Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Nam Xuan Nguyen

Network isolation for Kubernetes hard multi-tenancy

Master's Thesis
Espoo, July 31, 2020

Supervisors:	Professor Tuomas Aura, Aalto University
	Professor Danilo Gligoroski, NTNU
Advisor:	Alireza Ranjbar, M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

ABSTRACT OF MASTER'S THESIS

Author:	Nam Xuan Nguyen		
Title: Network isolation for Kubernetes hard multi-tenancy			
Date:	July 31, 2020	**Pages:**	90
Major:	Security and Cloud Computing	**Code:**	SCI3084
Supervisors:	Professor Tuomas Aura Professor Danilo Gligoroski		
Advisor:	Alireza Ranjbar, M.Sc. (Tech.)		

Over the past decade, containerization is increasingly popular due to its advantages in performance compared to virtualization. The rise in the use of containers leads to the emergence of container orchestration tools. Kubernetes is one of the top widely used tools serving this purpose. One critical point in the design of this tool is that one cluster can only serve one tenant. As the number of Kubernetes users is continuously increasing, this model generates considerate management overheads and resource fragmentation to the cluster. As a result, multi-tenancy was introduced as an alternative model. However, the major problem of this approach is the isolation between tenants. This thesis aims to tackle this isolation issue. While many cluster resources need to be isolated, we concentrate on handling one crucial feature in Kubernetes hard multi-tenancy: Network isolation. Our solution for this problem is intended to work regardless of the implementation flexibility of the Kubernetes network. The solution can also pass most of our security tests. The remaining issues are not significant, and one of them is solvable. Besides, our performance experiments recorded that this solution generated delays in cluster activities. However, in most cases, this delay is noticeable but nevertheless acceptable. The proposed method can potentially be a part of real Kubernetes multi-tenant systems where network isolation is one of the essential requirements.

Keywords: Multi-tenancy, Kubernetes, container technology, network isolation, sidecar container, iptables

Language: English

Acknowledgements

I wish to express my deep gratitude to Professor Tuomas Aura, my main supervisor, for his valuable guidance and constructive suggestions, and for his review of my thesis report. I would also like to thank Alireza Ranibar, my advisor, for his enthusiastic encouragement, his useful advice, and for review of my report. I also like to say thank to Jan Melen, my supervisor in Ericsson, for bringing me to EST team, introducing me this interesting topic, and for his support for my thesis.

I would like to extend my thanks to Maël Kimmerlin, my colleague, for his valuable suggestions, thoughtful explanations, and useful critiques. I would also like to thank all EST team members for your advice and encouragement, which kept my motivation high, so I could finish my thesis.

I also wish to thank Professor Danilo Gligoroski, my secondary professor in NTNU, for his support for my thesis.

Finally, I wish to thank the SECCLO coordinators for building a wonderful master's program and bringing me to it.

Espoo, July 31, 2020

Nam Xuan Nguyen

Abbreviations and Acronyms

PoC	Proof of Concept
ABAC	Attribute Based Access Control
AMQP	Advanced Message Queuing Protocol
RBAC	Role-based Access Control
NoP	Number of Pods
Mb/s	Megabits per second
ms	Millisecond
RTT	Round Trip Time
TCP	Transmission Control Protoco
IP	Internet Protocol
TLS	Transport Layer Security
MAC	Media Access Control
DNS	Domain Name System
NAT	Network Address Translation
SNAT	Source Network Address Translation
CA	Certification Authority
API	Application Programming Interface
CNI	Container Networking Interface
SR-IOV	Single-Root Input/Output Virtualization
IPIP	IP in IP
PCIe	Peripheral Component Interconnect express
VFs	Virtual Functions
PFs	Physical Function
YAML	YAML Ain't Markup Language (recursive acronym)
JSON	JavaScript Object Notation
CRD	Custom Resources Definition
OS	Operating System
RAM	Random-Access Memory

Contents

Chapter 1

Introduction

In the last few years, containers have become a widely used method for quick, cheap, and reliable application deployments. Compared to virtual machines, containers are more lightweight with higher performance [59]. A bare-metal host can run ten times as many containers as virtual machines. As containers become more and more popular, the application paradigm gradually moves from monolithic to microservice architecture. In this model, an application is a collection of independently deployable containers acting as portable components, and they communicate to each other via simple network protocols, including Hypertext Transfer Protocol (HTTP), Advanced Message Queuing Protocol (AMQP), and Transmission Control Protocol (TCP) [61]. The microservice model has been changing the software development and testing procedures as well as the methods to deliver product to customers. As the need of microservice application development continuously increases, we need a container orchestration tool that automatically manages a large number of containers. Kubernetes [26] is one of the most widely used tools for this purpose. It allows us to quickly provision, create, scale, and delete up to 300000 [6] containers simultaneously. Especially, Kubernetes can form a cluster, a set of nodes provisioned by Kubernetes. Containers inside these nodes can easily interact with each other as if they were in the same node.

The increasing number of Kubernetes users leads to a scalability problem. One Kubernetes cluster was originally designed to serve one tenant, in which a tenant is a group of users that absolutely trust each other, such as members in a team. The involvement of multiple tenants in a cluster leads to the issue that a misbehaving tenant can occupy all the cluster resources or compromise the other tenants' applications. One workaround of this problem is to apply the single-tenant-per-cluster approach. Figure 1.1 shows a cluster architecture following this approach.

In this architecture, on a pool of bare metal machines, a virtualization

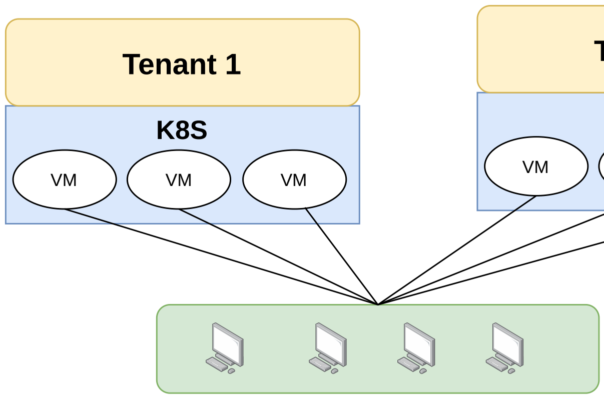

Figure 1.1: Single tenant per cluster architecture

platform is deployed to monitor and distribute computing, storage, and networking resources and ensure that they are utilized securely and efficiently. The virtualization hypervisor can spawn several virtual machines, and a Kubernetes cluster runs on the top of them. As a result, each tenant can have the full permission to access one dedicated Kubernetes cluster, which is isolated from other tenants.

In this model, the virtualization platform acts as a management system that provides the isolation between tenants. However, it also generates overhead that we expected to avoid when replacing virtualization by containerization. Besides, deploying a dedicated cluster for each tenant can lead to resource fragmentation. Although the single-tenant-per-cluster model can solve the tenant isolation problem, it hinders the overall performance of the physical cluster. To enjoy the benefit of the container technology, we need to deploy a Kubernetes cluster directly to the physical hosts, as shown in Figure 1.2.

1.1 Problem Statement

With the architecture above, it is essential to find another method for isolation in multi-tenancy environments. Multi-tenancy can provide various levels of isolation and management. Soft multi-tenancy solutions can be employed if tenants can partially trust each other, e.g., tenants are teams or departments in a company. Hard multi-tenancy approaches are required when each tenant can potentially be an attacker, and they cannot trust each other. Section 2.3.2 describes these concepts in detail. Hard multi-tenancy attracts more attention of researchers as well as engineers due to its advantages. The

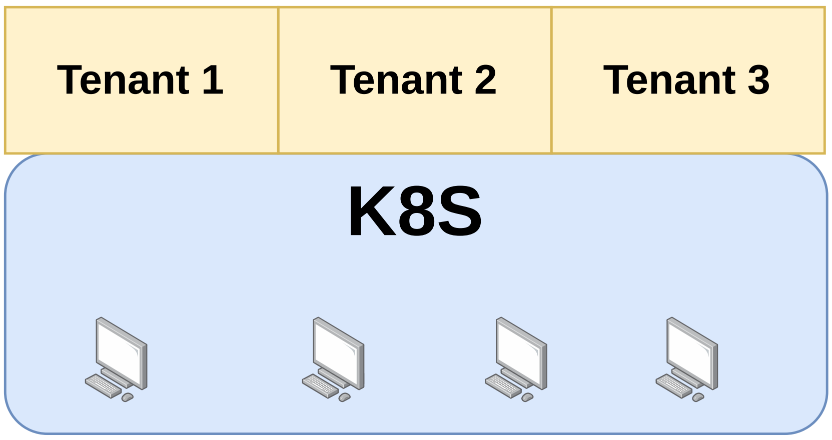

Figure 1.2: Kubernetes on bare metal hosts

main advantage is that this approach can serve more tenants from various backgrounds and organizations. The tenant generally prefers to join an instance of a hard multi-tenant cluster to avoid security and isolation concerns. Besides, solving the hard multi-tenancy problem in containerization means moving one step further to improve container security, which is one critical point for the success of this technology.

To compare between soft and hard multi-tenancy, we can consider this example. To provide soft multi-tenancy, it is possible to create a management interface that only authenticated tenants can access and interact with their resources in a shared cluster. However, this interface offers no actual shared resources isolation between tenants, such as shared memory or shared network, thus allowing a malicious tenant to compromise other tenants' resources. To handle this problem, it is critical to apply harder, more robust mechanisms that can isolate the resources used by the tenants. This is the reason that we need hard multi-tenancy.

While hard multi-tenancy requires hard isolation in workload, storage, memory, and network, in this thesis, we focus on network isolation, which is a critical requirement in multi-tenant environments.

The main network isolation issue in Kubernetes is due to the design of the Kubernetes network (see Section 2.2). By default, a pod (see Section 2.1.1) can connect to any other pods with no restrictions, even if they are in different namespaces. This thesis intents to solve this problem, blocking cross-namespace pod communication. Assume that each tenant is assigned to a namespace, the restriction of cross-namespace communication leads to the network isolation between tenants.

To sum up, this thesis proposes a solution for blocking pod-to-pod communication between the tenants in a multi-tenant Kubernetes cluster.

1.2 Structure

The rest of this thesis is organized as follows: Chapter 2 gives background knowledge about Kubernetes. It also introduces more detail about the multi-tenancy problem and related work. Chapter 3 proposes our ideas to solve the network isolation problem. The main solution in this chapter is to modify the inner pod firewall. Chapter 4 introduces an implementation of the solution from the previous chapter. Chapter 5 explains the test environment used to evaluate our solution. Chapter 6 and Chapter 7 discuss the methods and results of our security and performance evaluation, respectively. Finally, Chapter 8 concludes the thesis and discusses the future work.

Chapter 2

Background and Related Work

2.1 Kubernetes

Kubernetes is an open-source orchestrator designed for managing containerized applications. It supports a range of automatic operations, such as finding a working node with available resources for running a workload, configuring network, and maintaining the desired status of the cluster [47]. It originates from Google before being maintained and developed by Cloud Native Computing Foundation (CNCF). Kubernetes is predecessor of Borg [39], a Google internal container-oriented cluster-management system.

Kubernetes possesses a declarative nature. It ensures that every object in the cluster achieves and retains a desired state declared by the user with the Kubernetes native application programming interface (API). The full list of Kubernetes objects can be found in [12]. The objects which are related to this thesis are described in the following section.

2.1.1 Kubernetes Objects

(i) **Pods:** To orchestrate containers, Kubernetes aggregates multiple containers into a pod, which is the basic scheduling unit [58]. Each pod has its unique internet protocol (IP) address, and all pods in the cluster can connect to it even if they are located in different physical hosts [36]. Inside a pod, containers share the same storage volume and network. They are always located in the same physical host and can communicate with each other on the localhost. However, containers in one pod are unable to address containers in another pod [25] directly. They need to use the pod IP addresses. Moreover, an application service should not be accessed via the pod IP addresses. The reason is that Kubernetes pod is ephemeral and can die anytime, thus being unreliable to become a network endpoint. For example, in a cluster, ten pods run Nginx servers, and a client pod demands this a HTTP service. If the client pod is set to connect to one specific server pod continually, the communication will be broken when this server crashes or is terminated. Therefore, the concept of Kubernetes service was introduced to handle this problem.

(ii) **Services:** A service is a mechanism for forwarding traffic from client pods to a specific set of server pods. In the example above, the client pod can send packets to the IP address of the service. Then, the packets are randomly (or based on a pre-defined scheduling policy) sent to one of ten server pods, and a connection is established between the two pods. If some of these pods die, the service can simply forward traffic to live pods, and thus the operation is not interrupted.

(iii) **Namespaces:** Namespace is a mechanism that allows binding a group of resources to a name. Namespaces are applied in cases where the Kubernetes cluster has multiple users divided into different teams or departments. They can be exploited for the multi-tenancy purpose. Objects in the same namespace are required unique names, while objects in different namespaces can have the same name [21]. Namespaces are also exploited to enforce resource quotas, access control, and isolation for users. In terms of multi-tenancy, each tenant can be assigned to one namespace, and any objects belonging to this tenant will bind to this namespace.

(iv) **Network Policy:** A Network policy object is a set of rules that allows or blocks the communication between a group of pods to a network endpoint object, including pods or services. For example, one network policy can deny the access of all pods in the namespace *Alice* to pods in the namespace *Bob*, but grant the access to services in the namespace *Carol*. In terms of Kubernetes multi-tenancy, network policy can be used to solve the network isolation problem between different tenants. However, since network policy is a namespace-scoped object [22], it can be controlled by the tenant, leading to security risks for isolation. This problem is explained in more detail in section 2.3.2.

2.1.2 Authentication and Authorization

For authentication, Kubernetes offers two categories of users: service account [13] and normal user [5]. While a service account enables pods to be authenticated when connecting to the API server, the normal user provides a method to authenticate a human user, e.g., using *kubectl*. The API server can authenticate a normal user by several methods, including static token file, bootstrap tokens, password [5], and X.509 [49] client certificates. The last method provides strong security and undeniable convenience, which is why we select it as the primary authentication method in our experiments. It requires each user to possess a pair of a public key and a private key. The user then requests a certificate for the public key. Next, the API server issues and signs a certificate for the client public key using its own key pair. After that, whenever the user interacts with the API server, it is required to establish a secure transport layer security (TLS) connection. The user needs the client certificate and the private key to be authenticated by the server, and it needs the certification authority (CA) certificate (a self-signed certificate issued by the API server) to authenticate the API server.

For authorization, Kubernetes supports several widely known mechanisms that can be used to grant permission for certain actions in the cluster using the Kubernetes API. These mechanisms include role-based access control (RBAC) [43], attribute-based access control (ABAC) [75], Node [31], and Webhook [32]. Among these mechanisms, only ABAC and RBAC provide configurable policies. Although ABAC is a powerful tool that allows administrators to fully customize their authorizer, it is difficult to employ and manage. In contrast, RBAC is an easy-to-follow concept, and most development efforts of the Kubernetes community are toward this mechanism. For that reason, the RBAC approach is employed in this thesis.

In RBAC, a role is a set of permissions that allow particular operations. When a user is assigned to a role, this user is granted all the permissions included in that role. Therefore, two critical concepts in the RBAC model are the role and the relationship between users and roles. In Kubernetes, they are represented by two objects: *Role* and *RoleBinding*.

(i) **Role:** A *role* [43] object is a set of permissions consisting of only "allow" rules because, by default, everything is forbidden. *Role* is namespace-scoped objects, which means they can only include permissions in one particular namespace.

(ii) **RoleBinding:** A *role binding* object binds a set of subjects, such as service accounts, groups, and users, to a list of roles granted to these subjects. *Role binding* objects can refer to *role* objects within the same namespace. An overview of the relation between *role*, *role binding* and user can be found in Figure 2.1.

    A user can be included in many role binding objects, and a role binding object can include many users. Similarly, a role binding object involves

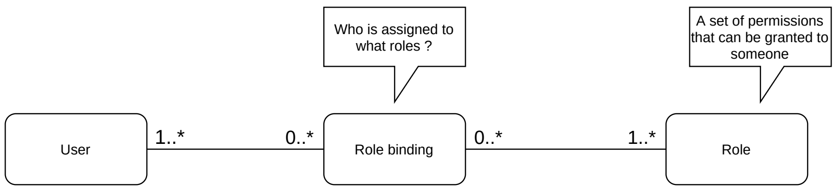
Who is assigned to what roles ?
A set of permissions that can be granted to someone
User
1..*
0..*
Role binding
0..*
1..*
Role

Figure 2.1: The relation between role, role binding and user

an unlimited number of role objects, and vice versa.

2.1.3 Kubernetes Architecture

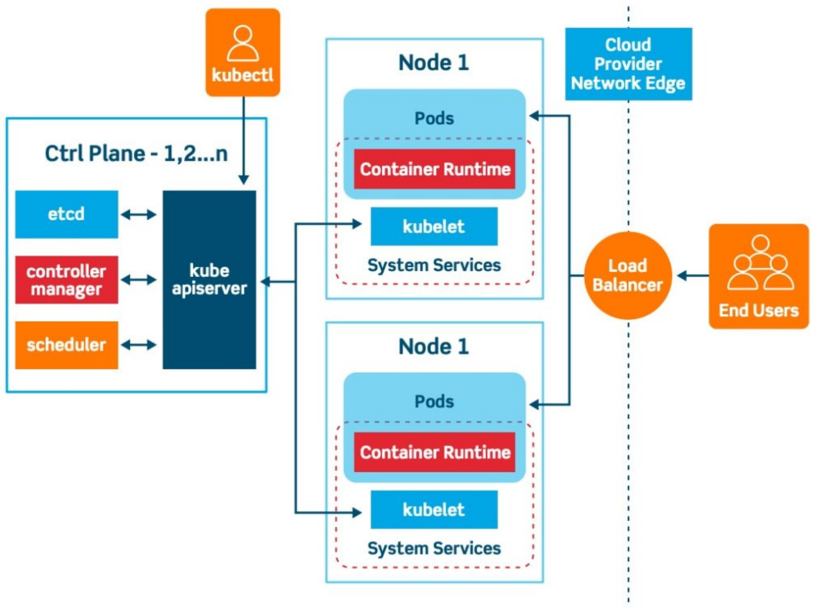

Figure 2.2: Kubernetes cluster architecture

The Figure 2.2 describes an example Kubernetes architecture of a cluster with three nodes, including two worker nodes for handling workloads and one control plane node. The control plane ensures that every object in the cluster achieves its declared state. The control plane consists of the following main components:

- **Kubernetes Api server** receives requests and sends responses to the user. By communicating with the Kubernetes Api server using the

Kubernetes API, users can load their pre-defined object configurations to the cluster, including their choices of the application images, number of replicas, CPU power and storage consumed by each container [12]. Besides, this server acts as the cluster gateway. More precisely, this API server is the only endpoint to send commands and update cluster information. For this reason, it is expected to be accessed from the outside.

- **Etcd** is a key-value database that stores the records of all Kubernetes objects, such as the current state and the configuration. Any declarative configuration sent from users is persisted in etcd.
- **Kubernetes controller manager** controls the state of controllers. A controller is a logical entity that maintains the status of one object, and the controller manager is the daemon that implemented all the controllers.
- **Scheduler** is responsible for allocating containers to worker nodes. The scheduler guarantees that workloads are distributed within the pre-defined constraints.
- **Kubelet** receives a set of pod specifications from the Kubernetes API server and ensures that containers running on the worker nodes are healthy and follow the pod specifications.

To interact with the cluster, cluster users, such as administrators or developers, need to communicate with the API server using the Kubernetes API [29]. This API is actually a HTTP API, and thus users can use a HTTP client, e.g., curl or wget, to send API requests. However, for convenience and enhancing productivity, a dedicated tool, *kubectl* [24], is introduced for controlling the Kubernetes cluster from a client endpoint. This useful tool allows users to perform all possible actions, including to create, delete, get, describe, and apply. The details of the actions can be found in [24].

2.2 Kubernetes Network

Kubernetes network provides an easy-to-use and convenient mechanism for pod communication. In this network, every pod has a unique IP address. The network design ensures that pod IP addresses are reachable by any pod on any host in the cluster without the use of network address translation (NAT). These IP addresses are also accessible by agents running on nodes, including kubelet or system daemons [10]. This network model also handles the communication between containers in a pod so that they can reach each other on the localhost. In a pod, containers stay in the same Linux network namespace [73], and share the same IP address (pod IP address) and port space. More precisely, one container can connect to a port on another container in the same pod on the localhost, which is the same as two processes on the same host open ports and communicate on the loopback interface [34].

Although the Kubernetes network model requires specific network features, it is open to a variety of implementations [10]. The Kubernetes community provides a specification to explain their expectations of the pod communication. This specification is called *Container Network Interface* (CNI) [11]. CNI provides the requirements of the Kubernetes network, defines an interface to integrate a network implementation to a cluster, and includes a library used as the API for network implementation. The actual design and implementation of the Kubernetes network come from third-party companies, and their products are called *network plugin* for Kubernetes. They can also be called by the word “CNI”. Since Kubernetes constantly gains popularity with cloud developers and providers, many organizations provide their CNIs to the public as open-source projects. The CNI implementations vary from L2 and L3 overlay networks to reconfiguring the underlay network. Some widely used CNIs include Calico, Flannel, Macvlan, Cilium, SR-IOV, Mutus, and Weave [37].

Kubernetes network model is flat. That means any pod can connect to any other pods with no restriction. This also applies to the communication between two pods from different Kubernetes namespaces or different nodes (see Figure 2.3). This results in a critical security issue when tenants sharing the same Kubernetes cluster cannot trust each other. Unfortunately, the issue exists in all CNIs.

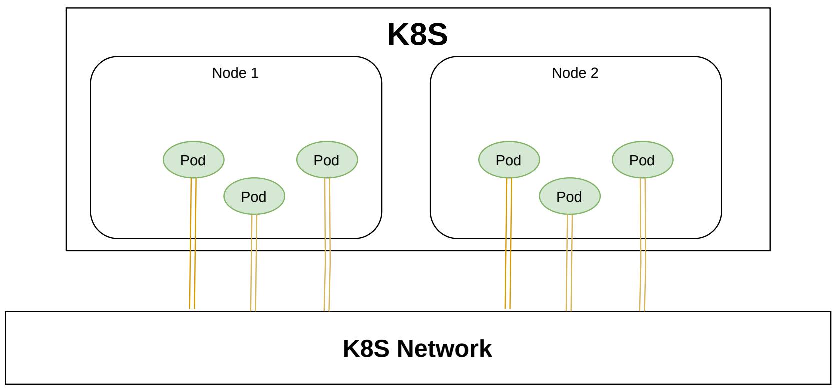

Figure 2.3: Overview of the Kubernetes network

As mentioned above, CNIs implement the Kubernetes network in various approaches. That means the path of a packet moving from pod to pod varies between different CNIs. In the next subsection, we introduce more details of the Linux network namespace before describing the widely known CNIs.

2.2.1 Linux Network Namespace

Linux network namespace [18] is a Linux mechanism that allows a process or a set of processes to stay inside a logical copy of the network stack. More precisely, processes inside a network namespace possess dedicated routing tables, firewall rules, network devices, and port space. In terms of Kubernetes, each pod, including all processes in containers in the pod, is inside a Linux network namespace. That means each pod has its own iptables, an internal layer 3 firewall, leading to our main solution discussed in Section 3.3.

Introducing the Linux network namespace concepts can lead to confusion while we also have the Kubernetes namespace. In fact, these two concepts are independent. Besides, no relation exists between the Linux network namespace and Kubernetes network policies. The network policies can control the connections between pods, while the Linux network namespace provides a dedicated network stack to all the containers inside a pod.

2.2.2 Calico

Calico network plugin provides a layer 3 network solution. As mentioned above, each pod is inside its own Linux network namespace. This CNI sets up a virtual Ethernet (veth) between the pod network namespace and the default network namespace of the host, in which physical network interfaces, such as **eth0**, exist. A veth is a pair of virtual interfaces, and in this case, one interface is in the pod network namespace, and the other is in the default network namespace. Any packet that come to one interface will immediately go out from the other interface. Figure 2.4 shows an example of pod-to-pod communication in the Calico network. In this example, pod A is communicating with pod B.

1. Pod A sends packets to *veth0A*.

2. The packets go out from *veth1A* in the default network namespace. The kernel in the host uses a routing tables rule to decide the destination of the packets.

3. The packets are routed to *veth1B*.

4. The packets go out from *veth0B* and reach the pod B.

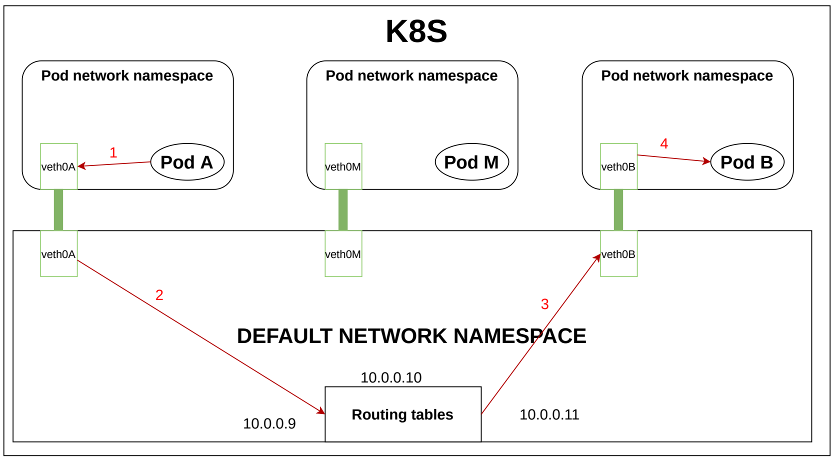

Figure 2.4: Pod-to-pod communication in Calico

To allow pod-to-pod communication between different nodes, Calico employs IP in IP (IPIP) protocol [52]. In the IPIP tunnel, the sender endpoint encapsulates the original packet in an IP packet, sends it to the receiver endpoint. The receiver endpoint then decapsulates this packet and passes it to the handler. Figure 2.5 explains the whole process of the communication between pod A in node 1 and pod B in node 2.

1. After the packets come to the default network namespace, the kernel scans the routing tables in the default network namespace and routes the packets to *ipip0*. By default, any packets coming to this interface are sent to the specific daemon that is responsible for IPIP tunnel.

2. The IPIP daemon encapsulates the packets in an IP packet. The new packets have the IP address of node 1 as the source address, and the IP address of node 2 as the destination address. These packets are then sent on the node network, the underlay network that connects the nodes.

3. The kernel in node 2 decapsulates and routes the packets to the destination pod.

To sum up, Calico CNI is a network L3 solution that uses the routing tables to allow the pods to reach each other in one node and employ IPIP tunnel to connect pods in different nodes.

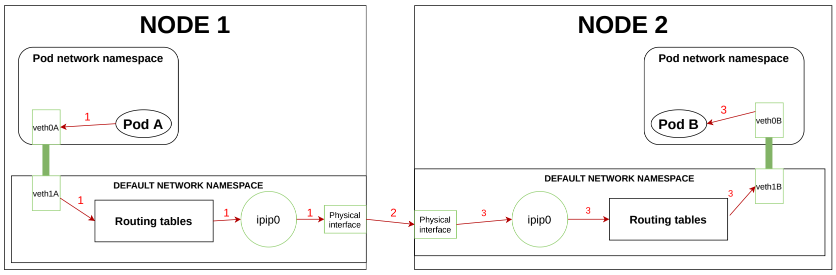

Figure 2.5: Pods in different nodes communicating in Calico

In addition, Calico CNI supports Kubernetes network policy. The CNI provides a daemon, named Felix [7], that runs on every node of the cluster. It translates the network policies to iptables rules. In other words, Felix implements the network policies using iptables on each host.

2.2.3 Flannel

Flannel CNI [3] is a network L2 solution. In this CNI, pod network namespace also connects to the host network namespace via *veth*. However, instead of using the routing tables to guide packets to their destination pods, Flannel adopts Docker bridges [53] for pod-to-pod communication in one node. More precisely, in one node, all pods connect to a common bridge (*docker0* by default). During the pod creation process, the *veth* virtual interface in the host network namespace of this pod is attached to that bridge. Therefore, with Flannel CNI, pods in a node communicate on layer 2 of the network.

Flannel also utilizes network tunneling to operate the connection between pods in different hosts. This CNI provides three options for the encapsulation and packet handling methods [41]. These options are *VXLAN*, *host-gw*, and *UDP*. According to [41], it is recommended to use *VXLAN* as this is the default option, *host-gw* is for advanced users, and *UDP* should be used only for debugging purposes. Since the details of the encapsulation techniques are not related to other parts of this thesis, we would refer to [41] for more details.

One shortcoming of this CNI is that it does not support the Kubernetes network policy.

2.2.4 Canal

Canal CNI [17] is a hybrid CNI that combines Calico for policy and Flannel for networking. In detail, Canal uses Flannel to provide Kubernetes network, and it also provides a daemon running on each host (similar to Felix) to implement the Kubernetes network policies. Therefore, this CNI is a solution that overcomes the lack of support for the Kubernetes network policy support of Flannel CNI.

2.2.5 Macvlan

Although Macvlan is a CNI, it is not designed for pod-to-pod communication. Instead, this CNI is usually used to provide a new network that operates alongside a primary network provided by Calico or Flannel. This other network allows a specific set of pods to perform particular operations which require a dedicated pod network. Macvlan [40] is a mechanism that will enable more than one media access control (MAC) addresses to exist in one physical network interface. In detail, Macvlan allows users to configure sub-interfaces of the physical one, and each subinterface is assigned to a unique MAC address. Macvlan CNI binds each pod to a sub-interface, so that from the outside, each pod appears to have a real MAC address and can communicate on network layer 2. Figure 2.6 shows the Macvlan architecture.

2.2.6 SR-IOV

Single-root input/output virtualization (SR-IOV) allows a pod to directly bind to a virtual interface and appear to possess a real network interface and a real MAC address. SR-IOV is a specification that enables a peripheral component interconnect express (PCIe) device to be split into several separate instances [42]. Since the network interface cards follow PCIe standard, they are also integrated SR-IOV implementation. In this implementation, there are two types of functions that are introduced, namely *Physical Functions* (PFs) and *Virtual Functions* (VFs). PFs provide the ability to fully control the network card, including resource allocation and input/output management. In other words, entities having access to PFs cannot only read and send data, but also assign VFs instances to other objects.

On the other hand, a VF can only manipulate incoming and outgoing data from its instance. According to [57], in one single PCIe device, there can be up to 256 VF instances. These instances can be assigned to SR-IOV Kubernetes pods, one instance to each pod. In this case, pods can “think” that other pods have real, physical network interfaces with unique MAC addresses, and the traffic separation is enforced by the SR-IOV implementation in the network card.

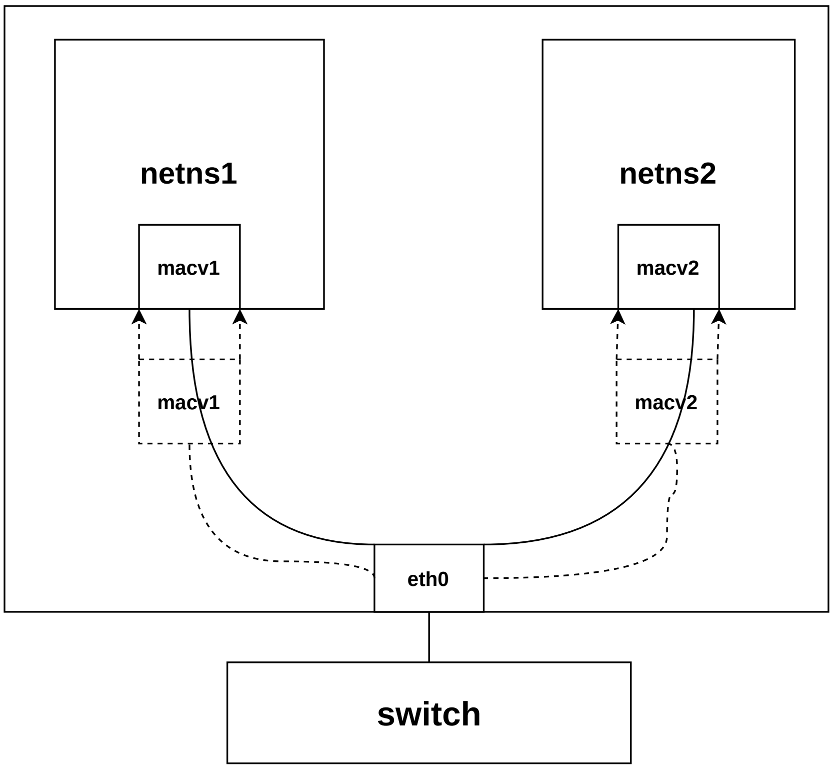

Figure 2.6: Macvlan Architecture

2.3 Kubernetes Multi-Tenancy and Related Work

Multi-tenancy is a mechanism that allows different customers to access one software instance [54]. This mechanism saves costs for service providers by leveraging the resource utilization and scalability of cloud-native solutions. In a multi-tenant application, a tenant can be defined as a closed group of users, such as team members, or colleagues in a company. In Kubernetes, multi-tenant architectures enable a cluster to serve users from various backgrounds. For example, they can be team members from different departments in a company, or be teams from separate companies. Although multi-tenant clusters inevitably reduce the cost of management overhead that exists in the single-tenant clusters, they also come at a price.

Kubernetes, unfortunately, is originally designed to serve one tenant per cluster only. Therefore, alongside the cost-effectiveness, Kubernetes multi-tenancy introduces a new class of management and security issues. Issues related to management come from the difficulty of fairly distributing resources to every tenant while holding the resource fragmentation at an acceptable level. In other words, when several tenants share a cluster with a limited amount of available resources, it is challenging to ensure that one tenant cannot retain a large share at the resources, which hinders the other tenants' activities. Resource quotas [28], which is a tool for administrators to limit CPU and memory aggregate consumption per Kubernetes namespace, can be employed to handle this fair resource allocation problem. Besides, in [74], Xu et al. introduced a design for network management in Kubernetes. This work enables the Kubernetes cluster to manage the network bandwidth as a resource, which is similar to CPU and memory usage, and the administrator can set a bandwidth quota for each tenant.

Another critical problem of Kubernetes multi-tenancy is security. The main security problem in any multi-tenancy architecture is the isolation between tenants. When different customers share a cluster, several resources need to be isolated, such as computation, memory, storage, and network. The level of necessary isolation may depend on the degree of trust between tenants, which decides the type of multi-tenancy architecture that needs to be used. *Soft multi-tenancy* architecture is suitable in the case that tenants can partially trust each other, e.g., tenants are different departments in a company. On the other hand, *hard multi-tenancy* can be adopted in the situation that tenants cannot trust others, and each tenant is a potential attacker. Hard multi-tenancy is preferable when tenants from very different backgrounds share the same cluster, e.g., customers from competing companies or different countries. Open systems, such as a Kubernetes operator that allows anyone to sign up as its customer, also requires stranger isolation than closed systems with well-known tenants.

This thesis focuses on network isolation, one crucial requirement in hard multi-tenant systems. The network isolation problem stems from the fact that Kubernetes clusters are not prepared to operate in multi-tenant environments, and thus there is no restriction in communication between two pods even if they belong to different tenants.

2.3.1 Soft Multi-Tenancy

There are known methods that can be exploited to implement network isolation in soft multi-tenancy. Kubernetes network policy can be used to enforce the isolation. The administrator can assign each tenant to a Kubernetes namespace, then create a network policy object for each of these namespaces, and this object can block any traffic from other Kubernetes namespaces. An example of the YAML file (the description of a Kubernetes object) of this network policy object can be found in Figure 2.7. Besides, Istio [19] provides a multi-tenancy solution using microservices and a service mesh. Microservices [68] are loosely coupled services that work together to form an application. Each microservice is an independently deployable program acting as a component of the application. These microservices communicate with each other using a lightweight and widely used protocol, e.g., *HTTP*. The use of microservices reduces the complexity of the software design and implementation by breaking it into smaller, manageable units. However, as the number of microservices can grow significantly, the infra-connection and the internal data transferring between them can be increasingly complex. To that end, service mesh [68] is introduced to assist the management of these infra-connections. This management with Istio service mesh can be used to provide multi-tenancy on Kubernetes. Each pod can be treated as a microservice in Istio, and each tenant can be allocated an independent service mesh.

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: secondary
  name: deny-from-other-namespaces
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
```

Figure 2.7: Deny-all-traffic-from-other-namespaces network policy

2.3.2 Hard Multi-Tenancy

For hard multi-tenancy, as required for absolute isolation, network policy and Istio service mesh may not fulfill this requirement. Network policy is a namespace-scoped object, meaning that it needs to bind to a Kubernetes namespace and be under the control of the tenant who can access that namespace. Since the tenants also want to create other network policy objects for their use, it is likely that their custom network policies can unintentionally overwrite the administrator's default policies, thereby leading to the threat that other tenants can connect to this tenant. On the other hand, the solution from Istio can only solve the management problem. In detail, this solution provides a management interface that only allows a tenant to access their resources and prevent them from interacting with other tenants.

However, because the service mesh runs on top of the Kubernetes network, it is impossible to isolate the network from the service mesh level. More precisely, Istio adds a new overlay network on top of the default Kubernetes network, and it allows the administrators of the cluster to control this overlay network by issuing rules, and thus they can restrict the communication between tenants. However, this restriction can be applied only for the overlay network. While the pods and the other network endpoints in the cluster using Istio are expected to communicate via the overlay network, a malicious pod can intentionally connect to another pod of the other tenants using the default Kubernetes network, because Istio provides no restriction for that actions. Therefore, although Istio may solve the soft multi-tenancy problem, a stronger mechanism is required to handle the network isolation problem in multi-tenancy environments.

Currently, there is no standard method to solve the Kubernetes hard multi-tenancy problem. Cluster administrators tackle this problem with their approaches. However, there is an open-source community that is actively devoting efforts toward this problem. This group is *SIG-Multitenancy* [20]. One interesting project of this group is *Virtual Cluster*. The idea of this project is to deploy virtual Kubernetes clusters inside a real Kubernetes cluster. Each tenant can access to their virtual cluster with full functionality of the real cluster. However, this project is in an early implementation stage. Another large project is *Hierarchical Namespace CRD*, where CRD stands for custom resources definition. Since each tenant can themselves create Kubernetes namespaces for their own use, this project aims to handle the problem when the namespaces of different tenants are duplicate. One Kubernetes namespace is required to be unique in one Kubernetes cluster. However, the network isolation issue remains unsolved, and this master thesis will, for the first time, provide a robust academic solution for this isolation problem.

Chapter 3

Network Isolation Solution

In this chapter, we discuss and propose solutions for the network isolation problem in Kubernetes multi-tenant cluster. They include solutions for each specific CNI, and a solution that can work on any CNIs. The latter is our primary solution that is implemented and evaluated in the next chapters.

3.1 One Kubernetes Namespace per Tenant

In this thesis, we assign each tenant a Kubernetes namespace. This approach allows the Kubernetes cluster to acknowledge which resources belong to which tenant. For example, a pod inside the namespace Alice should belong to tenant Alice. However, the cluster has no default mechanism to isolate the network between two Kubernetes namespaces. That is the main problem that we aim to solve using the solutions in the next sections.

3.2 CNI-Specific Solutions

3.2.1 Calico

As mentioned above, Kubernetes network policy can be exploited to provide network isolation in the case of soft multi-tenancy only. It cannot satisfy the requirement of hard multi-tenant systems because tenants can modify, delete, or overwrite the network policy objects created by the administrator for isolation purpose. Calico CNI, fortunately, introduces a variant of Kubernetes network policy, the global network policy (GNP) [16], which remedies this shortcoming. The GNP functions similarly with the vanilla network policy, except that it is a cluster-scoped object. Tenants are unable to edit or delete these objects since the administrator has created them, thus avoiding any accidental or malicious action by the tenants that impairs the network isolation.

Figure 3.2 shows an example of a GNP that denies TCP traffic coming from outside the Kubernetes namespace. In this example, the pods in the namespace Alice are allocated to the IP range **10.0.0.0/28**. This GNP object issues two ingress rules (ingress rule is applied for incoming traffic). The first rule enables the TCP inter-connections between pods in the namespace **Alice**. The second rule blocks the communications from outer pods to the inner pods. The parameter **protocol** can be utilized to choose other network protocols that the rule enforces on.

We experimented with checking the security of this approach. It actually provides the network isolation between pods in different Kubernetes namespaces. However, one critical point is that tenants can overwrite these GNP objects by issuing vanilla network policies. Figure 3.1 indicates a network policy that can overwrite the GNP in the Figure 3.2. This policy permits any incoming traffic from **Bob**, thus disabling the protection from the GNP rule.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: overwrite
  namespace: Alice
spec:
  podSelector: {}
  ingress:
    - to:
      - namespaceSelector:
          matchLabels:
            name: Alice
      - namespaceSelector:
          matchLabels:
            name: Bob
  policyTypes:
  - Ingress
```

Figure 3.1: A network policy that overwrites the default GNP

To solve this problem, we can add the parameter **order** to the description of the GNP object. This parameter defines the order of priority of the GNP object. If two GNP objects issue rules that conflict with each other, Calico enforces the rules from the higher-priority object. More precisely, to enforce a network policy rule as well as a GNP rule, Calico CNI deploys a bunch of iptables rules on the worker nodes. These iptables rules directly perform

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: isolation-Alice-ns
spec:
  selector: projectcalico.org/namespace == "Alice"
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    metadata:
      annotations:
        from: Alice
        to: itself
    protocol: TCP
    source:
        nets: ['10.0.0.0/28']

  - action: Deny
    metadata:
        annotations:
          from: other_namespace
          to: Alice
    protocol: TCP
    source:
      notNets: ['10.0.0.0/28']
```

Figure 3.2: An example of Global Network Policy

network operations, including allowing, blocking, or modifying the traffic in the cluster. The rules corresponding to the higher priority GNP objects are placed in higher order in iptables, which allows them to take effect. Although vanilla Kubernetes network policy objects do not provide the parameter **order**, Calico considers that their **order** value is 1000, and this value cannot be changed by anyone. As a result, setting the order of GNP to a value that is smaller than 1000 (in the example in Figure 3.3: 800) prevents efforts to overwrite the rules issued by this object.

With the use of **order** parameter, GNP objects can be exploited to provide a network isolation solution for hard multi-tenancy. The benefit of this approach is that GNP is fully configurable, enabling the administrator to adopt the isolation policy to fulfill customer requirements. However, this solution is limited to clusters using Calico CNI.

```yaml
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: isolation-Alice-ns
spec:
  order: 800
  selector: projectcalico.org/namespace == "Alice"
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    metadata:
      annotations:
        from: Alice
        to: itself
    protocol: TCP
    source:
        nets: ['10.0.0.0/28']

  - action: Deny
    metadata:
        annotations:
          from: other_namespace
          to: Alice
    protocol: TCP
    source:
      notNets: ['10.0.0.0/28']
```

Figure 3.3: An example of Global Network Policy with **order** parameter

3.2.2 Flannel

Flannel CNI does not support the Kubernetes network policy [22]. If we wish to use Flannel as the only CNI in the cluster, it cannot enforce network isolation using the network policy. To overcome this shortcoming, Flannel CNI provides an ability to combine with Calico CNI, which becomes Canal 2.2.4. Canal can also support GNP. Thus, the same solution for Calico can be applied to Canal to handle the network isolation issue. This approach in Canal offers the same advantages as in Calico. However, the drawback of this approach is that Flannel CNI alone is unable to support this solution.

3.2.3 Macvlan and SR-IOV

Unlike the two CNIs above, Macvlan and SR-IOV are not used for pod-to-pod communication. These CNIs are employed to deploy dedicated networks for particular purposes. Even so, pods in these networks can connect to other pods in other networks in the cluster. Consequently, network isolation is also necessary for networks using Macvlan and SR-IOV CNIs. However, the solution for Calico and Flannel cannot be applied to these two CNIs. The first reason is that these CNIs do not support network policy. The second is that, these CNIs implement the network on Layer 2 (Macvlan) or the layer of PCIe network cards (SR-IOV). In this case, traffic between pods cannot go through iptables that processes traffic on network layer 3. Fortunately, the next section describes a solution that can work on these CNIs.

3.3 CNI-Independent Solution

Adopting a solution that works on only one CNI may become a problem when a cluster integrates more than one CNI [51]. Therefore, it is preferred to apply a solution running on whatever CNIs we use to provide the Kubernetes network. To achieve that, we need to design our solution based on common elements shared between all CNIs.

As mentioned in Section 2.2, the Kubernetes pod is always inside a Linux network namespace (see Section 2.2.1). A network namespace possesses a dedicated port space, routing table, and iptables firewall. It can also be assigned to a dedicated IP address, which is different from the host IP address. Using network namespace is required by Kubernetes design regardless of any CNIs used to provide the cluster network. As a result, if a network isolation solution utilizes the Linux network namespace of the pod, it can apply to any CNIs. This is the critical point that leads to the primary solution introduced in this thesis.

Network isolation can be achieved by setting up a firewall inside a pod network namespace. Two resources that are suitable for these tasks are iptables and routing tables [44]. Routing tables can manipulate network routing, while iptables has powerful capabilities to control network traffic coming in and out from the network namespace. In this thesis, we select iptables as the main tool to implement the network isolation since it can easily block a packet if it matches a rule defined by the network administrator. In other words, iptables inside the pod network namespace is selected as the internal firewall used to provide network isolation. Note that the these iptables are separate from the iptables on the host network namespace. Therefore, even with the CNIs implementing network on a lower layer than layer 3, such as SR-IOV or MACVLAN, every pod is inside a dedicated network namespace and possesses its own iptables, as explained in the previous paragraph. Consequently, this approach works on these CNIs. More details about iptables, benefits and drawbacks of using it can be found in Section 3.3.1.

This approach has advantages as well as disadvantages. The most important advantage is that this solution can be used with all CNIs. This saves time and effort to manage a multi-tenant cluster that integrates a variety of CNIs. Besides, the design and configuration of this solution is simple and straightforward. It is not necessary to patch the Kubernetes source code to apply the solution. However, this solution may not provide 100% isolation. In some cases, the isolation can be bypassed. However, it is rare, and the impact is insignificant. More details can be found in Chapter 6.

3.3.1 IPTables

Iptables [48] is a legacy program that allows network administrators to set up the Linux kernel firewall by issuing and configuring chains of IP packet filter rules. When a packet comes in or out from the system, a kernel module named Netfilter [71] is responsible for matching the packet to lists of iptables rules. Each rule describes the desired packet and the action to do with this packet. The action can be DROP, ALLOW, FORWARD, or to pass the packet to another chain of rules. A table may contain several chains, including built-in chains as well as user-defined chains. By default, iptables consists of three essential tables: FILTER, NAT, and MANGLE. Inside the FILTER table, the three main chains of rules are INPUT, OUTPUT, and FORWARD. In this thesis, we use only the INPUT and OUTPUT chains in the FILTER table for implementing network isolation. All incoming packets have to be checked by rules in the INPUT chain, while packets coming out from the system are monitored and manipulated by the OUTPUT chain. More details in the use of these chains are provided in Chapter 4.

There are many benefits of using iptables. Firstly, iptables is easy to configure. In this thesis, a few simple rules are sufficient to provide the isolation. Secondly, iptables can also be used to build a fully customized and featured-rich local firewall that allows to flexibly enable or block any connections between pods from different Kubernetes namespaces. Finally, iptables rules are portable, and therefore the same set of rules can be used in all pods in one Kubernetes namespace. However, iptables also has its drawbacks. It can quickly become complicated when the number of rules increases. Besides, the network performance may be slowed as more rules are added to the tables [48]. In our case, as the number of rules added to each pod network namespace is small, the overhead generated by iptables should be low.

The network isolation solution is implemented by modifying the iptables inside each pod network namespace. From the host terminal, we can access the pod network namespaces to manually modify the iptables. However, this method cannot serve the whole cluster since it frequently spawns and deletes pods. To that end, the sidecar container and admission controller can be employed to create an automatic system to deploy the network isolation.

3.3.2 Sidecar Container

As mentioned in Section 2.1.1, Kubernetes pod is the basic unit of deployment in a cluster. Although it is typical that only one container runs in a pod, it is possible to create a pod with multiple containers inside. All containers in the pod share the same network namespace, which means they share a port space, routing tables, and especially, the same iptables. Subsequently, if one container changes its own iptables, the changed iptables will affect not only that container, but also the other containers in the same pod. This is the reason why we introduce the use of sidecar container. Sidecar container [35] is an additional container added to a pod, providing extra functionality that the main application container cannot provide. This extra container can modify iptables to a firewall blocking any traffic from other tenants. Figure 3.4 describes the containers inside a new pod. Inside this pod, the application container provides the actual application, while the sidecar container is responsible only for modifying the iptables of the pod.

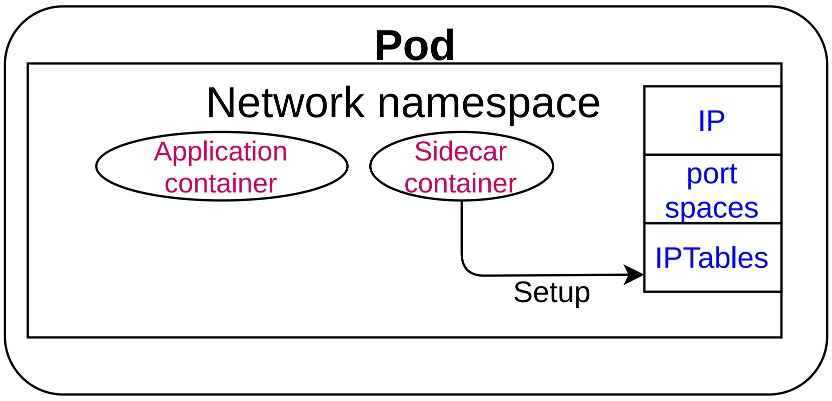

Figure 3.4: Inside a new pod that is injected a sidecar container

Obviously, the sidecar container should be added to the pod before it is entirely created and connects to the pod network. Otherwise, there will be a short period when the pod runs without the network isolation, thus leading to the race condition vulnerability [27]. Therefore, we need a mechanism to inject the sidecar container during the pod creation process automatically. Besides, this injection should be transparent to tenants. The sidecar should be added even though the tenant can see only one main container in their pod definition YAML file.

3.3.3 Admission Controller

Admission controller [30] is a mechanism in Kubernetes that allows intercepting requests sent by Kubernetes user to the API server. Admission controllers capture the API requests before they alter the cluster, but after they are authenticated and authorized. There is a list of controllers, each of which captures a certain type of requests [30]. In the list, there are two special controllers: *MutatingAdmissionWebhook* and *ValidatingAdmissionWebhook* that act as admission controller frameworks, allowing users to create customized admission controllers. These controllers intercept the requests coming from tenants, send them to a webhook server, a HTTP web server that is dedicated to receiving requests from admission controllers, handles the requests, and sends them back to the controllers. The webhook server then mutates and validates the requests before returning them to the API server. Since we need to inject a sidecar container to a pod during the pod creation, the *MutatingAdmissionWebhook* can be exploited to capture and modify the pod deployment requests. The modification is simply adding an extra container, the sidecar, with a specific pre-defined image. Then, the requests coming to the API server will contain the description of two containers: the main one and the sidecar one. The cluster, after that, will deploy a pod corresponding to that request. Figure 3.5 describes the flow of the pod creation request.

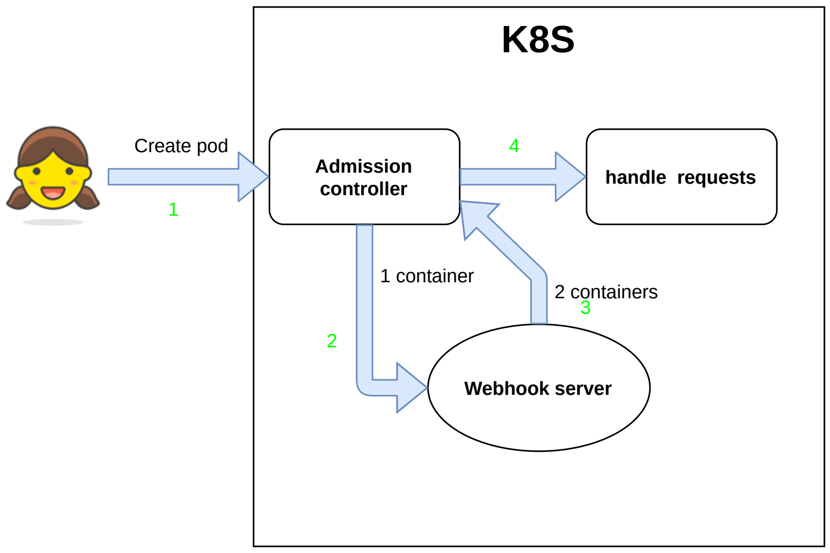

Figure 3.5: The admission controller intercepts and modifies the pod creation request

Chapter 4

Architecture and Implementation

In this section, we design a proof of concept (PoC) of the solution proposed in Chapter 3. The goal of this PoC is to show that the solution is able to isolate the network between pods from different Kubernetes namespaces. We use this PoC to conduct several experiments and to evaluate the security and performance of the proposed solution in Chapters 6 and 7.

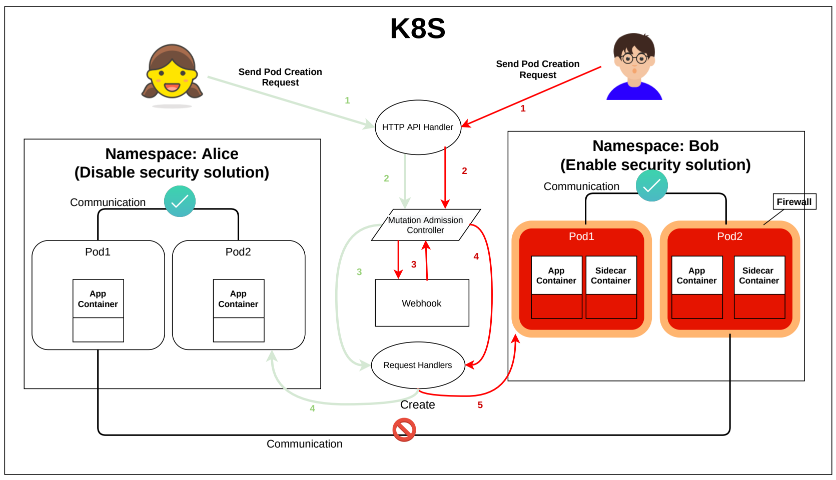

Figure 4.1: Overview of the solution

The PoC is an implementation of our network isolation solution. Figure 4.1 describes our solution at a high level. Alice and Bob are two tenants in this system. Each tenant is assigned to a Kubernetes namespace. In this figure, we disable the solution on Alice's namespace to highlight the difference between applying and not applying our solution. When Alice sends a pod creation request, it will come to the HTTP API handler before being forwarded to the admission controller. Since we disable the solution on her site, the admission controller ignores her request and passes it to the next handlers. Other components of the cluster handle her request and create pods in Alice's namespace. These new pods comprise only Alice's desired containers, and the network isolation solution does not protect her pods.

While Bob's pod creation requests follow mostly the same route as Alice's, the admission controller intercepts it and sends it to our webhook server. The webhook modifies this request, injecting an extra container to new pods. Therefore, Bob's new pods consist of at least two containers: his application containers, and the sidecar container. This sidecar executes a pre-defined script that sets up the internal iptables inside itself. As mentioned in Section 3.3.2, all containers in a pod share the same iptables. Therefore, the modified firewall also works on the application containers and the whole pod. In other words, Bob's pods now are protected by their internal firewall and cannot be accessed by Alice's pods.

4.1 IPTables in Pod Network Namespace

The iptables rules added to the network namespace of each pod follow the format in Figure 4.2:

```
-A INPUT -s <Alice's pod IP range> -d <Alice's pod IP range>
   -j ACCEPT
-A INPUT -s <All tenants pod IP range> -d <All tenants pod IP
    range> -j DROP
-A OUTPUT -s <Alice's pod IP range> -d <Alice's pod IP range>
    -j ACCEPT
-A OUTPUT -s <All tenants pod IP range> -d <All tenants pod
   IP range> -j DROP
```

Figure 4.2: Format of iptables in the network namespace of each pod in the Alice's namespace. Alice is a tenant

The first two rules are added to the INPUT chain, while the third and the forth rules are added to the OUTPUT chain. The first rule allows all connections between pods in the current tenant. The second rule is responsible for dropping any traffic that comes from or reach a destination in other tenants' pods. The third and fourth rules are similar to the first and second, respectively.

Figure 4.3 shows an example of rules added to a pod in the PoC

```
-A INPUT -s 172.16.2.0/24 -d 172.16.2.0/24 -j ACCEPT
-A INPUT -s 172.16.0.0/14 -d 172.16.0.0/14 -j DROP
-A OUTPUT -s 172.16.2.0/24 -d 172.16.2.0/24 -j ACCEPT
-A OUTPUT -s 172.16.0.0/14 -d 172.16.0.0/14 -j DROP
```

Figure 4.3: IPTables rules in a pod

In this example, **172.16.2.0/24** is the IP range allocated for the current tenant. **172.16.0.0/14** is the IP range used for entire collection of pods controlled by tenants in this cluster. The IP address allocation is explained in Section 5.3.

4.2 Sidecar Container Content

This sidecar container should be deployed at the same time as the main container, and it needs to be ready before the pod can be in use. After the sidecar container has been successfully created, it executes a setup script, which can be a compiled binary or a script, to add the required iptables rules mentioned in the previous subsection. More details about input and output of this script:

1. Input: Information about the Kubernetes namespace in which the current pod is inside, including the name of the namespace, or the IP range associated to the namespace (the IP range that is allocated to the tenant who own this Kubernetes namespace).

2. Output: Deployment of iptables rules that provides the network isolation. These rules follow the format in the previous section. The script replaces the place holder <**Alice's pod IP range**> by the IP range extracted from the input.

Next, the script can proceed to the sleep mode (continue to run without actually doing anything to keep the container alive) or work as a watchdog. More details can be found in the discussion chapter.

In the PoC, the *Dockerfile* in Figure 4.4 is used to build the sidecar container image. Two files are added to this image. *multi-tenancy-sidecar.py* is a Python script that acts as the script for adding IPtables rules. It receives the name of the namespace as an input and then uses it to find the corresponding IP range. In order to do that, it searches on a table mapping between Kubernetes namespaces and their IP range, and this table is stored in a JavaScript object notation (JSON) file. This JSON file is *iplist.json*, which is the second file added to the image. The image is finally pushed to the container registry [67]. After we push the sidecar image from the development workstation to the registry, the image is available online, and the cluster can pull it to deploy sidecar container.

```
FROM ubuntu:18.04
MAINTAINER Nam Xuan Nguyen
RUN apt update -y
RUN apt install -y python3-pip iptables
RUN pip3 install python-iptables
ADD multi-tenancy-sidecar.py .
ADD iplist.json .
RUN chmod u+x multi-tenancy-sidecar.py
ENTRYPOINT ["./multi-tenancy-sidecar.py"]
```

Figure 4.4: The Dockerfile used to build an image for the sidecar container

4.3 Admission Controller

The use of the admission controller is to inject a sidecar container configuration to pod creation requests automatically. Firstly, the *MutatingAdmissionWebhook* controller needs to be configured to intercept pod creation requests and send this traffic to a webhook server. This server can be deployed as a Kubernetes pod or as an external server. It is responsible for listening to the incoming request; extracting the namespace information from the request; adding this information as a parameter to the sidecar container configuration of the requests, and finally sending the modified request back to the admission controller.

An admission controller is available through an instance of the target controller. In the PoC, an instance of *MutatingAdmissionWebhook* is created by a definition YAML file (Figure 4.5) This configuration file indicates the information of the webhook server. The name of this webhook is *sidecar-injector.multi-tenancy.local*, and it sits behind a service named *multi-tenancy-sidecar-injector-webhook-svc*. When the admission controller captures a request, it sends the request to the service, and the service then forwards it to the webhook server. Another important point of the configuration file is the **namespaceSelector** parameter. This parameter specifies the namespace that uses this admission controller. In the PoC, the admission controller only works with the namespaces having the label: **sidecar-injector: enabled**. More precisely, it only intercepts pod creation requests

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: multi-tenancy-sidecar-injector-webhook-cfg
  labels:
    app: multi-tenancy-sidecar-injector
webhooks:
  - name: sidecar-injector.multi-tenancy.local
    clientConfig:
      service:
        name: multi-tenancy-sidecar-injector-webhook-svc
        namespace: default
        path: "/mutate"
      caBundle: ${CA_BUNDLE}
    rules:
      - operations: [ "CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
    namespaceSelector:
      matchLabels:
        sidecar-injector: enabled
```

Figure 4.5: Mutating webhook configuration

generated from these Kubernetes namespaces. This mechanism is useful for the evaluation part of this thesis, and it is also an efficient method to apply different security profiles to different Kubernetes namespaces, which can be utilized in the future work.

The webhook server in the PoC is a derivative work of an open-source project [50], which was initially designed to add a sidecar container to the pod creation request. In this thesis, the webhook is modified to pass the Kubernetes namespace where the request comes from to the sidecar container configuration as a parameter. This parameter is used as the input of the sidecar container.

In order to employ the image that is specially prepared for the sidecar (see Section 3.3.2), a configmap (a Kubernetes object containing configuration information) is deployed to provide the required specification (see Figure 4.6). This configmap specifies the name of the sidecar image, which was prepared and pushed to the container repository earlier (see Section 4.2), and the capabilities [45] granted to the container. In order to access iptables, the sidecar container needs to be granted the *NET_ADMIN* and *NET_RAW* capabilities. More details about these capabilities can be found in Section 6.8.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: multi-tenancy-sidecar-injector-webhook-configmap
data:
  sidecarconfig.yaml: |
    policy: enabled
    containers:
      - name: multi-tenancy-sidecar
        image: namnx228/k8s-multitenancy-sidecar-container-
   amd64-${i}:latest
        imagePullPolicy: IfNotPresent
        securityContext:
          capabilities:
            add:
              - NET_ADMIN
              - NET_RAW
        volumeMounts:
            - name: multi-tenancy-output
              mountPath: /out
    volumes:
      - name: multi-tenancy-output
        hostPath:
            path: /out
            type: DirectoryOrCreate
```

Figure 4.6: The configmap YAML file

To sum up, when a tenant sends a pod deployment request, the admission controller intercepts it and sends it to a webhook server. The webhook server adds the sidecar container configuration to the request before sending it back to the admission controller to continue the pod creation process. A new pod is subsequently created with two containers inside: the application container and the sidecar container. The sidecar container executes a script that adds the iptables rules to the network namespace of the pod. These rules are applied to all containers in the pod and therefore ensure the network isolation.

Chapter 5

Test Environment

This section introduces the methods and results of experiments conducted to evaluate the security and performance impact of the solution described in Chapter 4.

5.1 Development Workstation

We implemented and evaluated our solution, including developing the PoC, hosting the virtual cluster, and running tests, on a Linux computer. The system information of this computer can be found in Table 5.1.

The Kubernetes cluster used in the PoC and the evaluation runs on top of virtual machines provided by Vagrant and VirtualBox. VirtualBox [55] is a free and open-source hypervisor that provides virtualization for x86 and x86_x64 hardware. It is a powerful tool to create virtual machines on top of physical machines, such as laptops, desktops, and servers. Vagrant [46] is also an open-source tool for automatically building and managing virtual machines in a simple workflow. Vagrant does not provide virtual machines, but only it interacts with hypervisors, e.g., VirtualBox. The hypervisors perform

Model	Dell-Latitude-7490
CPU	Intel®Core™i7-8650U CPU @ 1.90GHz * 8
Memory	32GB
Hard disk	500 GB
Operating system	Ubuntu 18.04 bionic
Kernel	x86_64 Linux 5.3.0-45-generic

Table 5.1: System information of the computer used for running experiments

the actual actions, such as creating new machines, starting, shutting down, restarting, and networking. With Vagrant, we can add desired configurations into a file, named *Vagrantfile*, and then run only one command *vagrant up*. After that, Vagrant automatically creates the virtual machines and configures them according to the *Vagrantfile*. The configurations in this file include the machine name, CPU settings, amount of random-access memory (RAM), type of network, IP address, and provisioning commands [63] that run only once when the virtual machines start. Provisioning commands set up necessary software and dependencies for deploying a Kubernetes cluster and multi-tenancy environment. Vagrant supports several provisioners, such as Shell, Salt, Chef, Puppet, and Ansible. Ansible [4] is an open-source tool for provisioning and application deployment automation. It offers a simple declarative language to describe the configuration of a machine. In this work, Ansible is employed as the provisioner to download and install Docker, Kubernetes, and git. It is also used to pull our source code from Github, then run the code to set up a multi-tenancy environment and the PoC introduced in Chapter 4. Figure 5.1 shows an overview of the cluster setup process explained above.

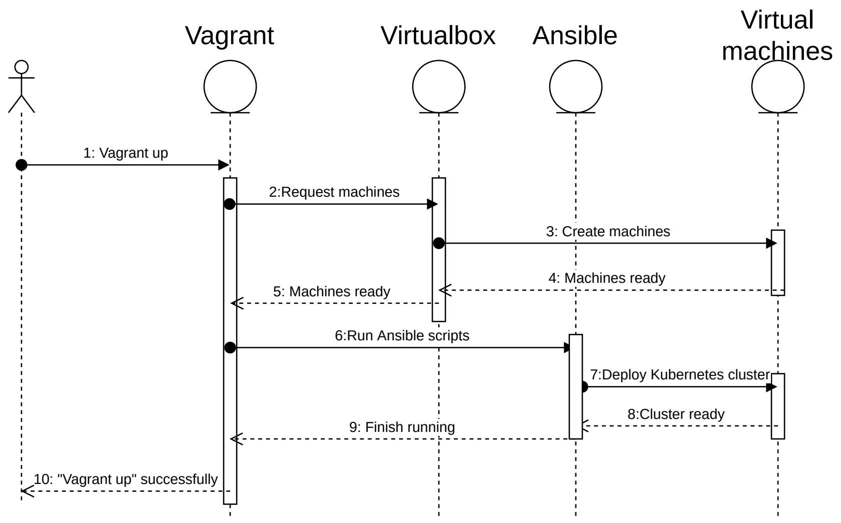

Figure 5.1: An sequence diagram of the cluster setup

The cluster includes three virtual machines **k8s-master**, **node-1**, and **node-2**. **K8s-master** is the master node containing the Kubernetes control plane (API server, etcd, scheduler, controller manager) while the other two nodes are the worker nodes handling the actual workload. In addition, **tenants-machine**, which is not a part of the cluster, is deployed and pre-

Machine	k8s-master	node-i	tenants-machine
CPU (Number of cores)	2	4	2
Memory	3GB	8GB	8GB
OS	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04
IP address	192.168.50.10	192.168.50.(i+10)	192.168.50.13

Table 5.2: System information of virtual machines (*i=1,2*)

pared as a place from which users can access and interact with the cluster. The aim of using this machine is to prevent users from directly accessing **k8s-master** because user activities in this node can cause security and performance issues to the cluster. The method that users use the **tenants-machine** to interact with the cluster is described in Section 5.2. Table 5.2 describes the system information of these machines. Figure 5.2 explains the topology of the PoC system. For the development process, code from the development machine (the host machine) is pushed into a Github remote repository, after which the *tenants-machine* pulls the code down to its local repository before executing it.

5.2 Multi-Tenancy Setup

To test our network isolation solution, a concrete soft-multi-tenant cluster (see Section 2.3.1) is required to provide tenant isolation, at least from a management point of view. A simple soft multi-tenancy solution allocates each tenant a Kubernetes namespace. Within each Kubernetes namespace, a role object (see Section 2.1.2) is created to define a set of permissions. A role is then assigned to a normal user by a role binding object (see Section 2.1.2). That means the normal user is granted the entire list of permissions included in that role. The normal user (see Section 2.1.2) is a representation of a human tenant. To send a command to the cluster, the tenant needs to access *tenants-machine*, then use her certificate and private key to proceed with the authentication process. If the API server can authenticate this tenant as a legitimate normal user, it authorizes her based on her role and finally executes her command. The tenant issues commands via kubectl (see Section 2.1.3), a tool that allows interacting with the Kubernetes cluster using the Kubernetes native HTTP API.

Figure 5.3 shows the relation between the components of the soft multi-tenancy setup mentioned above. This setup aims to ensure that a tenant is unable to access the resources of another tenant. For example, tenant Bob cannot read the private key and certificate of tenant Alice since these files

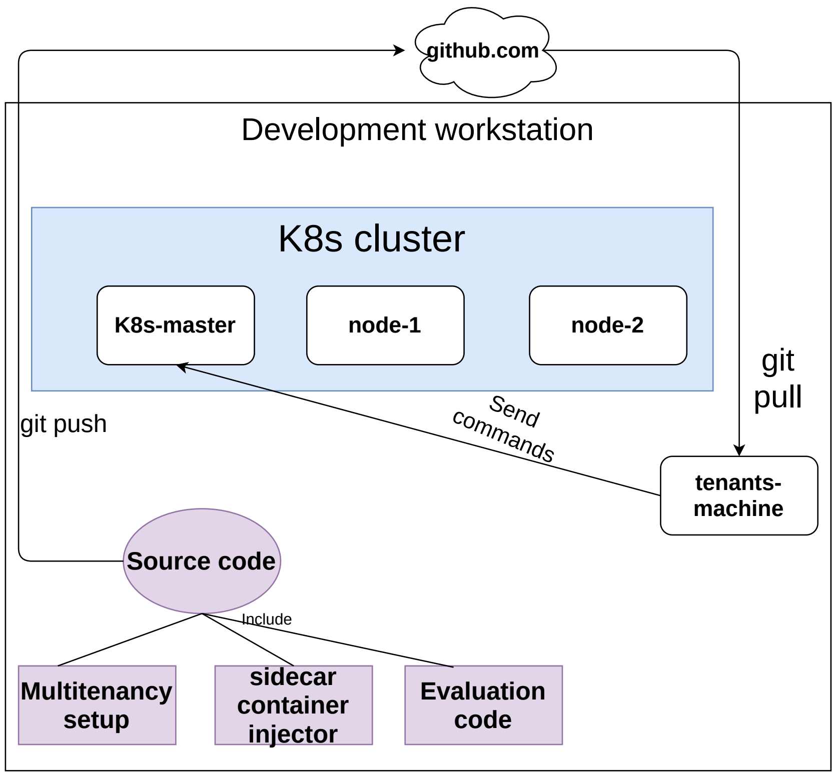

Figure 5.2: The topology of the node setup in the PoC

are protected by Linux access control, and only Alice can read them. Bob is also unable to bind to the role object of Alice because the role binding object only defines the relation between the user Alice and the role Alice. For that reason, Bob cannot do any actions in the Kubernetes namespace Alice, such as to create pods or delete pods. To sum up, with this setup, a tenant cannot access and perform any actions on another tenant's Kubernetes namespace.

This multi-tenant environment is used only for testing purpose, and we highly recommend to avoid this design in production. In reality, each tenant should store their private key in their own computer, and from here, they call HTTP API to interact with the cluster remotely. We can also deploy one virtual machine for each tenant, but this creates extra workloads to the physical Linux computer and affects its performance. For that reason, we allow tenants to access a shared virtual machine **tenants-machine** using SSH, and from here, they can call the Kubernetes API to work with the cluster. Storing all tenant's private keys used to log in to the cluster in one virtual machine is not a secure approach, that is why we recommend to

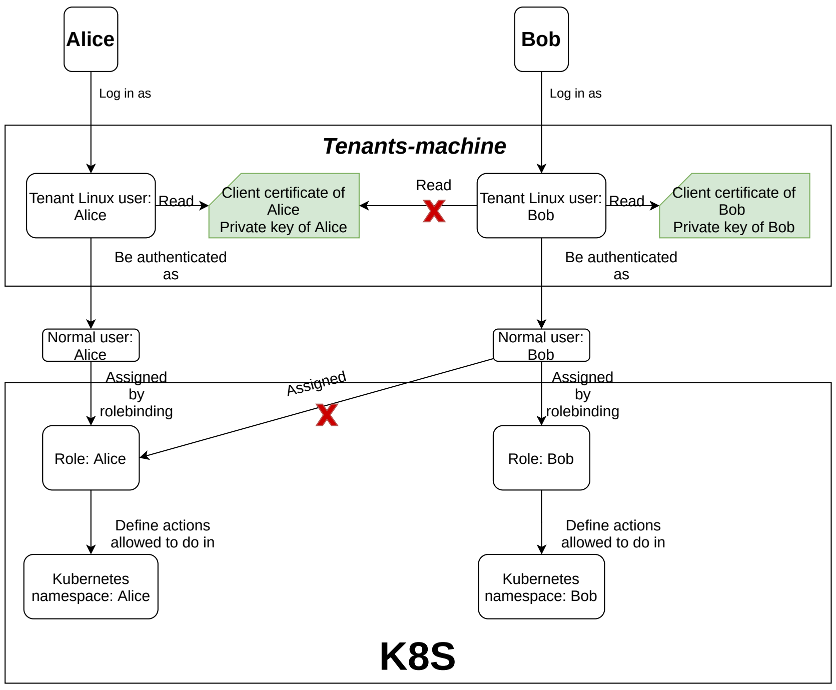

Figure 5.3: An overview of the soft multi-tenancy setup

avoid it. Besides, an example of the role object in this setup can be found in Figure 5.4.

According to this example, the role contains permissions to create, delete, get pods, service, role, role binding, and other resources. Since Alice is assigned to this role (by the role binding object in Figure 5.5), she is granted all permissions in it. On the contrary, Bob is not assigned to this role, and therefore he has no permissions in Alice's namespace, because the default action in Kubernetes RBAC is to deny everything.

5.3 Software Version and IP Address

In the PoC, several network endpoints need an IP address to join the network, including nodes, pods, services. Table 5.3 introduces the IP ranges used in the PoC.

The table indicates that the IP address range allocated for all pods of the cluster is *172.16.0.0/13*. This range is divided into smaller ranges for pods within the default Kubernetes namespace and the tenant ones. Table 5.4 shows more details about these ranges.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: Alice
  name: Alice
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods", "networkpolicy", "services", "role", "
    rolebinding", "deployments", "node", "pods/attach", "pods/
    exec", "pods/log"]
  verbs: ["view", "create", "get", "watch", "list", "edit", "
    delete", "scale"]
- apiGroups: [ "apps" ] # "" indicates the core API group
  resources: ["pods", "networkpolicy", "service", "role", "
    rolebinding", "deployments", "deployments/scale","node"]
  verbs: ["view", "create", "get", "watch", "list", "edit", "
    delete", "scale", "patch"]
- apiGroups: [ "networking.k8s.io" ] # "" indicates the core
    API group
  resources: [ "networkpolicies"]
  verbs: ["view", "create", "get", "watch", "list", "edit", "
    delete"]
```

Figure 5.4: An example of the role object

Endpoints	IP range
Nodes	192.168.50.10/25
Services	10.92.0.0/12
Pods	172.16.0.0/13

Table 5.3: IP ranges of nodes, services and pods

We performed all IP range allocations in the table. The IP range allocated for all tenants and the default Kubernetes namespace were set in the cluster network configuration file before deploying the cluster. We allocated the other IP ranges after the Kubernetes network had started. To dynamically assign these ranges, calicoctl [8] was used as a means of communication with the Calico network engine running in the cluster. The allocations were written in YAML files and then sent to the engine. One example of the YAML file can be found in Figure 5.6.

Version number of tools and programs, which are employed in the PoC, are summarised in Table 5.5. The source code of all programs, test environment and experiments that we implemented in this thesis is at `https://github.com/namnx228/MasterThesis`

Namespace	IP range
Default	172.23.0.0/18
Tenant *Alice*	172.16.0.0/24
Tenant *Bob*	172.16.1.0/24
...	...
For all tenants	172.16.0.0/14

Table 5.4: IP ranges assigned to pods in default and tenant Kubernetes namespaces

Tools	Version
Vagrant	2.2.7
VirtualBox	6.1.4
Tenant *test 2*	172.16.1.0/28
Ubuntu	18.04
Ansible	2.9.7
Docker	19.03.8
Ubuntu (used in Docker images)	18.04
Kubernetes	1.18.3
Nginx (used in pods)	1.19.0
Python 2	2.7.17
Python 3	3.6.9
iperf	2.0.10
Calico	3.13.1
IPTables	v1.6.1
Git	2.7.1
hping3	3.0.0-alpha-2

Table 5.5: Summary of the tools used in this thesis

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: Binding Alice to Alice role
  namespace: Alice
subjects:
- kind: User
  name: Alice # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role
  name: Alice # this must match the name of the Role you wish
    to bind to
  apiGroup: rbac.authorization.k8s.io
```

Figure 5.5: An example of the role binding object

```yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: Alice-pool
spec:
  cidr: 172.16.1.0/24
  blockSize: 29
  ipipMode: Always
  natOutgoing: true
```

Figure 5.6: IP Allocation for tenant Alice

Chapter 6

Security Evaluation

This chapter demonstrates several experiments running on the multi-tenancy platform described in Chapter 5 with the network isolation solution explained in Chapter 4. This evaluation examines the security of the network isolation method, analyzing its potential and weaknesses that can be improved.

In short, the network isolation method injects an extra container to tenant pods. This container is called *sidecar container*, and its mission is to modify the iptables of the pod. The added iptables rules act as a firewall that blocks traffic coming from and to pods in different Kubernetes namespaces. In this thesis, we assign each tenant a Kubernetes namespace. Therefore, the added rules can provide the network isolation in the multi-tenancy environment. Thus, this section examines the isolation that it can offer, leading to two main questions:

1. In what cases can the isolation be broken?
2. How critically can the broken isolation impact the tenant?

These two questions can be answered by conducting a series of experiments. Each experiment runs a test case, and this set of test cases is expected to cover typical connections between pods. In the experiments, we set up a multi-tenant cluster serving two tenants, Alice and Bob (see Section 5.2). The network isolation solution is enabled for both tenants. Since the subject in these experiments is the network isolation enforcement, it is necessary to generate intra-tenant and inter-tenant network traffic. In our experiments, the traffic is generated by wget [33], a HTTP client tool on one pod, and sent to a Nginx [60] web server running on another pod. In the sender pod, if wget receives the reply from the server, it saves the HTTP response into a file or returns it to the standard output (stdout) [2]. Otherwise, wget returns an error message to the standard error (stderr) [1]. This behavior can be exploited to determine whether the connection between the two pods is successfully established or not.

6.1 Experiment 1

In this experiment, a pod running wget in the Kubernetes namespace Alice attempted to connect to a pod running Nginx web server in the namespace Bob. The goal of this experiment is to show that the network isolation solution can successfully block the communication between two pods in the namespaces of two different tenants. Figure 6.1 shows more details of the experiment.

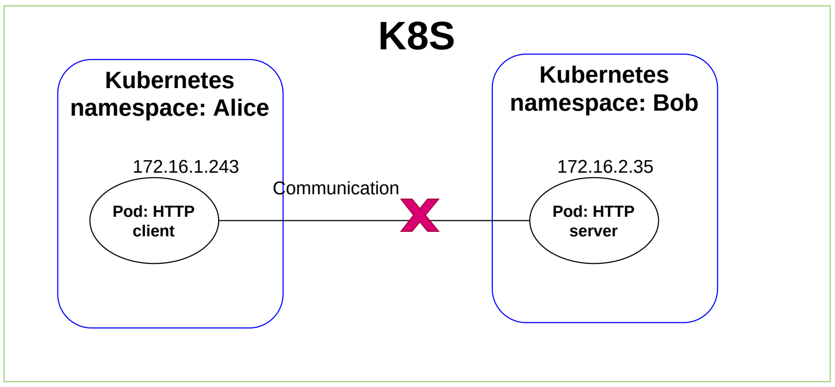

Figure 6.1: An overview of experiment 1 in the security evaluation

The result is the same as our expectation. The wget pod in tenant Alice could not establish a connection to the HTTP server pod in tenant Bob (see Figure 6.2). The reason is that the iptables in the source pod had been configured by the sidecar container to block connections to the tenant Bob.

6.2 Experiment 2

In this experiment, two pods in tenant Alice attempted to connect to each other. The experiment is expected to provide evidence that this connection is allowed when we deploy the network isolation solution. Figure 6.3 depicts this experiment. Figure 6.4 shows that the connection between the two pods in the same Kubernetes namespace is allowed. This connection is enabled by two rules (see Figure 6.5), which are injected to iptables of the pods in tenant Alice.

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$
   kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
   the containers in this pod.
/ # wget 172.16.2.35
Connecting to 172.16.2.35 (172.16.2.35:80)
wget: can't connect to remote host (172.16.2.35): Operation
   timed out
```

Figure 6.2: The connection in experiment 1 is blocked

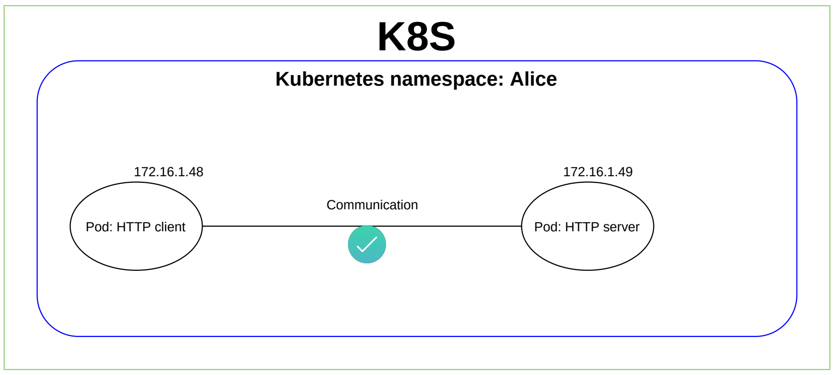

Figure 6.3: An overview of experiment 2 in the security evaluation

6.3 Experiment 3

This experiment again verifies the ability to block traffic traveling between different tenants of the network isolation solution. However, the sender in this experiment communicates with a service deployed to forward packets to two HTTP web servers. This experiment is depicted in Figure 6.6.

The basis of this experiment is that pods are ephemeral and can die anytime. If a consumer contacts an application pod by its IP address, but the pod dies and is replaced by another pod, the consumer will be unable to access the application. More details can be found in Section 2.1.1. As a result, the IP address of a service referring to the application pods is more reliable and normally offered to the consumer. For example, in this experiment, the *HTTP client* pod is a consumer, sending HTTP requests to the *HTTP load balancer* service. The service then forwards the requests to one of the two HTTP server endpoints. If the two pods were terminated, the cluster would create new server pods to replace them. While the endpoint

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$
   kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
   the containers in this pod.
/ # wget 172.16.1.49
Connecting to 172.16.1.49 (172.16.1.49:80)
index.html           100% |***************|   612   0:00:00
   ETA
/ #
```

Figure 6.4: The connection in experiment 2 is allowed

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  172.16.1.0/24        172.16.1.0/24
DROP       all  --  172.16.0.0/14        172.16.0.0/14

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  172.16.1.0/24        172.16.1.0/24
DROP       all  --  172.16.0.0/14        172.16.0.0/14
```

Figure 6.5: The iptables of pods in tenant Alice

IP addresses could change, the consumer would not be influenced since the service is responsible for forwarding the traffic to the newborn servers.

Figure 6.6 shows that the service belongs to tenant Bob, the receiver, not Alice, the sender. The reason is that Bob, who deploys the HTTP server pods, needs to introduce the service as a load balancer to his two server pods. Alice and other clients can then connect to the service and enjoy the HTTP application provided by Bob's pods without knowing the real IP addresses of these pods. We discuss more details about Kubernetes service in the next paragraph.

The result of this experiment is indicated in Figure 6.7. Again, the solution blocked the connection from a pod in the tenant Alice to a service referring to pods in tenant Bob. Understanding the implementation of Kubernetes services is required to understand the necessity of this experiment.

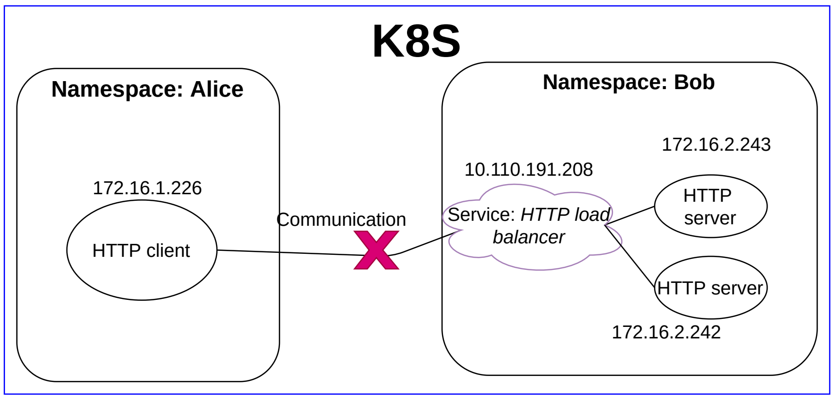

Figure 6.6: An overview of experiment 3 in the security evaluation

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test/$
   kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
   the containers in this pod.
/ # wget 10.110.191.208
Connecting to 10.110.191.208 (10.110.191.208:80)
wget: can't connect to remote host (10.110.191.208):
   Operation timed out
```

Figure 6.7: The connection in experiment 3 is blocked

By default, services are fully implemented by the host iptables (the main iptables in the host) in all nodes of the cluster. Although Bob, the server side, is the one who defines a service object, it is implemented by the iptables in every node of the cluster because Alice's pods, the client, can be located in any of the nodes. They can be even in the same node as the Bob's server pods. Figure 6.8 presents a set of rules used to forward traffic when it comes to the service *HTTP load balancer*. These rules are added to the iptables of every node in the cluster. However, only the netfilter module (the Linux module handling iptables rules) in the node, where the client pods run, handles packets sent from these pods. After being sent to the service IP address, the traffic is handled by the rule 1. From this rule, the traffic has a 50% possibility to flow to rule 2 or rule 3. If the traffic is processed by rule 2, it is then passed to rule 4 before being forwarded to the endpoint at *172.16.2.243*. Otherwise, it will reach the IP address *172.16.2.242*. Figure 6.9 represents the flow of the traffic when it comes to the service *HTTP service*.

```
1. -A KUBE-SERVICES -d 10.110.191.208/32 -p tcp -m comment --
   comment "bob/nginx-deployment: cluster IP" -m tcp --dport
   80 -j KUBE-SVC-JNZXJXY5BLALO2QH
2. -A KUBE-SVC-JNZXJXY5BLALO2QH -m comment --comment "bob/
   nginx-deployment:" -m statistic --mode random --
   probability 0.50000000000 -j KUBE-SEP-GMBG2FBMWB36ZA5Q
3. -A KUBE-SVC-JNZXJXY5BLALO2QH -m comment --comment "bob/
   nginx-deployment:" -j KUBE-SEP-OX2VTKLOZBMIEOIJ
4. -A KUBE-SEP-GMBG2FBMWB36ZA5Q -p tcp -m comment --comment "
   bob/nginx-deployment:" -m tcp -j DNAT --to-destination
   172.16.2.242:80
5. -A KUBE-SEP-OX2VTKLOZBMIEOIJ -p tcp -m comment --comment "
   bob/nginx-deployment:" -m tcp -j DNAT --to-destination
   172.16.2.243:80
```

Figure 6.8: IPTables rules that implement service *HTTP load balancer*

As explained above, packets traveling to the service are processed by the netfilter module on the client node, the destination of the packets is the HTTP server pods, and the source IP addresses of the packets remain unchanged. As a result, the iptables in the destination pods detect that the packets come from a foreign pods, thus blocking them.

6.4 Experiment 4

This experiment explores the case when a pod communicates with a service within the same tenant. The motivation of this experiment is the same as in experiment 3. Since the iptables in the destination pods detects that the source IP address of the incoming packet is in the IP range of tenant Alice, it allows the packet to come through, and the connection is established. An overview of this experiment can be found in Figure 6.10, and the result can be observed in Figure 6.11.

6.5 Experiment 5

The goal of the network isolation method is to prevent communications between tenants. However, pods in a tenant Kubernetes namespace should be able to connect to external services on the internet and to Kubernetes default services. The setup of this experiment is depicted in the Figure 6.12. The pod in the namespace Alice can connect to `google.com` and interact with the Kubernetes domain name system (DNS) service. The reason is that the

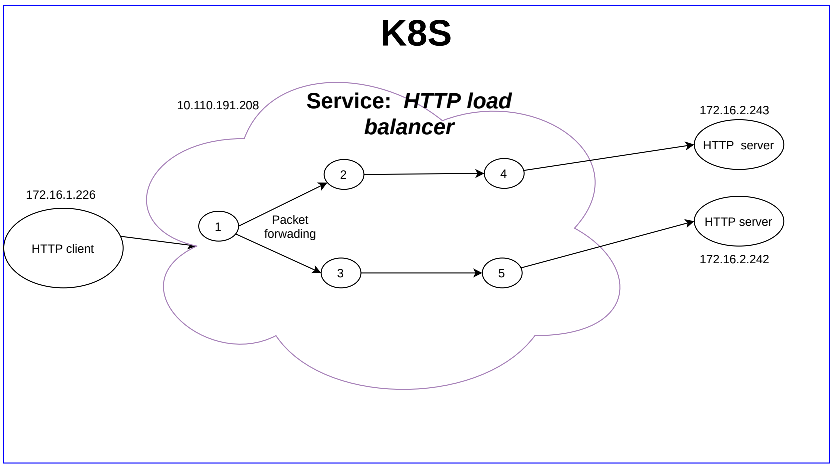

Figure 6.9: Implementation of service *HTTP load balancer*

“DROP” rules added to pod iptables only match tenant pod IP addresses. These iptables rules do not block traffic traversing to the Internet or to the Kubernetes default namespace. The result of this experiment is illustrated in Figure 6.13 and Figure 6.14.

From a security point of view, enabling the connection between tenant pods and external services can lead to security issues. If a tenant decides to expose her pod to the outsider on the internet (this pod can be a web service that is publicly available to clients on the internet), the network isolation solution is incapable of stopping other tenants from connecting to this exposed pod through the internet. A misbehaving tenant can exploit these issues to compromise the isolation.

Figure 6.15 indicates one scenario this is possible. The red arrows represent the route that tenant Alice can exploit to connect to HTTP servers in the Kubernetes namespace Bob. When packets from the namespace Alice come to the kernel, their source IP addresses are masqueraded as the host IP address (SNAT) before leaving to the internet. They then reach the public IP address which the cloud provider allocates to Bob’s HTTP services. Subsequently, their destination IP addresses are masqueraded as the IP address of service *External* (DNAT), and they are sent to this service. Finally, the service passes the traffic to the backend servers.

This security issue is not critical. In most cases, the issue is policy misconfiguration. It simply is not feasible to block access by other tenants while allowing access from the open internet. Before an application is ready to be exposed to the internet, developers need to install security measures to the

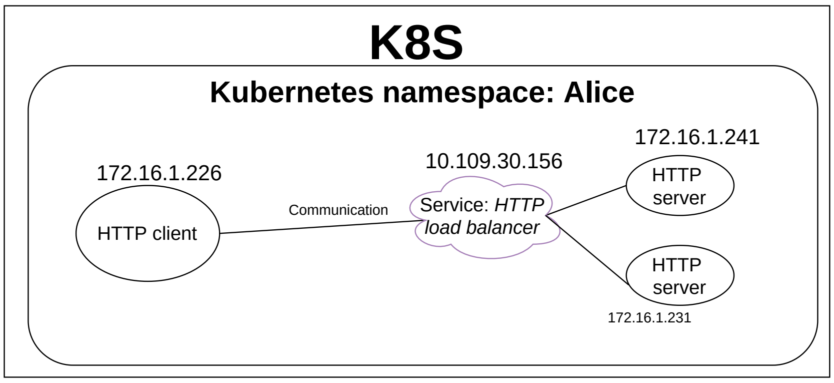

Figure 6.10: An overview of experiment 4 in the security evaluation

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test/$
    kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
    the containers in this pod.
/ # wget 10.109.30.156
Connecting to 10.109.30.156 (10.109.30.156:80)
index.html           100% |*****************|   612
    0:00:00 ETA
```

Figure 6.11: The connection in experiment 4 is allowed

application to prevent unauthorized access. For example, a web application needs to equip authentication mechanisms and only provide services to legitimate users. Consequently, it is possible to rely on this application protection layer to block any unauthorized access from other tenants. Therefore, we rate the critical level of this issue as low.

The solution for this issue is to keep in mind that in a data center, tenants can connect to the other's public services via the internet, regardless of the approach we design and implement the isolation policy within the data center.

6.6 Experiment 6

In this experiment, we assume that the tenant Alice is an attacker who aims to bypass the isolation and connect to other tenants. The attacker first drops the iptables rules added to her pods as these rules prevent her from contacting the other tenants. Next, she sends requests to the HTTP servers in tenant

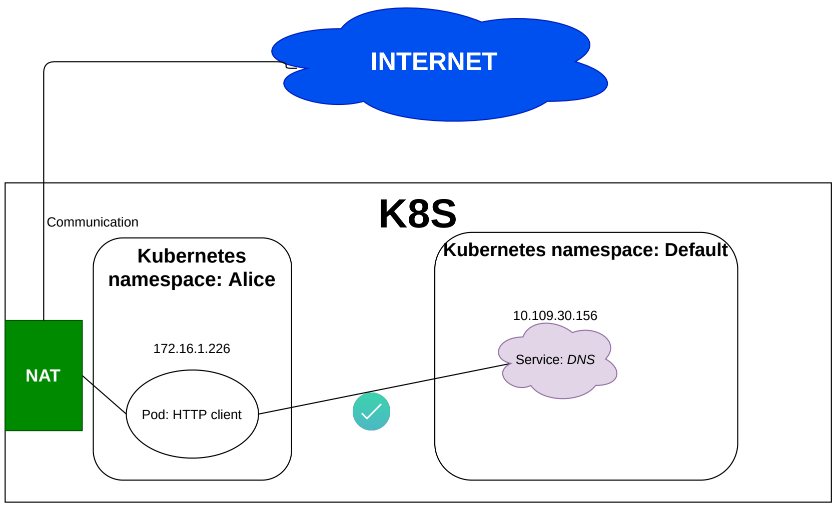

Figure 6.12: An overview of experiment 5 in the security evaluation

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$ kubectl get svc -o wide
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE    SELECTOR
nginx-deployment    ClusterIP   10.109.30.156   <none>        80/TCP    162m   app=nginx
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$ kubectl exec dnsutils -it --
    /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of the containers in this pod.
/ # nslookup nginx-deployment
Server:     10.96.0.10
Address:    10.96.0.10#53

Name:  nginx-deployment.alice.svc.cluster.local
Address:  10.109.30.156

/ #
```

Figure 6.13: The tenant pod can connect to the default DNS service of the cluster

Bob. Figure 6.16 describes an overview of this experiment, and the result is in Figure 6.17.

Even though the traffic can survive through the sender's IPtables since the protection rules have been removed, it is blocked by the receiver's iptables.

6.7 Experiment 7

As above, this experiment investigates the case that tenant Alice is an attacker. However, in this experiment, the tenant Bob unintentionally deactivates the firewall setting provided by our solution. An overview of this experiment can be seen in Figure 6.18, and the result is shown in Figure 6.19.

```
/ # wget google.com
Connecting to google.com (172.217.21.174:80)
Connecting to www.google.com (172.217.20.36:80)
index.html           100% |*******************| 12910
    0:00:00 ETA
```

Figure 6.14: The tenant pod can connect to a website on the internet

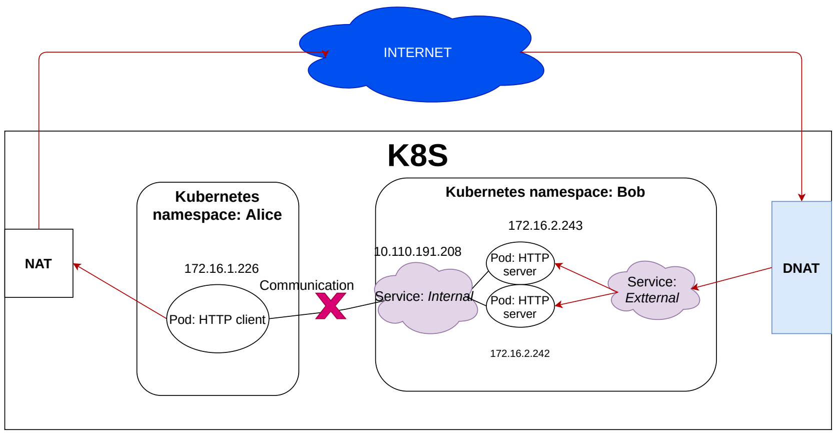

Figure 6.15: An scenario describes the attack through the Internet

Tenant Alice successfully connected to the Kubernetes namespace Bob's HTTP server because the isolation had been disabled. The iptables rules used for isolation in the sender and receiver pods had been removed. This is one weakness of the inner-firewall, the type of firewall we employ for isolation [66]. A misbehaving insider can turn off the security protection and allow an outsider to come in. However, this issue is minor in our evaluation. The main reason is that Kubernetes users regularly configure the network and firewall at the namespace scale using services or network policies, because this approach simultaneously affects all the running pods, as well as future pods. In other words, tenants rarely access the pod one by one and change the network configuration, including the iptables.

Consequently, the probability that a tenant accesses a pod and carelessly overrides or deletes the protection rules is low. Even if a malicious pod has deactivated the security measurement, the other pods are not threatened since their iptables remain unchanged. Therefore, bypassing the isolation of one pod does not lead to compromise the isolation of the tenant's other pods. Thus, the impact is modest. However, this issue remains unsolved. For these reasons, the risk level of this issue is medium.

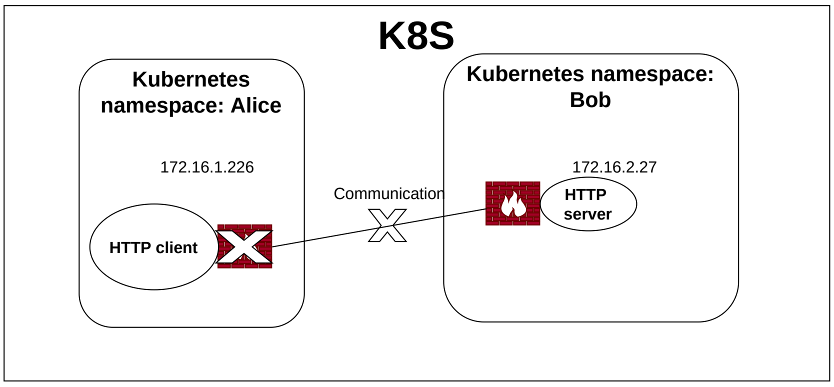

Figure 6.16: An overview of experiment 6 in the security evaluation

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$
    kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
    the containers in this pod.
/ # wget 172.16.2.27
Connecting to 172.16.2.27 (172.16.2.27:80)
wget: can't connect to remote host (172.16.2.27): Operation
    timed out
```

Figure 6.17: The connection in experiment 6 is blocked

6.8 Experiment 8

In this experiment, we investigate the case where the attacker is able to run privileged pods. By default, a tenant can deploy a privileged pod [65]. Figure 6.20 shows an example of this powerful pod.

The root user in this pod has the same privilege as the root of the host. Using this pod, tenant Alice (the attacker) can drop the protection rules of the HTTP pod in tenant Bob (the victim) before connecting to it (see Figure 6.21).

However, one fundamental condition to successfully perform this attack is that the attacker pod and victim pod need to be in the same worker node. The exploitation follows these steps:

1. Deploy a privileged pod using the YAML file in Figure 6.20. Note that the pod needs to mount the */proc* directory from the host.

2. Using */proc*, we can view the entire list of IDs of processes running

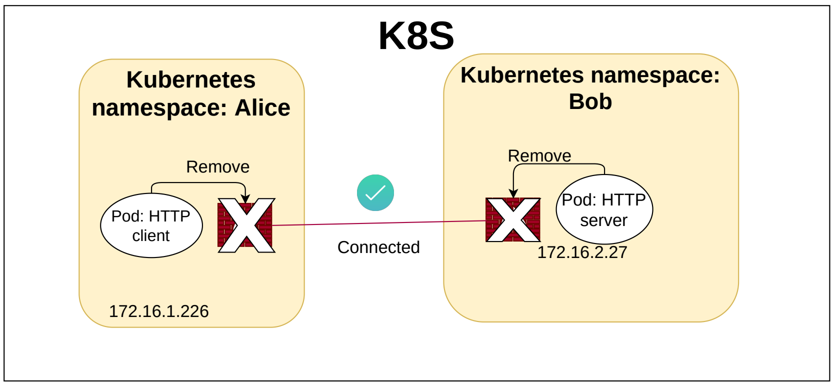

Figure 6.18: An overview of experiment 7 in the security evaluation

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test$
    kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
    the containers in this pod.
/ # wget 172.16.2.27
Connecting to 172.16.2.27 (172.16.2.27:80)
index.html           100%
    |********************************************|   612
    0:00:00 ETA
/ #
```

Figure 6.19: The connection in the experiment 7 is allowed as protection rules have been removed

on the host. This list includes IDs of the processes running in the victim pods. Bruce force [38] attack can help to determine the right IDs belonging to the victim. However, for the testing purpose, in this experiment, we find this ID manually.

3. To manually find the process ID, it is possible to ssh to the worker node, then run the command as in Figure 6.22.

4. The obtained ID is the input of a tool named Nsenter [23]. This tool allows executing commands in the network namespace of a specific process. Run the command as in Figure 6.23 to drop the iptables rules of the victim.

5. Finally, the tenant Alice successfully connected to the tenant Bob (see Figure 6.24).

```
apiVersion: v1
kind: Pod
metadata:
  name: attacker
  labels:
    env: test
spec:
  # shareProcessNamespace: true # To share the host PID space
  containers:
  - name: attacker
    image: docker.io/namnx228/k8s-evaluation-security-
   attacker-amd64
    imagePullPolicy: Always
    command:
      - sleep
      - "3600"
    volumeMounts:
    - mountPath: /hosts/proc/
      name: proc-node-1

    securityContext:
      privileged: true # ---> Root permission
  nodeSelector:
      kubernetes.io/hostname: node-1
  restartPolicy: Always
  volumes:
  - name: proc-node-1
    hostPath:
      path: /proc #----> mount the /proc file system
      type: Directory
```

Figure 6.20: An example of the privileged pod

The attack enables a tenant to intentionally remove another tenant's security rules without their acknowledge and approval, thus wholly compromising the isolation. However, one limitation of this exploitation is that the attacker can only compromise the isolation of pods running within a common worker node. Besides, the ability to create privileged pods empowers the tenant to break the network isolation and compromise the whole worker node. As a result, it is recommended to ban this critical feature. However, tenants may require higher privileges than the default configuration to perform particular operations (e.g., to ping, to access iptables). This situation can be handled by using Linux capabilities. This Linux feature breaks the mighty privilege of the root user into several smaller capabilities [45]. In total, Linux offers 40 capabilities.

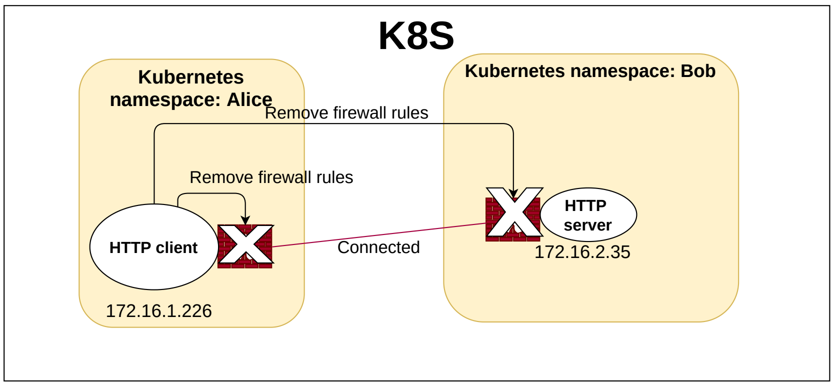

Figure 6.21: An overview of experiment 8 in the security evaluation

```
vagrant@node-2:~$ docker ps | awk '{ for (i=NF; i>1; i--) printf("%s ",$i); print $1; }'
      | awk '{print $1}' | grep nginx-deployment-6b474476c4-vkjlr # --> Name of the
      victim pod
k8s_perf-sidecar_nginx-deployment-6b474476c4-vkjlr_bob_b6113e61-3f7f-4aee-bfb4-7
      cd1f9beef20_0
k8s_nginx_nginx-deployment-6b474476c4-vkjlr_bob_b6113e61-3f7f-4aee-bfb4-7cd1f9beef20_0
k8s_POD_nginx-deployment-6b474476c4-vkjlr_bob_b6113e61-3f7f-4aee-bfb4-7cd1f9beef20_0
vagrant@node-2:~$ docker top k8s_nginx_nginx-deployment-6b474476c4-vkjlr_bob_b6113e61-3
      f7f-4aee-bfb4-7cd1f9beef20_0 # --> Container inside the victim pod
UID                       PID
root                      2009 (the target PID )
systemd+                  2032
vagrant@node-2:~$
```

Figure 6.22: Commands to manually obtain the PID of a process running in the victim pod

We investigated these capabilities and found three capabilities that are required to perform the attack. They are *CAP_SYS_ADMIN*, *CAP_NET_RAW* and *CAP_NET_ADMIN*. The first capability grants the ability to access another network namespace, while the other two are required to access iptables. To replace the privileged pod, we deployed a new pod which was granted these three capabilities. Then, we repeated the exploitation steps in the experiment. As was our expectation, the result was the same. The new pod granted the three capabilities in tenant Alice can connect to the HTTP pod in the namespace Bob. The definition of the new pod can be found in Figure 6.25.

From this result, we invent an idea that bans the use of these capabilities to prevent the attack. *CAP_SYS_ADMIN* is recommended not to use it in products [9], as it is called “the new root” and it overlaps many other capabilities. Therefore, this capability should not be granted. The two other capabilities cannot be avoided because the sidecar containers need them to modify the iptables inside the pod network namespace.

```
root@attacker:/# nsenter -t 2009 --net=/hosts/proc/2009/ns/net iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  172.16.2.0/24        172.16.2.0/24
DROP       all  --  172.16.0.0/14        172.16.0.0/14

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  172.16.2.0/24        172.16.2.0/24
DROP       all  --  172.16.0.0/14        172.16.0.0/14
root@attacker:/# nsenter -t 2009 --net=/hosts/proc/2009/ns/net iptables -F
root@attacker:/# nsenter -t 2009 --net=/hosts/proc/2009/ns/net iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@attacker:/#
```

Figure 6.23: The command removes all iptables rules of the victim

To sum up, the solution for this attack is to disallow the use of privileged pods and the capability *CAP_SYS_ADMIN*. This restriction is straightforward and does not hinder the tenant experience. Tenants can continue to perform operations requiring higher privileges while not able to access each other's network namespaces. With these countermeasures in place, we evaluate the risk level of this attack as low.

In conclusion, our network isolation solution passes most of the security tests in the evaluation. The remaining security issues are not significant. Table 6.1 summarizes the results, risk levels, and solutions of the experiments in this section.

Tables 6.2 summarizes vulnerabilities of our network isolation method.

```
alice@tenant-machine:/home/vagrant/MasterThesis/main-test/
    evaluation/security$ kubectl exec dnsutils -it -- /bin/sh
Defaulting container name to dnsutils.
Use 'kubectl describe pod/dnsutils -n alice' to see all of
    the containers in this pod.
/ # wget 172.16.2.35
Connecting to 172.16.2.35 (172.16.2.35:80)
index.html           100% |**************************|   612
      0:00:00 ETA
```

Figure 6.24: The pod in tenant Alice can bypass the isolation to connect to pod in tenant Bob

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: attacker-cap-sys-admin
  labels:
    env: test
spec:
  # shareProcessNamespace: true # To share the host PID space
  containers:
  - name: attackercapsysadmin
    image: docker.io/namnx228/k8s-evaluation-security-
   attacker-amd64
    imagePullPolicy: Always
    volumeMounts:
    - mountPath: /hosts/proc/
      name: proc-node-1
    command:
      - sleep
      - "3600"
    securityContext:
      # privileged: true # ---> Root permission
      capabilities:
        add: ["SYS_ADMIN", "CAP_NET_RAW", "CAP_NET_ADMIN"]
  nodeSelector:
      kubernetes.io/hostname: node-1
  restartPolicy: Always
  volumes:
  - name: proc-node-1
    hostPath:
      path: /proc #----> mount the /proc file system
      type: Directory
```

Figure 6.25: The YAML definition of the alternative attack pod

Experiment	Description	Result	Match expectation	Risk level	Solution
Experiment 1	Two pods from different Kubernetes namespaces	Blocked	✓		
Experiment 2	Two pods from different Kubernetes namespaces (use service)	Blocked	✓		
Experiment 3	Two pods from one Kubernetes namespace	Allowed	✓		
Experiment 4	Two pods from one Kubernetes namespace (use service)	Allowed	✓		
Experiment 5	Pod connects to external services	Allowed		Low	
Experiment 6	Attacker drops her firewall	Blocked	✓		
Experiment 7	Bob unintentionally disables firewall rules	Allowed		Medium	
Experiment 8	Privileged pod	Allowed		Low	✓

Table 6.1: Summary of the experiments in the security evaluation

Security issue	Experiment	Solution available ?	Risk level
Tenant pods exposed on the internet	5		Low
Victim unintentionally disables firewall rules	7		Medium
Privileged pod deployment	8	✓	Low

Table 6.2: Summary of security issues of the network isolation method

Chapter 7

Performance Evaluation

7.1 Test Environment

The motivation of the performance evaluation is that when an additional container is injected into a pod, the cluster may require more time to process tenant requests, such as creating pods, deleting pods. Besides, the use of iptables as the firewall for isolation can cause network delay. This delay needs to be acknowledged so that we can understand its performance impact on the cluster.

The experiments in this section are conducted on a cluster shared by two tenants Alice and Bob. The configuration of these tenants is similar to the configuration in Section 5.2. However, to measure the impact of our solution, we create two security profiles. The first profile does not apply the network isolation solution, and it is assigned to tenant Alice. The second profile applies the solution, and tenant Bob adopts this profile. In other words, we disable the network isolation solution on tenant Alice and enable it in tenant Bob (see Figure 7.1).

Table 7.1 describes the the two security profiles mentioned above.

The experiments in this section are carried out with only two tenants. While the test environment setup script can deploy up to 1000 tenants, it is unnecessary and time-inefficient to do that. Therefore we only report results obtained from a two-tenant cluster. The details of this can be found in

Profile name	Kubernetes namespace	Network isolation solution
Insecure profile	Alice	Disabled
Secure profile	Bob	Enabled

Table 7.1: Description of security profiles

```
$ kubectl get ns -L sidecar-injector
NAME              STATUS   AGE   SIDECAR-INJECTOR
default           Active   55d
kube-node-lease   Active   55d
kube-public       Active   55d
kube-system       Active   55d
alice             Active   29d   disable
bob               Active   29d   enabled
```

Figure 7.1: The network isolation solution is disabled for the Kubernetes namespace Alice and enabled for the namespace Bob

Chapter 8.

In a distributed system like a Kubernetes cluster, many system components can indirectly influence the performance and create unpredictable performance hits. These uncontrollable elements become noise that affects the experiment results. Besides, the experiments are carried out on the virtual platform that also produces a certain amount of noise from the hypervisor activities and other processes running on the physical host. Thus, it is essential to remove or at least minimize the amount of noise in the experiments. Subsequently, each experiment needs to run in several iterations. Then, average, standard deviation, and other statistic parameters are calculated on the results obtained from these iterations. The definition of one iteration depends on each experiment. However, an experiment does not finish its iterations before starting the next one. For example, an experiment with **N** iterations runs **N/10** iterations in 10 different chucks. The purpose of this action is that an unusual noise can occur over some time period, but it can only affect the result of a small number of iterations. Therefore, running in 10 separate moments lessens the influence of this noise to the experiment.

We organize and schedule to run the experiments as the algorithm in Figure 7.2. The outside loop consists of ten iterations. In the first iteration, function *Experiment1_Insecure_N/10* runs the first $N_{A1}/10$ iterations of the experiment 1, with N_{A1} is the total number of iteration in experiment 1 in the *Insecure profile*. Next, the function *Experiment1_Secure_N/10* runs the first $N_{B1}/10$ iterations of the experiment 1, with N_{B1} is the total number of iteration in experiment 1 in the *Secure profile*. It is similar to other functions performing other experiments. After the function *Experiment6_Secure_N/10* finishes running, the next big iteration is executed until the iteration 10. Besides, it is worth mentioning that the results obtained from each **N/10** iterations in the function are used to compute the average of these results,

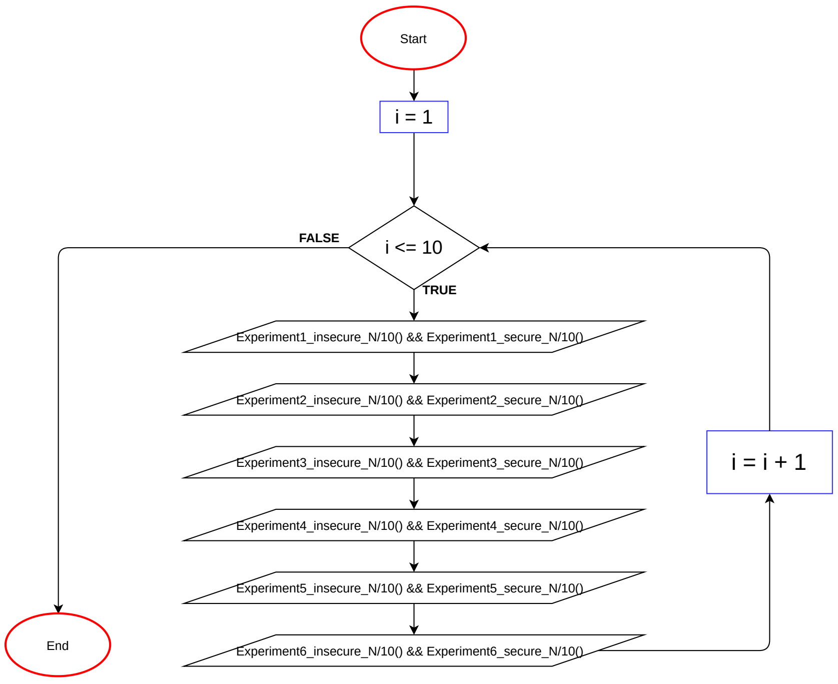
Start
i = 1
i <= 10
FALSE
TRUE
Experiment1_insecure_N/10() && Experiment1_secure_N/10()
Experiment2_insecure_N/10() && Experiment2_secure_N/10()
Experiment3_insecure_N/10() && Experiment3_secure_N/10()
Experiment4_insecure_N/10() && Experiment4_secure_N/10()
Experiment5_insecure_N/10() && Experiment5_secure_N/10()
Experiment6_insecure_N/10() && Experiment6_secure_N/10()
i = i + 1
End

Figure 7.2: The algorithm to runs the experiments in this section

then this average value is forwarded to other statistical measurements.

7.1.1 Statistical Functions

This section introduces the list of statistical methods adopted to scientifically measure the difference in performance between the two security profiles.

1. Mean [72] is used as the average value of the results of a function after each batch finishes.

2. F-Test [62] is employed to calculate the probability that the variances of two populations are not significantly unequal. In this thesis, the two populations are the test results of the two security profiles.

3. T-Test [62] measures the probability that the means of two populations are statistically similar. Depending on the result of the F-Test, if the variances of the two groups are equal, the two-sample equal variances version of the T-Test is applied. Otherwise, the unequal variances version is picked up. Besides, as the assumption is that the *secure profile* may produce a delay, the mean of the population representing this profile is regularly bigger than the mean of the *insecure profile*. As

a result, we apply the one-tailed T-Test [15], only check either one mean value is bigger than the other, but not both cases. The one-tailed test provides more power to detect an effect in one direction, in comparison to the two-tailed test that tests the probability of both directions: that the first mean is significantly greater than the second mean, and that the first mean is less than the second one.

7.2 Experiment 1

This experiment investigates the delay in the pod creation process that our network isolation solution can cause. The delay may come from the sidecar containers added by the solution. To find this delay, we measured the pod deployment time in both security profiles and compared them. With each profile, P pods are deployed concurrently, and we measured the duration from the beginning until a completion signal was triggered. The number of pods P is assigned in turn to each value in the set $\{2, 5, 10, 15, 20, 25, 30, 35, 40\}$. The tool used for time measurement is the built-in *time* command of the Bash terminal [56]. The completion signal is the element *replicas* in the JSON output obtained after we send the deployment status request. If the value of this element equals to P, the deployment process is considered as completed. The full command to send this request and filter out the *replicas* element can be found in Figure 7.3.

```
kubectl get deployments.apps ${DEPLOYMENT_NAME} -o jsonpath
    ="{.status.replicas}"
```

Figure 7.3: Command used to receive and filter out the element *replicas*

The result of this process is the duration of the pod deployment process in milliseconds (ms). The whole process was repeated multiple times. In total, 300 iterations were performed, divided into ten 30-iteration-chunks. Each chunk returned a value that is the average duration of the iterations in that chunk. Table 7.2 and Table 7.3 present the average value of ten chunks for each number of pods (NoP) in the *insecure profile* and the *security profile*, respectively.

For each NoP, we calculated the mean of 10 chunks, then compared these mean values between the two profiles. The mean of 10 chunks is similar to the mean of 300 iterations, and this chunking approach helps simplify the measurement process while not losing much accuracy. Figure 7.4 indicates the comparison between the mean values of the two profiles.

Chunk / NoP	1	2	3	4	5	6	7	8	9	10
NoP 2	0.11	0.09	0.11	0.11	0.12	0.10	0.11	0.09	0.10	0.09
NoP 5	0.14	0.15	0.14	0.13	0.15	0.16	0.15	0.14	0.12	0.14
NoP 10	0.21	0.20	0.21	0.21	0.23	0.23	0.20	0.18	0.19	0.22
NoP 15	0.26	0.24	0.24	0.27	0.29	0.29	0.27	0.23	0.28	0.24
NoP 20	0.27	0.34	0.30	0.29	0.33	0.34	0.36	0.26	0.34	0.37
NoP 25	0.43	0.46	0.43	0.50	0.54	0.47	0.78	0.35	0.42	0.41
NoP 30	0.58	0.76	0.66	0.80	0.81	1.10	0.96	0.56	0.61	0.65
NoP 35	0.94	1.09	1.00	1.46	1.51	1.26	1.41	0.85	0.97	0.98
NoP 40	1.22	1.36	1.37	1.69	1.56	1.50	1.84	1.08	1.22	1.26

Table 7.2: The results of ten chunks in the *insecure profile* in experiment 1 (seconds)

Chunk / NoP	1	2	3	4	5	6	7	8	9	10
NoP 2	0.20	0.17	0.11	0.11	0.11	0.11	0.12	1.13	0.11	0.11
NoP 5	0.21	0.19	0.21	0.17	0.22	0.22	0.19	0.24	0.24	0.19
NoP 10	0.29	0.27	0.25	0.40	0.34	0.46	0.32	0.31	0.35	0.30
NoP 15	0.44	0.46	0.33	0.54	0.45	0.81	0.54	0.47	0.40	0.45
NoP 20	0.65	0.73	0.71	0.94	1.18	0.99	0.89	0.54	0.59	0.56
NoP 25	0.88	0.93	1.22	1.02	0.93	1.21	1.03	0.68	0.62	0.61
NoP 30	1.01	1.13	1.14	1.25	1.43	1.34	1.25	0.84	0.88	0.82
NoP 35	1.33	1.64	1.34	1.66	1.62	1.77	1.49	1.19	1.15	1.12
NoP 40	1.75	1.92	1.99	1.92	2.09	2.05	1.86	1.54	1.44	1.44

Table 7.3: The results of ten chunks in the *secure profile* in experiment 1 (seconds)

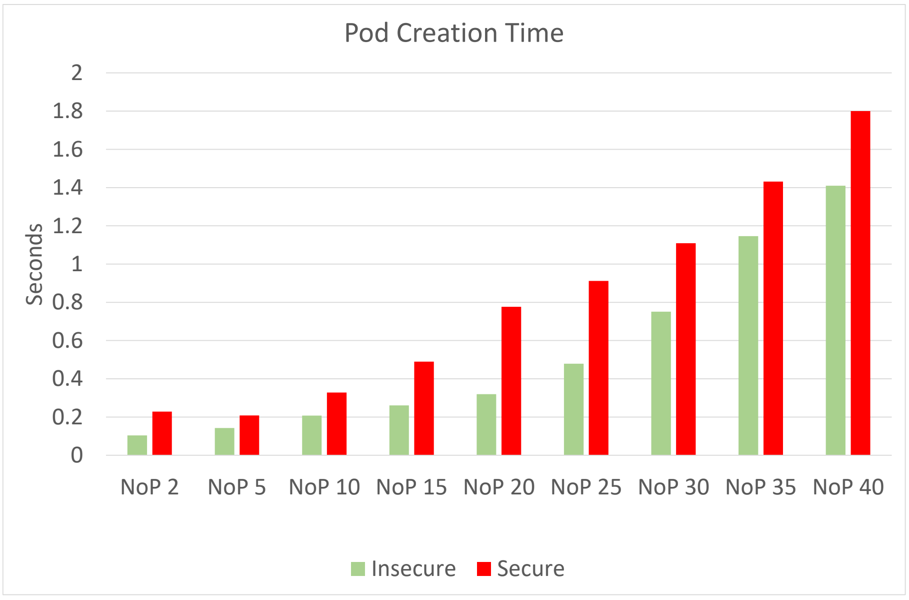

Figure 7.4: A comparison between means of chunks in the two security profiles in experiment 1

According to this graph, the means of the *insecure profile* are smaller than the *secure profile* for the whole range of NoP. Moreover, T-Test and F-Test (as mentioned in Section 7.1.1) are taken to provide more scientific evidence. Details of the F-Test are as following:

1. Null hypothesis H_0: The variances of the two lists are equal.
2. Alternatives hypothesis H_1: The variances of the two lists are unequal.
3. Input: The lists of 10 chunks of the two security profiles.
4. Significance level: 0.05. This means that the confidence level of the test is 95%.
5. p-value: The probability that two variances are equal. It is identified by the test.
6. Output: If the p-value is smaller than the significance level, the null hypothesis can be rejected.

Number of pods	F-Test	T-Test	Reject Null hypothesis
NoP 2	6.33E-12	0.12	
NoP 5	0.05	9.73E-07	✓
NoP 10	0.0004	6.85E-05	✓
NoP 15	1.69E-05	0.0001	✓
NoP 20	1.69E-05	3.61E-05	✓
NoP 25	0.07	1.73E-05	✓
NoP 30	0.52	0.0003	✓
NoP 35	0.94	0.008	✓
NoP 40	0.88	0.0009	✓

Table 7.4: Results of the F-Test and T-Test in experiment 1

The output of F-Test decides whether using the equal variances T-Test or unequal variances T-Test. Details of the one-tailed T-Test are as following:

1. Null hypothesis H_0: The means of two lists are equal.
2. Alternatives hypothesis H_1: The means of two lists are unequal.
3. Input: The lists of 10 chunks of the two security profiles; whether the two variances are equal or not.
4. Significance level: 0.05. This means that the confidence level of the test is 95%.
5. p-value: The probability that two means are equal. It is identified by the test.
6. Output: If the p-value is smaller than the significance level, the null hypothesis can be rejected.

Table 7.4 presents the result of the tests mentioned above. The table shows that the null hypothesis is rejected for the entire range of NoP. Rejecting the null hypothesis means that two mean values are statistical different, and it implicates that the mean of the *insecure profile* is smaller than the *secure profile*. We can conclude that the network isolation solution produces a delay to the pod deployment process. However, Figure 7.4 shows that the delay is within the bounds of acceptable.

Chunk / NoP	1	2	3	4	5	6	7	8	9	10
NoP 2	7.40	5.92	11.31	8.69	8.06	7.03	9.47	9.62	10.37	6.82
NoP 5	6.52	26.58	12.84	12.31	10.75	11.15	12.84	11.34	12.09	10.99
NoP 10	9.75	10.13	9.78	9.33	9.84	9.76	10.12	14.89	24.76	8.58
NoP 15	10.74	12.60	9.74	9.82	20.16	33.99	9.29	8.59	11.36	6.93
NoP 20	15.50	29.50	36.67	29.69	10.87	25.77	22.90	15.51	11.36	31.19
NoP 25	10.25	13.36	15.91	29.64	22.79	22.53	29.54	27.75	46.31	35.36
NoP 30	34.86	28.61	32.17	21.65	30.76	56.44	42.39	29.86	55.96	46.07
NoP 35	22.63	35.80	42.85	30.59	35.82	38.75	34.11	41.73	41.15	39.94
NoP 40	48.67	15.34	51.63	58.64	38.91	46.06	25.13	31.73	15.64	41.43

Table 7.5: The results of ten chunks in the *insecure profile* in experiment 2 (seconds)

7.3 Experiment 2

This experiment determines whether our solution creates a delay in the pod deletion process. In the experiment, we apply a similar setup and statistic methods to experiment 1. However, to detect the event that the pod deletion process finishes, we continuously send commands to check the number of remaining pods. If the entire list of pods is empty, the deletion process has finished. Table 7.5 and Table 7.6 describe the results of ten chunks in the *insecure profile* and the *secure profile.*

Table 7.7 shows the result of the F-Test and T-Test. It can be seen from the table that the null hypothesis is rejected for six of the nine cases. Unfortunately, it is hard to draw a conclusion based on this result. In fact, the pod deletion process contains a great deal of noise due to the Kubernetes cluster itself. Therefore, measuring the pod deletion time is an unreliable approach.

7.4 Experiment 3

This experiment and the rest verify the assumption that the use of iptables in the pods can produce network delay because the Netfilter needs to match packets to the whole list of iptables rules. This experiment examines the case of high bandwidth connections. The setup of the experiment is explained in Figure 7.6.

Iperf [69] was adopted in this experiment to create network communication between the pods. It is a powerful tool to measure the maximum

NoP \ Chunk	1	2	3	4	5	6	7	8	9	10
NoP 2	7.42	10.29	20.10	19.28	30.60	18.82	33.19	9.99	22.78	21.10
NoP 5	20.02	10.11	10.28	18.04	7.07	11.37	6.77	17.76	28.51	15.56
NoP 10	6.03	6.41	22.57	6.45	6.06	6.28	6.33	31.27	7.40	31.57
NoP 15	22.29	9.22	30.62	32.43	19.01	20.08	29.23	44.46	36.16	30.79
NoP 20	35.90	46.57	48.44	19.41	32.78	45.33	42.80	42.74	44.66	46.20
NoP 25	45.91	42.65	42.89	49.93	36.03	32.97	39.49	24.32	47.13	46.08
NoP 30	55.87	35.41	49.30	55.62	23.05	32.69	46.05	47.73	44.68	59.24
NoP 35	59.50	62.55	55.65	45.65	46.43	49.96	42.39	45.87	55.78	45.74
NoP 40	45.12	71.95	58.78	55.64	61.69	58.48	72.21	65.37	68.90	64.25

Table 7.6: The results of ten chunks in the *secure profile* in experiment 2 (seconds)

Number of pods	F-test	T-Test	Reject Null hypothesis
NoP 2	5.80788E-05	0.001362449	✓
NoP 5	0.439941611	0.255386815	
NoP 10	0.025406248	0.363944698	
NoP 15	0.539794053	0.001368701	✓
NoP 20	0.944893459	0.000181504	✓
NoP 25	0.335025209	0.000909529	✓
NoP 30	0.925574402	0.095644236	
NoP 35	0.735908402	4.63194E-05	✓
NoP 40	0.091635587	0.000109036	✓

Table 7.7: Results of the F-Test and T-Test in experiment 2

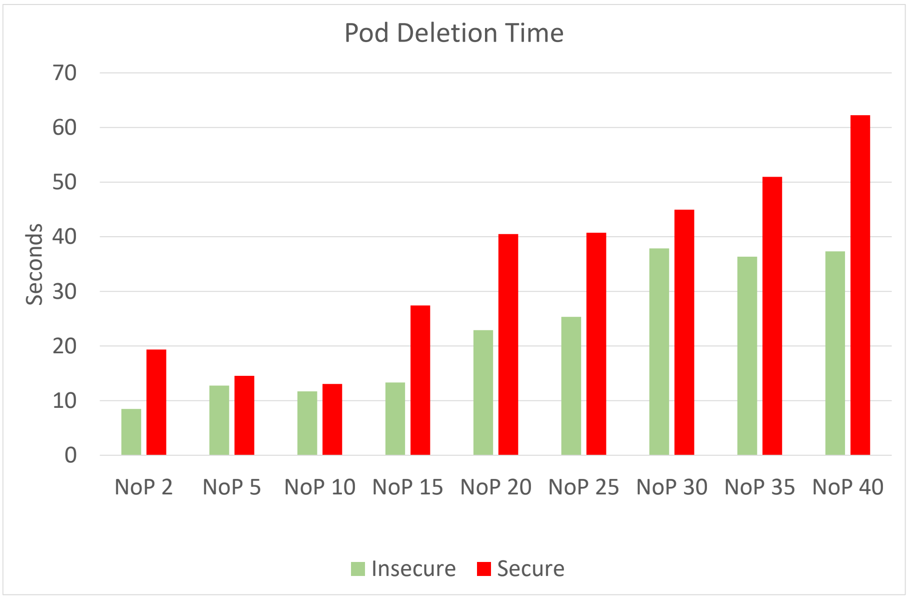

Figure 7.5: A comparison between means of chunks in the two security profiles in experiment 2

bandwidth on IP networks. For each Kubernetes namespace in the security profiles, we deployed two pods. One pod ran an Iperf server using the command in Figure 7.7. The server was listening on the port 5000. The second pod ran an Iperf client using the command in Figure 7.8. The parameter "-c" means the target that is connected by this client. In this case, we deploy a service that forwards traffic to the Iperf server, and connect the client to this service. "-t" means the amount of time the client sends traffic to the server. "-i" means the duration between two periodic bandwidth reports. "-p" is the server port. The amount of communication time is set to 30 seconds, and during this time, the client measured the bandwidth. After the communication had finished, the client reported the bandwidth in megabits per second. This experiment was repeated ten times based on the algorithm in Section 7.1. The results of ten iterations are presented in Table 7.8.

The mean values of iterations in the two profiles are compared in Figure 7.9. The result of F-Test and T-Test are as following:

1. F-Test: 0.292257703 $>$ 0.05 Therefore, the equal T-Test is selected in this experiment.

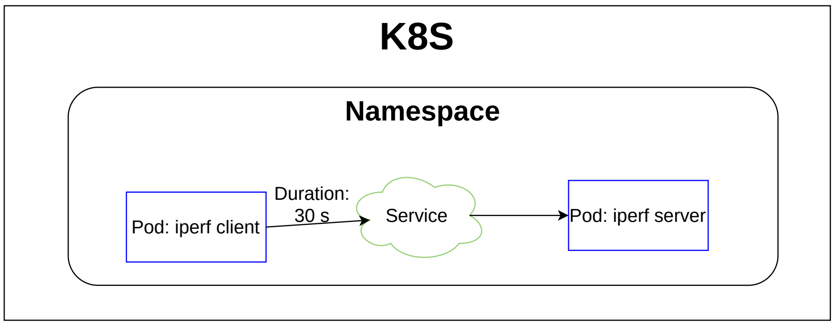

Figure 7.6: Setup of experiment 3 in the performance evaluation

```
iperf -s -p 5000
```

Figure 7.7: The command used to deploy an Iperf server

2. T-Test: $1.99658E - 06 < 0.05$. Therefore, the null hypothesis can be rejected.

According to the result of T-Test, the network isolation solution did cause a bandwidth loss. However, it is shown in Figure 7.9 that the impact is not excessive. The reason is that the number of iptables rules added to the pod is small, only four. Therefore, it required a negligible amount of time to process a packet through these rules.

7.5 Experiment 4

This experiment tests the case when a large number of connections are simultaneous active between two pods. It aims to answer whether the iptables rules added by the sidecar container reduce the aggregate bandwidth of these connections. The setup is similar to the experiment 3. The only exception is the command that Iperf client used to deploy a large number of connections to the server. The Figure 7.10 describes this command. "-P 100" means the client generates 100 connections to the server.

Test No.	1	2	3	4	5	6	7	8	9	10
Insecure	2645	2765	2736	2409	2196	2720	2717	2744	2729	2689
Secure	1877	2236	1661	1719	2259	2261	1694	2297	1816	1710

Table 7.8: Results of ten iterations in experiment 3 (Mb/s)

```
iperf -c ${SERVICE} -t 30 -i 30 -p 5000 -f m
```

Figure 7.8: The command used to deploy an Iperf client

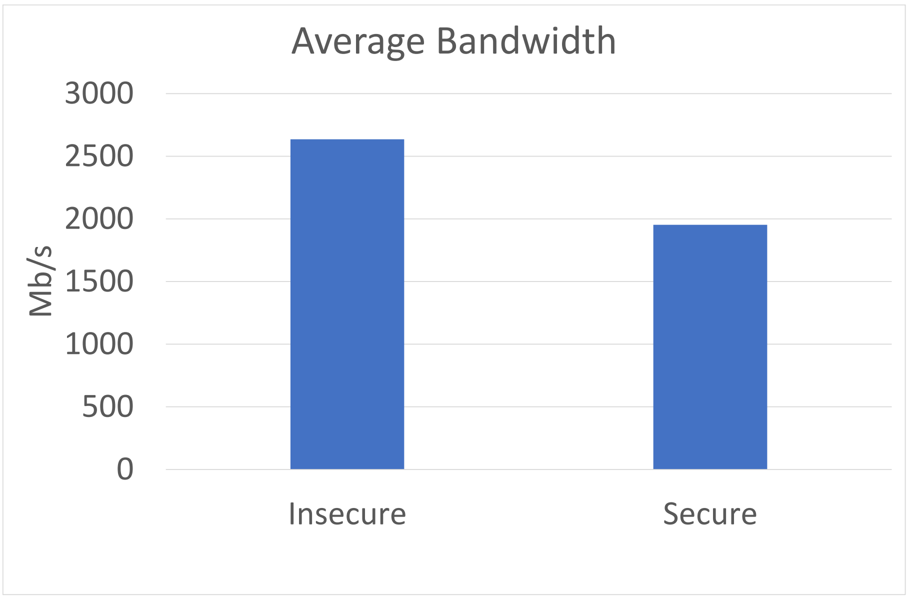

Figure 7.9: A comparison between means of chunks in the two security profiles in experiment 3

Table 7.9 presents the results of ten iterations. Figure 7.11 compares the means of the two profiles. The results of F-Test and T-Test are as following:

1. F-Test: $0.462198796 > 0.05$ Therefore, the equal T-Test is selected in this experiment
2. T-Test: $2.05956E - 06 < 0.05$. Therefore, the null hypothesis can be rejected.

According to the result of the T-Test, our solution caused a bandwidth loss in this case. However, as shown in Figure 7.11, it was not excessive as the number of iptables rules needed to be processed is small.

```
iperf -c  ${SERVICE} -t 30 -i 30 -p 5000 -f m -P 100
```

Figure 7.10: The command used to deploy an Iperf client in experiment 4

Test No.	1	2	3	4	5	6	7	8	9	10
Insecure	826	819	806	810	806	836	802	812	834	763
Secure	722	755	720	764	755	751	684	768	726	766

Table 7.9: Results of ten iterations in experiment 4 (Mb/s)

7.6 Experiment 5

This experiment measures the round trip time (RTT) of packets traveling between two pods and determines if the network isolation method increases the RTTs. The setup of this experiment consisted of two pods deployed in the same Kubernetes namespace. One pod ran a Hping3 client. Hping3 [64] is an open-source packet generator and analyzer for the TCP protocol. This tool is utilized in this experiment to send TCP packets and calculate their RTTs. The command used to run the client can be found in Figure 7.12. The second pod ran an Iperf server.

After being deployed, the client sent and measured the RTTs of 1000 TCP packets, then computed the average of these RTTs. As following the algorithm in Section 7.1, the experiment was repeated ten times, therefore in total, 10000 packets were sent. Table 7.10 recorded the average values obtained from these ten times running the experiment. The mean of the ten results are computed and compared in Figure 7.13. The result of F-Test and T-Test are as following:

1. F-Test: $0.052 > 0.05$ Therefore, the equal T-Test is selected in this experiment
2. T-Test: $0.048 < 0.05$. Therefore, the Null hypothesis can be rejected.

It can be seen from the result of the T-Test that the average RTT in the *secure profile* was longer than in the *insecure profile*, and the deployment of our solution caused this considerable additional amount of time.

Test No.	1	2	3	4	5	6	7	8	9	10
Insecure	0.42	0.43	0.28	0.31	1.87	1.29	2.08	0.34	3.37	1.40
Secure	0.30	0.99	0.33	0.26	1.64	0.22	0.24	0.93	0.25	0.24

Table 7.10: Results of ten iterations in experiment 5 (ms)

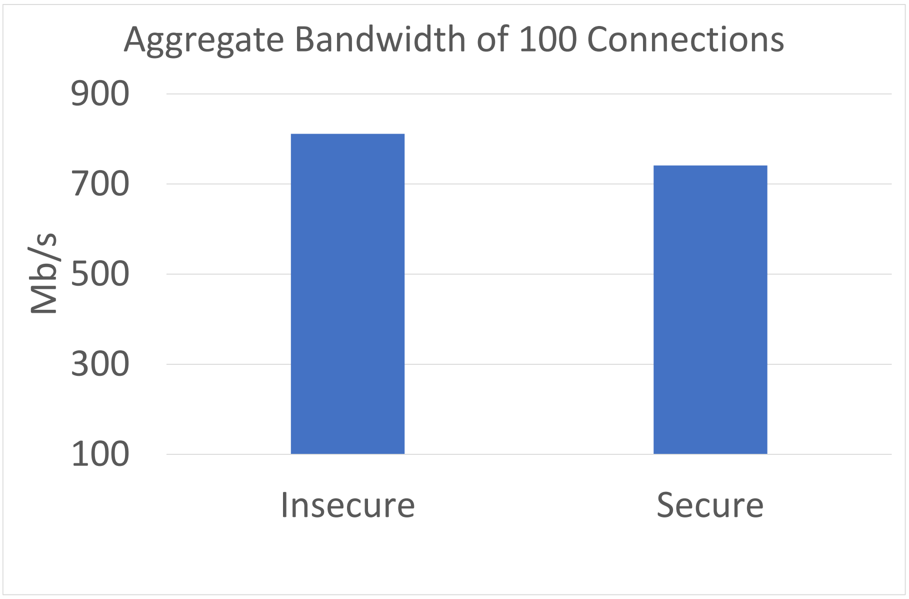

Figure 7.11: A comparison between means of chunks in two security profiles in experiment 4

```
hping3 -S -p 8000 -c 1000 ${SERVICE}
```

Figure 7.12: The command used to deploy an Iperf client in experiment 5

7.7 Experiment 6

This experiment also checks if the RTTs increase due to our solution. However, instead of TCP packets, experiment 6 investigates the HTTP requests. The setup is roundly the same with the previous experiment. The only difference is the client pod and server pod. The server pod ran a Nginx server, while the client ran Curl [14], a widely known HTTP client on command-line in Linux. The results of the experiment are recorded in Table 7.11, and the means are compared in Figure 7.14.

The results of F-Test and T-Test are as following:

1. F-Test: $0.96 > 0.05$ Therefore, the equal T-Test is selected in this experiment.

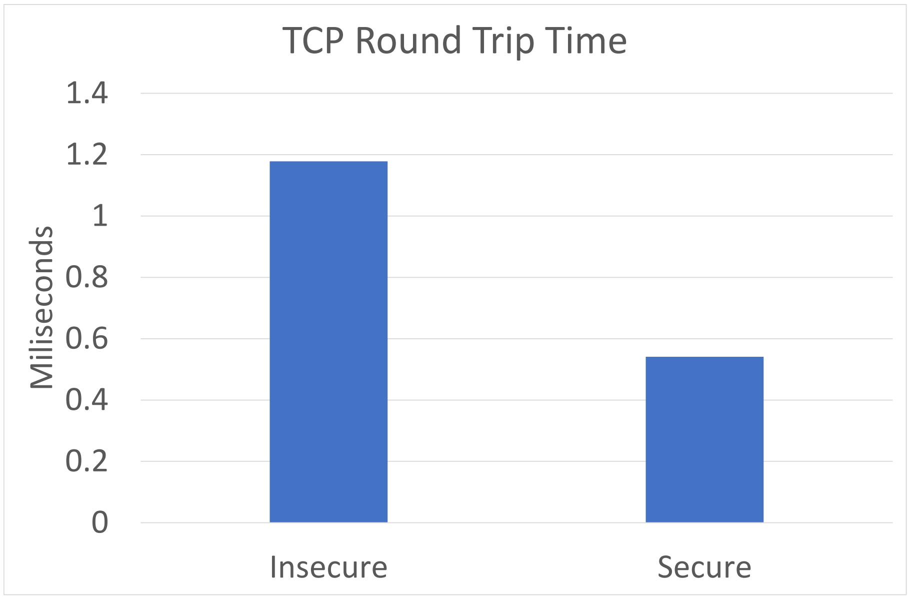

Figure 7.13: A comparison between means of chunks in the two security profiles in experiment 5

Test No.	1	2	3	4	5	6	7	8	9	10
Insecure	11.88	10.49	8.87	7.77	6.23	12.99	8.33	9.85	8.83	9.85
Secure	10.95	7.01	7.48	11.33	12.41	7.20	9.39	9.10	9.93	10.99

Table 7.11: Results of ten iterations in experiment 6 (ms)

2. T-Test: $0.47 > 0.05$. Therefore, the Null hypothesis cannot be rejected.

The T-Test output shows that the average RTT of HTTP requests in the *insecure profile* and the *secure profile* are not significantly different. As the test could not detect a delay in RTT, it could be negligible or not exist.

7.8 Summary

To sum up, we can conclude that our network isolation solution causes delay on cluster operations, including creating, possibly deleting pods, and network communication between pods. The delay in creating and deleting pods comes from the sidecar container added to each pod while the network delay comes

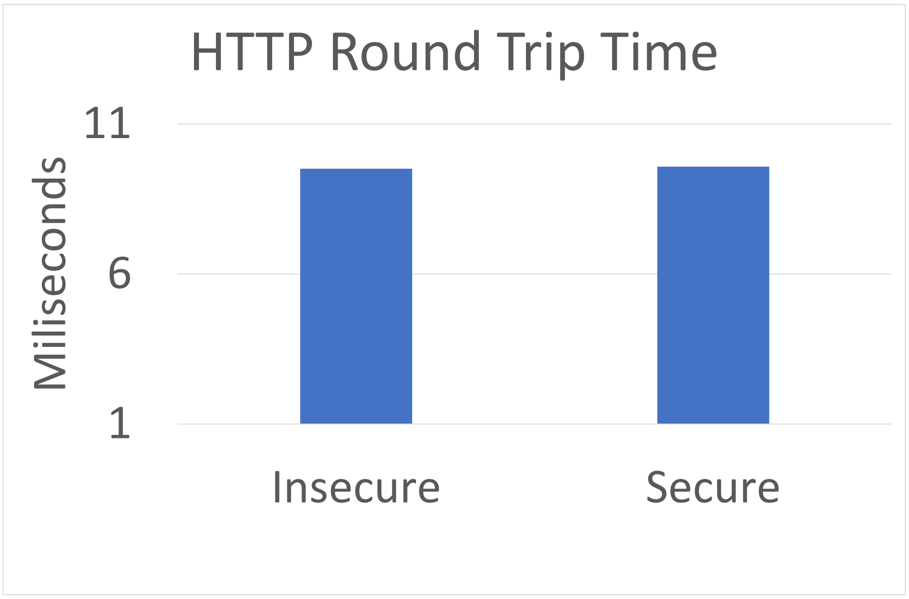

Figure 7.14: A comparison between means of chunks in the two security profiles in experiment 6

from the iptables usage to block cross-namespace (Kubernetes namespace) communication. However, the delay is noticeable but within the bounds of acceptable, because only one extra container and four iptables rules are added to a pod.

While we repeated the experiments several times to minimize noise, it is unavoidable in a virtual testing platform. However, since the delay is detected by running experiments on virtual machines, we can conclude that the network isolation solution will also produce the delay when applied on a native cloud platform.

Chapter 8

Conclusions and Discussion

8.1 Discussion

This section lists directions that we can move forward to improve the proposed solution. The first idea is to leverage the sidecar container. In the current solution, the sidecar container in a pod runs to set up the internal firewall once this pod is created. Our idea is to transform it to a watchdog sidecar container: it continues to run after setting up the firewall, and maintains the firewall configuration that provides network isolation. This approach can prevent unintentional or malicious behaviors that alters the firewall, as mentioned in Section 6.7, thus being one solution for this problem. However, it also has limitations. Firstly, a constantly running sidecar container would consume more resources, including computing, memory, and even networking, compared to a one-time run sidecar container. In other words, adopting watchdog containers may generate more performance delay to the cluster. Besides, this approach may not completely stop insider attackers. An insider attacker can bypass this watchdog by exploiting the a race condition vulnerability [70]. This vulnerability may come from the implementation of the watchdog. Even with these weaknesses, the watchdog can provide extra protection since launching a race condition attack may not be straightforward and requires much time and effort to prepare.

In addition, it is worth mentioning that our test environment can deploy up to 1000 tenants. However, we experimented and figured out that the number of tenants does not influence the performance test results. The reason is that the tenant number cannot affect the sidecar container inside each pod. The internal iptables in each pod is also independent of this number. Therefore, we can conclude that the tenant number cannot contribute to the delay created by our solution. Thus, two tenants are enough for our experiments.

Finally, as mentioned in Section 7.1, unpredictable behaviors of the virtualization hypervisor and other processes running on the physical host can create performance hits to the cluster, producing noise to our experiments. Even so, our tests could detect the loss of performance when we apply our solution, compared to the case that does not employ it. From this result, we can conclude that our solution would also produce performance loss when applied to a native cloud environment.

8.2 Conclusions

To sum up, this thesis introduces a network isolation method that can meet one critical requirement of Kubernetes hard multi-tenant systems. This solution can work regardless of the variety of Kubernetes network implementation since it leverages the internal firewall (iptables), an unchangeable component in Kubernetes, inside each pod. This firewall is set up by a sidecar container added to a pod by the admission controller. This solution can pass 5 out of 8 security tests, and 1 out of the remaining issues is solvable. Besides, we recorded delays in performance due to using our solution. The delay is observable but nevertheless acceptable in all the tested cases. Finally, based on this evidence, the proposed solution is feasible, and with the improvements mentioned in the previous section, it can be considered to become a part of a real Kubernetes multi-tenant cluster that provides fine-grained and reliable security.

Bibliography

[1] stderr - C++ Reference. https://www.cplusplus.com/reference/cstdio/stderr/?kw=stderr, Dec 2013. [Online; accessed 1. Jul. 2020].

[2] stdin - C++ Reference. https://www.cplusplus.com/reference/cstdio/stdin, Dec 2013. [Online; accessed 1. Jul. 2020].

[3] Flannel-CNI. https://github.com/coreos/flannel-cni, Sep 2017. [Online; accessed 8. May 2020].

[4] Ansible is Simple IT Automation. https://www.ansible.com, May 2020. [Online; accessed 31. May 2020].

[5] Authenticating. https://kubernetes.io/docs/reference/access-authn-authz/authentication, Jun 2020. [Online; accessed 3. Jun. 2020].

[6] Building large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large, Jul 2020. [Online; accessed 6. Jul. 2020].

[7] Calico architecture. https://docs.projectcalico.org/reference/architecture/overview#felix, Jul 2020. [Online; accessed 28. Jul. 2020].

[8] Calicoctl user reference. https://docs.projectcalico.org/reference/calicoctl/overview, Jul 2020. [Online; accessed 29. Jul. 2020].

[9] Capabilities(7) - Linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html, Jul 2020. [Online; accessed 2. Jul. 2020].

[10] Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking, Apr 2020. [Online; accessed 18. Apr. 2020].

[11] CNI. https://github.com/containernetworking/cni, Apr 2020. [Online; accessed 8. May 2020].

[12] Concepts. https://kubernetes.io/docs/concepts, May 2020. [Online; accessed 7. May 2020].

[13] Configure Service Accounts for Pods. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account, Jun 2020. [Online; accessed 3. Jun. 2020].

[14] Curl. https://curl.haxx.se, Jul 2020. [Online; accessed 2. Jul. 2020].

[15] FAQ: What are the differences between one-tailed and two-tailed tests? https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-what-are-the-differences-between-one-tailed-and-two-tailed-tests, Jul 2020. [Online; accessed 2. Jul. 2020].

[16] Global network policy. https://docs.projectcalico.org/reference/resources/globalnetworkpolicy, Jul 2020. [Online; accessed 5. Jul. 2020].

[17] Install Calico for policy and flannel (aka Canal) for networking. https://docs.projectcalico.org/getting-started/kubernetes/flannel/flannel, Jul 2020. [Online; accessed 28. Jul. 2020].

[18] ip-netns(8) - Linux manual page. https://man7.org/linux/man-pages/man8/ip-netns.8.html, Jul 2020. [Online; accessed 7. Jul. 2020].

[19] Istio. https://github.com/istio/istio, May 2020. [Online; accessed 9. May 2020].

[20] Multi-tenancy. https://github.com/kubernetes-sigs/multi-tenancy, May 2020. [Online; accessed 9. May 2020].

[21] Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces, Apr 2020. [Online; accessed 9. Apr. 2020].

[22] Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies, May 2020. [Online; accessed 8. May 2020].

[23] nsenter(1) - Linux manual page. https://man7.org/linux/man-pages/man1/nsenter.1.html, Jul 2020. [Online; accessed 1. Jul. 2020].

[24] Overview of kubectl. https://kubernetes.io/docs/reference/kubectl/overview, Jun 2020. [Online; accessed 3. Jun. 2020].

[25] Pods. https://kubernetes.io/docs/concepts/workloads/pods/pod, Apr 2020. [Online; accessed 8. Apr. 2020].

[26] Production-Grade Container Orchestration. https://kubernetes.io, May 2020. [Online; accessed 9. May 2020].

[27] Race condition | Wikiwand. https://www.wikiwand.com/en/Race_condition, May 2020. [Online; accessed 25. May 2020].

[28] Resource Quotas. https://kubernetes.io/docs/concepts/policy/resource-quotas, May 2020. [Online; accessed 5. May 2020].

[29] The Kubernetes API. https://kubernetes.io/docs/concepts/overview/kubernetes-api, Jul 2020. [Online; accessed 7. Jul. 2020].

[30] Using Admission Controllers. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do, May 2020. [Online; accessed 25. May 2020].

[31] Using Node Authorization. https://kubernetes.io/docs/reference/access-authn-authz/node, Jun 2020. [Online; accessed 4. Jun. 2020].

[32] Webhook Mode. https://kubernetes.io/docs/reference/access-authn-authz/webhook, Jun 2020. [Online; accessed 4. Jun. 2020].

[33] Wget. https://www.gnu.org/software/wget, Jul 2020. [Online; accessed 1. Jul. 2020].

[34] What is loopback interface in a Cisco Router. https://www.omnisecu.com/cisco-certified-network-associate-ccna/what-is-loopback-interface-in-a-router.php, Jul 2020. [Online; accessed 27. Jul. 2020].

[35] Ahmed, M. The Sidecar Pattern, May 2020. [Online; accessed 25. May 2020].

[36] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[37] Bischoff, M. Design and implementation of a framework for validating kubernetes policies through automatic test generation.

[38] Bosnjak, L., Sres, J., and Brumen, B. Brute-force and dictionary attack on hashed real-world passwords. pp. 1161–1166.

[39] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 70–93.

[40] Claassen, J., Koning, R., and Grosso, P. Linux containers networking: Performance and scalability of kernel modules. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium* (2016), IEEE, pp. 713–717.

[41] coreos. Flannel. https://github.com/coreos/flannel/blob/master/Documentation/backends.md, May 2020. [Online; accessed 8. May 2020].

[42] Dong, Y., Yang, X., Li, J., Liao, G., Tian, K., and Guan, H. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing 72*, 11 (2012), 1471–1480.

[43] Ferraiolo, D., Cugini, J., and Kuhn, D. R. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th annual computer security application conference* (1995), pp. 241–48.

[44] Frederickson, G. N., and Janardan, R. Designing networks with compact routing tables. *Algorithmica 3*, 1-4 (1988), 171–190.

[45] Hallyn, S. E., and Morgan, A. G. Linux capabilities: Making them work.

[46] Hashimoto, M. *Vagrant: up and running: create and manage virtualized development environments*. O'Reilly Media, Inc., 2013.

[47] Hightower, K., Burns, B., and Beda, J. *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, Inc., 2017.

[48] Hoffman, D., Prabhakar, D., and Strooper, P. Testing iptables. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research* (2003), IBM Press, pp. 80–91.

[49] Housley, R., Ford, W., Polk, W., and Solo, D. Internet x. 509 public key infrastructure certificate and CRL profile. Tech. rep., RFC 2459, January, 1999.

[50] Ibm. perf-sidecar-injector. `https://github.com/IBM/perf-sidecar-injector`, Jul 2020. [Online; accessed 3. Jul. 2020].

[51] intel. multus-cni. `https://github.com/intel/multus-cni`, Jul 2020. [Online; accessed 7. Jul. 2020].

[52] Ioannidis, J., Duchamp, D., and Maguire Jr, G. Q. Ip-based protocols for mobile internetworking. *ACM SIGCOMM Computer Communication Review 21*, 4 (1991), 235–245.

[53] James, T. Y. Performance evaluation of linux bridge. In *Telecommunications System Management Conference* (2004).

[54] Krebs, R., Momm, C., and Kounev, S. Architectural concerns in multi-tenant saas applications. *Closer 12* (2012), 426–431.

[55] Li, P. Selecting and using virtualization solutions: our experiences with VMware and VirtualBox. *Journal of Computing Sciences in Colleges 25*, 3 (2010), 11–17.

[56] Linuxize. Linux Time Command. *Linuxize* (Mar 2019).

[57] Lowe, S. What is SR-IOV? - Scott's Weblog - The weblog of an IT pro focusing on cloud computing, Kubernetes, Linux, containers, and networking, Jan 2020. [Online; accessed 3. May 2020].

[58] Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., and Rana, O. F. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering 68* (2018), 286–297.

[59] Morabito, R., Kjällman, J., and Komu, M. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering* (2015), IEEE, pp. 386–393.

[60] Nedelcu, C. *Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever.* Packt Publishing Ltd, 2010.

[61] NISHANIL. Communication in a microservice architecture, May 2020. [Online; accessed 9. May 2020].

[62] RIFFENBURGH, R. Chapter 24 - sequential analysis and time series. In *Statistics in Medicine (Third Edition)*, R. Riffenburgh, Ed., third edition ed. Academic Press, San Diego, 2012, pp. 509 – 533.

[63] ROUSE, M. provisioning. *WhatIs.com* (Aug 2010).

[64] SANFILIPPO, S. hping3 (8)-linux man page. *Online: https://linux. die. net/man/8/hping3* (2005).

[65] SARKALE, V. V., RAD, P., AND LEE, W. Secure Cloud Container: Runtime Behavior Monitoring Using Most Privileged Container (MPC). In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)* (2017), IEEE, pp. 351–356.

[66] SCHULTZ, E. E. A framework for understanding and predicting insider attacks. *Computers & Security 21*, 6 (2002), 526–531.

[67] SUAREZ, A. J., WINDSOR, S. K., HAYRAPETYAN, N., GERDESMEIER, D. R., AND PRAKASH, P. K. Software container registry service, Apr. 16 2019. US Patent 10,261,782.

[68] THÖNES, J. Microservices. *IEEE Software 32*, 1 (2015), 116–116.

[69] TIRUMALA, A. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/* (1999).

[70] TSYRKLEVICH, E., AND YEE, B. *Dynamic detection and prevention of race conditions in file accesses.* PhD thesis, University of California, San Diego, 2003.

[71] WELTE, H. The netfilter framework in linux 2.4. In *Proceedings of Linux Kongress* (2000).

[72] WITTE, R., AND WITTE, J. *Statistics.* Wiley, 2013.

[73] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013), pp. 233–240.

[74] Xu, C., Rajamani, K., and Felter, W. Nbwguard: Realizing network QoS for Kubernetes. In *Proceedings of the 19th International Middleware Conference Industry* (2018), pp. 32–38.

[75] Yuan, E., and Tong, J. Attributed based access control (ABAC) for Web services. In *IEEE International Conference on Web Services (ICWS'05)* (2005), p. 569.