data breach 2023*. https://www.ibm.com/reports/data-breach. Accessed on: March 28, 2024 (cit. on p. 1).

[3] Cisco. *Cisco Annual Internet Report (2018–2023) White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed on: March 28, 2024. Jan. 2022 (cit. on p. 1).

[4] *T-Pot Version 22.04 released*. https://github.security.telekom.com/2022/04/honeypot-tpot-22.04-released.html. Accessed on: March 28, 2024. Apr. 2022 (cit. on p. 4).

[5] *What is a virtual machine (VM)?* https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine. Accessed on March 28, 2024 (cit. on p. 6).

[6] Docker. *What is a Container? | Docker*. https://www.docker.com/resources/what-container/. Accessed on March 28, 2024 (cit. on pp. 6, 7).

[7] Docker Documentation. *Overview of Docker Hub*. https://docs.docker.com/docker-hub/. Accessed on March 28, 2024. Jan. 2024 (cit. on p. 7).

[8] *Welcome to Cowrie's documentation! — cowrie 2.5.0 documentation*. https://cowrie.readthedocs.io/en/latest/index.html. Accessed on March 28, 2024 (cit. on p. 7).

[9] Docker Documentation. *Docker Compose overview*. https://docs.docker.com/compose/. Accessed on March 28, 2024. Jan. 2024 (cit. on p. 7).

[10] Docker Documentation. *Swarm mode overview*. https://docs.docker.com/engine/swarm/. Accessed on March 28, 2024. Feb. 2024 (cit. on p. 7).

[11] Docker Documentation. *How nodes work*. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/. Accessed on March 28, 2024. Feb. 2024 (cit. on p. 8).

[12] Kubernetes. *Kubernetes components*. https://kubernetes.io/docs/concepts/overview/components/. Accessed on March 28, 2024. Jan. 2024 (cit. on p. 8).

[13] Kubernetes. *Overview*. https://kubernetes.io/docs/concepts/overview/. Accessed on March 28, 2024 (cit. on p. 8).

[14] Kubernetes. *Cluster Architecture*. https://kubernetes.io/docs/concepts/architecture/. Accessed on March 28, 2024 (cit. on p. 9).

[15] IVPN. *PPTP vs IPSec IKEv2 vs OpenVPN vs WireGuard*. https://www.ivpn.net/pptp-vs-ipsec-ikev2-vs-openvpn-vs-wireguard/. Accessed on March 28, 2024 (cit. on p. 9).

[16] Rancher Labs. *Cloud native infrastructure*. https://www.rancher.com/categories/cloud-native-infrastructure. Accessed on March 28, 2024 (cit. on p. 13).

[17] Google Cloud. *What is Kubernetes? | Google Cloud*. https://cloud.google.com/learn/what-is-kubernetes. Accessed on March 28, 2024 (cit. on p. 13).

[18] Kubernetes. *Using Minikube to create a cluster*. https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/. Accessed on March 28, 2024. Dec. 2023 (cit. on p. 13).

[19] Toolify. *Choosing between k0s and k3s for Your Home Lab*. https://www.toolify.ai/gpts/choosing-between-k0s-and-k3s-for-your-home-lab-110856. Accessed on March 28, 2024. Nov. 2023 (cit. on p. 13).

[20] K3s. *Architecture | K3s*. https://docs.k3s.io/architecture. Accessed on March 28, 2024. Mar. 2024 (cit. on p. 14).

[21] Jason A. Donenfeld. *Performance - WireGuard*. https://www.wireguard.com/performance/. Accessed on March 28, 2024 (cit. on p. 15).