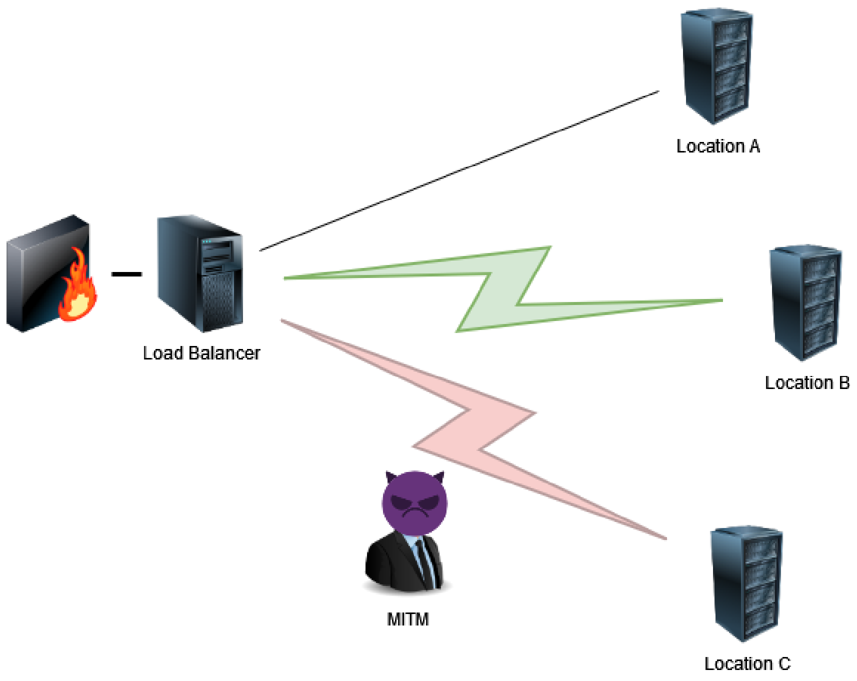

# Design, Implementation and Evaluation of a Moving Target Defense in Distributed Systems

An Open-Source Moving Target Defense System using Kubernetes Clusters

Master's thesis in Computer Systems and Networks

PHILIP TIBOM
MAX BUCK

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

# Design, Implementation and Evaluation of a Moving Target Defense in Distributed Systems

An Open-Source Moving Target Defense System using Kubernetes Clusters

PHILIP TIBOM
MAX BUCK

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Design, Implementation and Evaluation of a Moving Target Defense in Distributed Systems
An Open-Source Moving Target Defense System using Kubernetes Clusters
PHILIP TIBOM and MAX BUCK

Supervisor: Ahmed Ali-Eldin Hassan, Computer Science and Engineering
Examiner: Vincenzo Massimiliano Gulisano, Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A simple illustration of a moving target defense system. The red and green figures symbolizes the connection being rerouted from location C to B, while the connection to Location A is dormant.

Typeset in LaTeX
Gothenburg, Sweden 2022

Design, Implementation and Evaluation of a Moving Target Defense in Distributed Systems
An Open-Source Moving Target Defense System using Kubernetes Clusters
PHILIP TIBOM and MAX BUCK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

## Abstract

Cloud computing has recently become increasingly popular for server hosting. Additionally, a new model of cloud computing has emerged where cloud resources are placed at the edge of the network closer to the user. Both cloud and edge systems share many common security concerns, however, edge systems may suffer an increased risk of physical tampering and destruction. One way to harden the security in both cloud and edge systems is to use a technique called Moving Target Defense. The technique can be likened to the idea of frequency hopping in secure communication systems. Moving Target Defense is not yet widely adopted by industry and the current research in the area is very limited. Additionally, to our knowledge, there are no open-source implementations that can be easily replicated. The Moving Target Defense proposed in this thesis is an open-source implementation and can move a critical application between virtual and physical nodes in order to avoid and confuse adversaries. In addition to the implementation, we performed security, availability, and performance tests on the system. The results show that our system is able to successfully thwart some types of attacks while not significantly impacting availability and performance.

Keywords: moving target defense, distributed systems, kubernetes, cluster, cloud

# Acknowledgements

First and foremost, we would like to extend our gratitude to our supervisor Ahmed Ali-Eldin Hassan, for guiding and supporting us throughout the project. We would also like to extend our gratitude to our examiner Vincenzo Massimiliano Gulisano for taking us on. SNIC - Swedish National Infrastructure for Computing, for providing us with the cloud system that we used to conduct our experiments. AI Sweden (ai.se), for providing us with a workplace, seminars and guidance.

Philip Tibom and Max Buck, Gothenburg, August 2022

# List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

| | |
|---|---|
| IoT | Internet of Things |
| MITM | Man-In-The-Middle |
| MTD | Moving Target Defense |
| MVC | Model-View-Controller |

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

The cloud computing industry has grown exponentially during the last decade [2] and is estimated to be valued at hundreds of billions of US dollars, with growth predicted even further in the next few years [3], [4], [5]. There are large companies such as Amazon, Microsoft, Google, and thousands of other companies as well as billions of users relying on cloud technology to run their IT services [6]. Some of the greatest advantages of cloud technology are the operational abstraction layers that enable cost-effective and automatic scaling of IT services and features to achieve high availability. Edge computing is also an increasingly popular technology [7] where operational efficiency can be greatly improved by moving computation closer to the data collection mechanisms such as in IoT.

However, as with every computer system, cloud and edge computers are at risk from various digital threats and intrusions. Additionally, edge computers are at higher risk from physical attacks and tampering due to there unprotected positional nature. Traditionally, IT services are hardened by keeping software up-to-date, configuration of access permissions, firewall rules, and intrusion detection systems [8]. These defense mechanisms usually involve fixing bugs and reducing vulnerabilities, as well as using signature-based detection. However, these methods are for the most part only effective against known threats and attack vectors. For example, if an exposed service contains an unknown security vulnerability, then the defense mechanisms are likely to fail until the issue has been manually addressed. We are therefore in a position today where security is under constant scrutiny and it will always be one step behind an adversary.

Approaches to further harden the security of systems especially against unknown threats have been suggested before. One of these promising approaches is the so-called *Moving Target Defense*. In simple terms, a moving target should be more difficult to hit than a stationary target and this is true in most situations where an attack and defense are involved.

For example, a similar technology known as frequency hopping is used in wireless communication systems to reduce the chance of eavesdropping [9]. The idea is that the transmitter and receiver know in advance which frequency will be used at what time and an attacker would have to constantly monitor all frequencies to guarantee successful interception. Similarly, a moving target defense can be used in computer systems to make it more difficult for an attacker to successfully mount an attack. The attacker would first need to locate the position of the target, and then they would need to have enough time to prepare their attack before the target moves. For example, a computer system could use moving parts such as continuous switching of IP addresses, alternating between underlying software, and moving critical applications between locations. In theory, an application that is being moved between edge nodes located at different positions should be more resilient against physical tampering and destruction. Furthermore, a service that is continuously changing its underlying software stack might confuse fingerprinting techniques and exploitation tools and thus, potentially defend against some unknown threats.

Cloud environments do not only pose similar risks as traditional computer systems, but they also enable benefits such as the possibility of automatically scaling IT services up and down horizontally [10], which could also be used to achieve a moving target defense. Scaling and availability are often accomplished by using orchestration and container tools such as Kubernetes [11] and Docker [12]. Container tools allow applications to be containerized and to be easily deployed in container run-time environments. Orchestration tools manage multiple such run-time environments in a cluster and provide tools for high availability and scaling. This thesis explores the possibility of leveraging these orchestration and container tools in cloud and edge systems to achieve a moving target defense.

The idea of this thesis is to provide an open-source design and implementation of a moving target defense using cloud and edge systems in combination with Kubernetes. In addition, we will provide an analysis of the performance, availability, and security of the moving target defense system to investigate if the chosen implementation is viable in the real world. One key novel idea in our work is the idea of alternating equivalent software services at runtime, e.g., alternating Nginx, and Apache web servers to alternate potential vulnerabilities in order to prevent exploitation, and to confuse fingerprinting techniques. Additionally, the implementation involves moving the target service between multiple physical nodes to defend against potentially compromised nodes as well as physical tampering. The analysis involves measuring any changes in latency, and downtime as well as comparison of real-world threats against the moving target defense compared to the same setup without a moving target defense.

## 1.1 Problem

Typically, defense systems include rule-based solutions such as firewalls, intrusion detection systems, and anti-virus. It may also include storage encryption or communication encryption to protect against man-in-the-middle attacks and compromised hardware. These security measures are however not always enough, encryptions can sometimes be broken, encryption keys may be compromised [13], and rule-based security can be circumvented [14]. One way to improve the security against unknown threats is to add a moving target defense as an extra layer of defense. Moving target defense is a loose term that can be interpreted and implemented in many various ways and each implementation will defend against different types of threats.

One way to implement a moving target defense is to interpret the term quite literally, by moving the target application physically between different nodes. The approach in this thesis is to use a cluster of cloud nodes to move a critical application between the different nodes. Moving an application between nodes can defend against physical attacks against the hardware and man-in-the-middle attacks. Another feature is to alternate exposed software services and in turn, also change the vulnerabilities and fingerprints. For example, while a fingerprinting tool is running, the target software may be swapped to other versions or software with similar functionality, to reveal different fingerprints and thus, confuse the attacker. Alternatively, if there is physical tampering such as side-channel attacks, the target application may move to a different node avoiding the attack altogether, or in the case of destruction, the application may continue to run on a different node.

## 1.2 Purpose and Goals

The purpose of this thesis is to conduct an experiment using the concept of the moving target defense presented in section 1.1 and to evaluate whether the concept can work in reality. Furthermore, the purpose is to contribute to industry and academia with a moving target defense project that can be easily replicated, adapted, and built upon to help further increase security in services running in cloud or edge clusters. The goal is to design a moving target defense prototype and evaluate the viability of the idea and the implementation, in terms of performance and availability, and evaluate the potential security improvements. The goal is also to show any limitations that the system may have. To accomplish this, there will be two parts of this project, the thesis and the prototype.

1. The thesis focuses on the theoretical parts of the prototype such as the technical background, implementation design, experimentation, analysis, and discussion. It aims to provide the necessary details to fully comprehend the implementation and its underlying infrastructure.

2. The prototype is an open-source moving target defense system in cloud and edge clusters and should enable others to build upon it. It is also meant

to inspire ideas for future projects. The prototype aims to provide a public git repository with the necessary documentation and code.

## 1.3 Scope

The moving target defense will be tested on a small-scale distributed system due to time limitations and operational costs. This project aims to make a prototype and not a production-ready product. It would be logical to conclude that if the defense does not work for smaller systems, it would not work for larger ones either.

One of the potential benefits of an MTD system that we would like to test is that it can protect against new threats with unknown vulnerabilities, so-called zero-day threats. The problem with this is the task of identifying critical security bugs and then creating threats to exploit them would be too large a task for this project. To solve this, the penetration testing will involve known, already patched vulnerabilities against older and knowingly vulnerable software for the purpose of demonstration and testing. If successful, it is likely that it can defend against similar new threats.

The goal of the report and the moving target defense system is not to eliminate all threats but rather to strengthen security and to highlight the moving target defense system's strengths and weaknesses. The experiments will explore whether the moving target defense will improve defense against certain types of attacks. It will not conclude whether it fails or succeeds against every type of attack.

## 1.4 Research Questions

Based on the goals in section 1.1, we have outlined two research questions below that we will aim to answer throughout this thesis.

RQ1. Is the chosen type of MTD implementation viable in a real-world scenario, in terms of availability and performance?

RQ2. Does the MTD provide any security benefits, and if so, when and how much?

## 1.5 Our Contributions

We show that it is possible to create a moving target defense using Kubernetes with the help of the Kubernetes API and open-source tools. Our tests show that the system was not noticeably degraded in terms of availability and performance and that it could potentially be applied in a real-world scenario. Our security tests show that the system can defend against certain types of vulnerabilities, such as timing-sensitive attacks as well as MITM and physical attacks.

## 1.6 Thesis Outline

The thesis begins by introducing the topic in this chapter. Chapter 2 aims to contribute with the necessary technical background information that will help the reader to understand the underlying concepts and technology used in the project and experiments. Chapter 3 goes into related work where other moving target defense systems are discussed. Chapter 4 shows the design in theory. Chapter 5 shows the implementation and the necessary details in order to recreate the project from scratch. Chapter 6 contains experiments, evaluations and analysis of the implementation. Chapter 7 discusses the viability of the implementation based on the results from the experiments. Lastly, chapter 8 concludes this thesis.

# 2

# Technical Background

This chapter explains the relevant technical background to understand the design and implementation of the moving target defense system. We will start with an overview of virtualization and containerization since these concepts are necessary to understand the system. Then we will cover orchestration and load balancing which are key components of the implementation. Finally, we go over some relevant security concepts that we aim to address with the system.

## 2.1 Virtual Machines in the Cloud

A cloud consists of many physical servers, where each physical server contains a hypervisor [15]. The hypervisor is used to manage the available resources and to assign them to individual virtualized environments so that multiple virtual environments can run on a single physical server. One of the most elementary services in a cloud computing environment provides the ability to create virtual machines on-demand within the cloud, often referred to by cloud providers as compute instances [16]. These virtual machines are allocated a fixed amount of resources such as CPU, memory, storage, and bandwidth [17]. For each virtual machine, an operating system must be installed and configured. Additionally, every dependency and software that is required to run a specific application or service needs to be installed on each virtual machine. A simplified representation of a cloud architecture is shown in Figure 2.1 below.

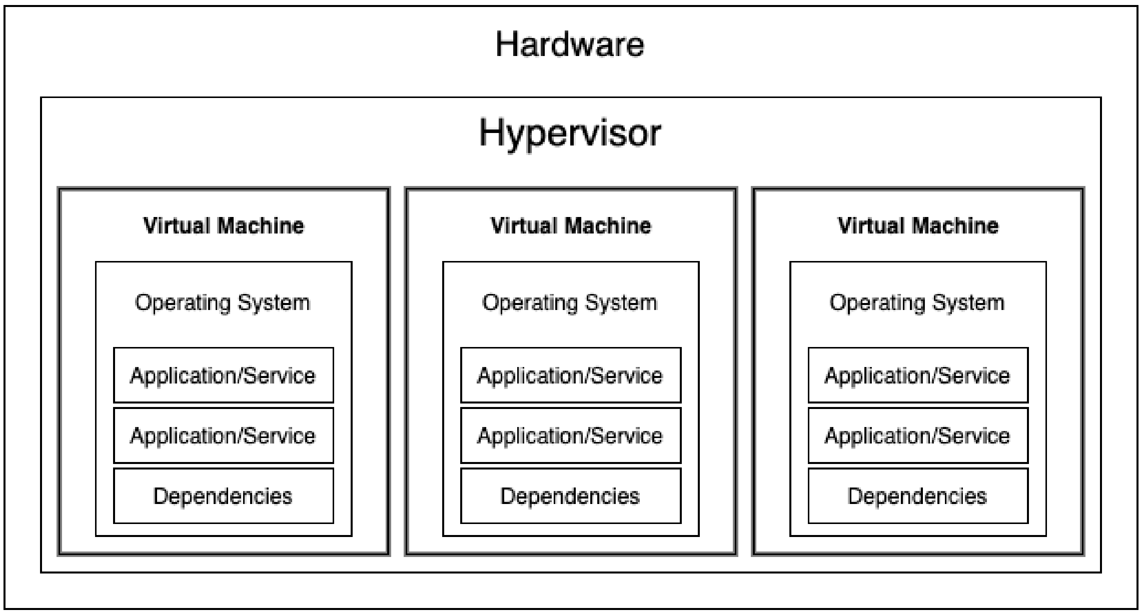

**Figure 2.1:** This is a simplified representation of a cloud architecture. A physical server contains a hypervisor, which in turn manages several virtual machines. Each virtual machine has its own operating system and manages its own software.

Furthermore, since each virtual machine must be installed, configured, and secured individually, it makes management very time-consuming and expensive. Especially if we need to scale the service horizontally, and run hundreds or thousands of instances. Though, it is possible to create an image of an instance and replicate it to other instances and thus somewhat reduce the manual work. However, such an image quickly becomes very large as it includes the entire operating system and all the installed software. So, while doable in some scenarios, it is very inefficient to replicate such an image to many nodes over a network, especially if the nodes are located on the edge. Alternatively, it is possible to develop custom procedures to automatically update and deploy specific software to many nodes.

Another part of managing a cluster in the cloud involves monitoring active instances and to ensure that they are running properly. For example, there must be a routine for handling instances that have encountered any sort of failure which is usually solved by restarting the failed instance. If there is an issue with the hypervisor, then the instance would need to be relocated to a working host. Additionally, a single instance may run multiple services which can all fail individually and thus require special monitoring for each service. As seen in the mentioned examples, the monitoring adds yet another layer of complexity when running many instances. This leads us to a technique called containerization, where many of these issues have already been improved. Containers will be explained in section 2.2.

## 2.2 Container Technologies

A container is essentially an application that has been encapsulated together with instructions on how the application should be executed [18]. For example, it may include installation instructions for required dependencies. This allows the application to be easily deployed to any compatible container runtime. It also reduces the required operational knowledge for a specific application significantly and it becomes easier to manage in the case of a failure.

To execute a container, a specific container runtime is necessary. There are many different container runtime technologies such as Containerd [19], Runc, and CRI-O [20] of which all follow a standard called OCI (Open Container Initiative) [21]. This standard makes the container runtimes compatible with various different orchestrators. Being OCI compatible also means they follow the same container formats, meaning the different OCI compatible container runtimes can run any OCI compatible container. There are also more feature-rich runtime environments such as Docker [12], which uses Containerd underneath to run its containers. More on why Docker is useful will be mentioned later.

In contrast to virtual machines which contain an entire operating system, a container includes only a specific application and its necessary dependencies [18]. The container runtime environment typically runs on a Linux-based operating system. The operating system can run directly on the hardware or it can run inside a virtual machine. However, a single container runtime environment can run multiple containers which are all isolated from each other, but at the same time, they can share the kernels. A simplified architecture stack is shown in Figure 2.2 below.

Containers and virtual machines offer different levels of security. Virtual machines provide better security because they are isolated from the underlying operating system. This means that any malicious activity or viruses on the host machine should not affect the virtual machines. In contrast, containers are not as isolated as virtual machines and can be affected by the underlying operating system. For example, if the host machine is compromised, the containers can be as well. Additionally, containers share the kernel with the host machine and with other containers. This means that a security vulnerability in the kernel can affect all containers.

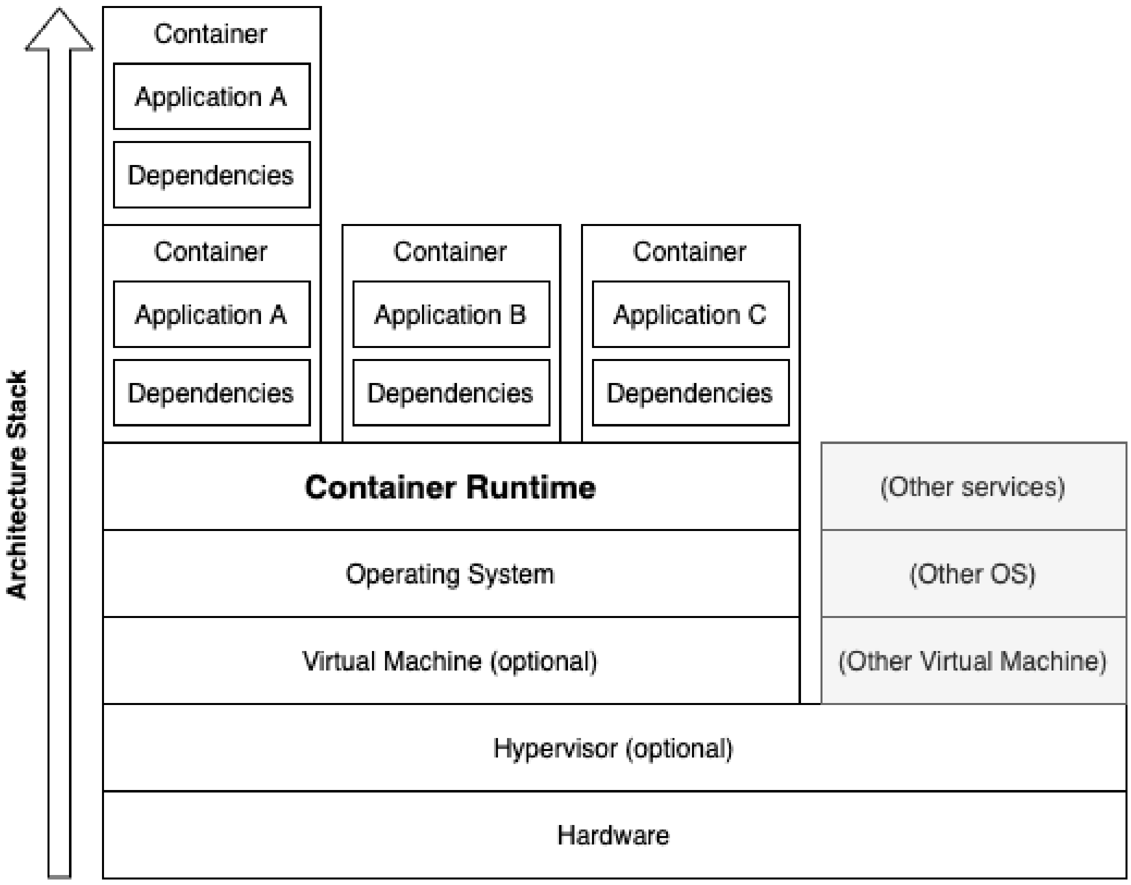

**Figure 2.2:** It is possible to run multiple instances of the same container image. As seen in the figure, there are two container instances of application *A*. It is also possible to run many different containers at the same time. Additionally, the containers can be configured so that in the case of a failure, the failed instance is automatically deleted, and a new instance of the same container is created.

### Docker

Docker [12] is a container runtime environment and platform that uses Containerd underneath. Docker comes with additional tools and functionality such as container repositories, monitoring, and a CLI (command line interface). A container repository is a source from which the Docker runtime can download copies of specific containers. For example, the official Docker repository contains popular containerized applications such as Nginx, Apache, MySQL, and many more. It is also possible for an organization to host its own repository with private container images. Using repositories is the typical method of deploying new containers and updates to existing instances. Once a container has been pulled to the runtime environment, it remains in the cache and can be used to start instances until it is removed. Thanks to the features of the containerized architecture, it becomes easy to deploy one or more applications to a single node [18]. It makes failure handling easier and decouples applications and dependencies. The infrastructure becomes modular and gets easier to scale up and down as required. However, scaling and replication across multiple nodes remain tedious as one would need to manually deploy the corresponding containers to each and every node. This is where orchestrators come into the picture, and they will be described in section 2.3.

## 2.3 Container Orchestration

An orchestrator is a tool to manage multiple container runtime environments [22]. It is also used for scheduling and deployment of the containers mentioned in section 2.2. Having multiple container runtime environments forms a so-called cluster. The orchestrator will monitor the entire cluster and make sure that the applications are running as they should. Apart from monitoring and failure handling, it will also come with features such as rolling updates which means it will update an application on the cluster gradually to ensure no down time of the service.

There are a couple of well-known orchestrators in the industry today such as Kubernetes [11], Docker Swarm[23], Openshift[24] and Mesos[25]. Kubernetes was originally created by Google but is now maintained and developed the Cloud Native Computing Foundation which is part of the nonprofit Linux Foundation. This makes Kubernetes open-source and vendor neutral which has allowed it to be implemented in all the major cloud services from companies such as Amazon, Google, Microsoft, IBM and more. This makes Kubernetes the currently most popular orchestration tool.

### 2.3.1 Kubernetes Architecture

On an abstract level, Kubernetes contains mainly three parts, the orchestrator/-manager, the container runtime environments, and the containerized applications [11]. Together, these parts form a Kubernetes cluster. Each container runtime environment is called a worker node. The worker node can be either on a physical or on a virtual machine, and each node can execute multiple containerized applications. Kubernetes supports any OCI-compliant container runtime which means that the user is not locked in a specific vendor such as Docker for as long as the container is OCI-compliant, and the user does not necessarily need to know which the underlying container runtime is.

In Kubernetes, the containerized applications are wrapped in something called a pod. A pod is the smallest deployable unit in Kubernetes. A pod can contain multiple containers and thus provide an entire application stack in a single deployment, but the most common usage is to have one container per pod. A pod includes additional information compared to a container such as logical hosts, how it is connected to other pods on a virtual network on a specific node etc. A simple example is illustrated in Figure 2.3 below.

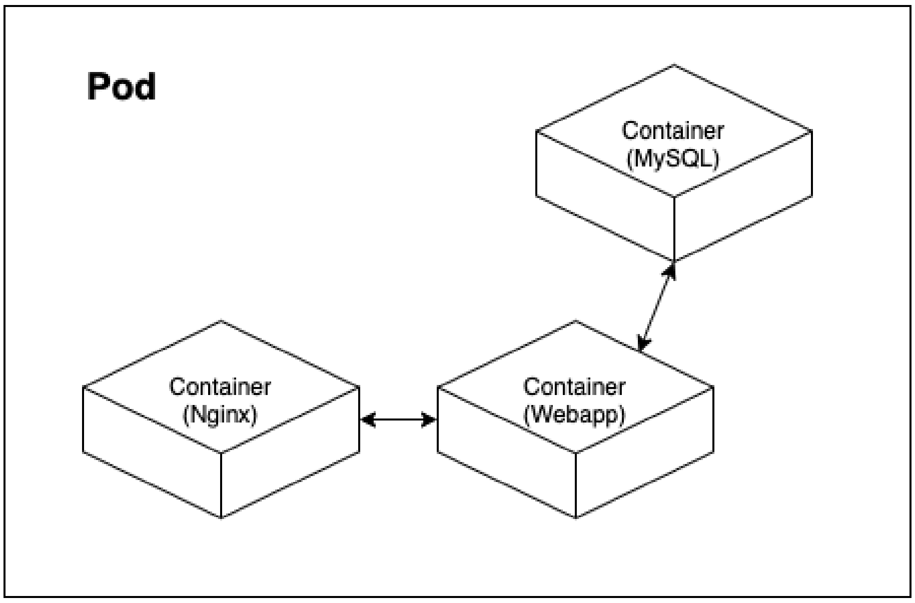

**Figure 2.3:** An example of a pod containing three different containers. Nginx serves as a web server and displays information given by the web application. The web application is connected to a MySQL database server. The pod can be replicated and scaled horizontally to run with multiple instances on the same worker node or distributed across different worker nodes

Orchestration and containerization technology is highly relevant to the moving target defense system presented in this thesis. This is because it allows applications to be deployed quickly with less configuration in comparison to virtual machines. For example, even a preconfigured virtual machine image will take significantly longer to boot up than a container or a pod, which will result in a moving target defense that will result in a larger overhead [18]. Furthermore, cluster management tools that an orchestrator provides, such as Kubernetes are already available.

### 2.3.2 Minikube – Local Kubernetes Cluster

For simple testing and learning the basics of Kubernetes, it is usually simplest to set up a cluster locally. Several tools have been developed to simplify the setup and reduce the time needed to get up and running. Minikube, maintained by the Kubernetes developers, is one of those tools [26]. With Minikube, a cluster can be set up locally on a computer. It supports multiple container runtimes such as Docker, Containerd and CRI-O and several ways to deploy the cluster. These include running the cluster directly on bare metal, virtually in a VM, or in a container.

When starting a Minikube cluster with the command `minikube start`, a number of specified nodes will be created for the user to populate with Kubernetes resources. One of the nodes is the master node which contains the control plane [11]. It consists of several components which are responsible for several different important tasks within the cluster. These include for example a scheduler and a key-value storage solution.

### 2.3.3 Kubernetes Clusters in the Cloud

Kubernetes clusters can also be set up in cloud environments such as Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, or DigitalOcean. These providers all offer managed Kubernetes services that take care of the underlying infrastructure [27]. When using a managed Kubernetes service, the user does not have to worry about the underlying infrastructure or setting up and maintaining the control plane. The cloud provider takes care of that automatically.

However, there are certain limitations to using managed Kubernetes services since the user does not have as much control over the cluster. For example, DigitalOcean does not currently support multi-region Kubernetes clusters. This means that if the user wants to set up a Kubernetes cluster in multiple regions, they would have to set up and manage the cluster themselves.

It is possible to set up and manage a cluster from scratch. This can be done on-premises or in the cloud, allowing specialized hardware and configurations, such as a cluster of edge devices. There are several tools available to help with the setup process, such as kubeadm and kops.

### 2.3.4 Kubeadm

Kubeadm is a command-line interface toolkit that helps get a Kubernetes cluster up and running [11]. It is the official way to provision a minimum viable Kubernetes cluster that conforms to best practices. It handles the heavy lifting of bringing up a Kubernetes cluster, from provisioning compute resources to configuring networking and storage. Kubeadm is also responsible for joining new nodes to the cluster.

In order to run Kubeadm and bootstrap a cluster, certain prerequisites must be fulfilled such as minimum available resources and software installation and configuration, which can be found on Kubernetes website [11]. Though, once all the prerequisites are met, mainly three commands are used to provision a minimum viable cluster, listed below.

**kubeadm init**
Used to create a control plane node. It does this by going through a series of tasks, mostly setting up the components which makes up the control-plane and generating a token which nodes uses to join the cluster.

**kubeadm token**
Used to create a token for a control plane. The token have an expiration-date, so new ones need to be generated if nodes want to join a cluster after a certain time.

**kubeadm join**
Used to create a Kubernetes node on the machine running the command and then joining it to the specified control plane.

## 2.4 Kubernetes Concepts

This section aims to provide a general overview of the conceptual aspects of Kubernetes that will be used throughout the thesis.

Kubernetes employs a number of abstractions that make up an important part of its API and are used to manage complex sets of containerized workloads. These abstractions include ReplicaSets and Deployments, which both provide a declarative way to manage groups of running workloads. Each of these two concepts will be described in subsection 2.4.1 and subsection 2.4.2, and how to declaratively configure such workloads will be described in subsection 2.4.3.

The API is organized around the concept of resources, which are collections of objects with similar characteristics. Each type of resource has its own set of endpoints for manipulating it. For example, the Deployment resource has an endpoint for creating new Deployments, and another for listing existing Deployments. Furthermore, how to access such endpoints will be described in subsection 2.4.4 with the help of the official tool called Kubectl, and later in subsection 2.4.5 on how to access the API using a custom-made client.

### 2.4.1 ReplicaSets

In Kubernetes, a ReplicaSet is a resource that ensures a specified number of pod replicas are always available [11]. It does this by creating and deleting pods as needed to maintain the desired replica count. ReplicaSets uses an algorithm to make sure all the pods are evenly distributed among the nodes with available resources. If a pod is deleted or crashes, the ReplicaSet will create a new pod to replace it. ReplicaSets are a critical part of orchestrating and managing pods in Kubernetes, ensuring that applications always have the resources they need to run effectively. This provides high availability for applications running in Kubernetes.

ReplicaSets are also often used to create scalable applications, as they allow to easily add or remove pods as needed to handle changing demand. For example, to have 10 instances of an application running at all times, a ReplicaSet with a replica count of 10 would have to be created. The number of replicas can also be scaled up and down on-demand. Pods that are part of the same ReplicaSet are usually identical, meaning they have the same labels and annotations. This allows the ReplicaSet to manage them as a group. When a new pod is created, it is automatically added to the ReplicaSet.

### 2.4.2 Deployments

A Deployment is a type of workload that can be created in a Kubernetes cluster [11]. Deployments provide a way to create and manage replicated applications, scaling them up or down depending on resource usage. Deployment is a higher-level concept that manages ReplicaSets (described in subsection 2.4.1).

Unlike ReplicaSets, updates to Deployments are managed declaratively, using Kubernetes' YAML configuration files. This allows specifying the number of replicas needed for an application and how those replicas should be deployed or upgraded. With a ReplicaSet, the replication controller must be manually updated every time there is a change in the application version or desired state.

Deployments also provide easier control over how the application is scaled and upgraded. For example, a deployment can have a rollout strategy specified when upgrading the application. It is also possible to roll back changes if they cause issues in the application, without having to manually delete or modify the replication controller. For example, during a rolling update, a Deployment manages two ReplicaSets ($A$ and $B$), where $A$ scales down one step as $B$ scales up one step. Whereas, a single ReplicaSet on its own cannot accomplish a rolling update.

Due to the high-level functionality and flexibility provided by Deployments, they are typically recommended for use instead of ReplicaSets. However, there are some cases where a ReplicaSet would be more appropriate – for example, if precise control over scaling is needed or a different deployment strategy needs to be used.

### 2.4.3 Kubernetes Config Files in YAML

YAML is a human-readable data serialization standard [28] that can be used to store data in a structured format. It is often used for configuration files, but can also be used to store data in a database or transmit data over a network.

Kubernetes supports both YAML and JSON for its configuration, though YAML is more common and is recommended in the best practices [29]. In Kubernetes, YAML is often used to specify which services should be provisioned as part of an application deployment and how those services should interact with each other. For example, a YAML file may contain instructions for launching a web service that contains several microservices behind it and specifying the ports on which those services should be exposed.

YAML is flexible and allows for a wide range of configuration options to be specified. This makes it possible to customize Kubernetes deployments in different ways, depending on the needs of the organization or application.

Finally, YAML is widely supported by many different tools and programming languages. This means that you can use a wide variety of tools to manage and modify Kubernetes configurations, without having to rely on proprietary formats that are only supported by certain vendors.

Some important YAML configuration options for Kubernetes include:

- The services that should be provisioned for the application. This may include container images and the ports on which they should be exposed.
- Storage volumes and any associated requirements, such as whether data in those volumes needs to be encrypted or replicated across multiple nodes.
- Configuration settings for network policies, such as which traffic should be allowed into the cluster and which services should be able to communicate with each other.
- The desired state of the Kubernetes cluster, such as how many nodes should be running and what version of Kubernetes should be deployed.
- Settings for cluster autoscaling, including minimum and maximum capacities for the nodes in the cluster.
- Settings for high availability, such as whether a replicated data volume should be used to store persistent data.

An example of a YAML deployment file is specified below:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

**Listing 2.1:** Deployment Example in YAML [1]

The example YAML file above will create a Deployment with three Pods, each running the nginx container. The Deployment will expose port 80 on the Pods, allowing users to access the application via a web browser. It will also automatically update or replace Pods if they become unavailable or have performance issues, ensuring continuous availability and high performance for the application.

- `Kind` specifies the type, such as Deployment or ReplicaSet
- `Metadata` specifies name and labels for the Deployment.
- `Spec` specifies what the Deployment should accomplish. It includes the number of replicas, selector, and template. Template is used to create Pods. It includes metadata and spec for the containers in the Pods.
- `Containers` specify the name, image, and port for each container in the Pod.

### 2.4.4 Kubectl

Kubectl is a command-line interface (CLI) that provides access to the Kubernetes API [11]. It is used to manage Kubernetes clusters, deploy applications, and perform other tasks. It can be used to create, update, delete, and view resources in a Kubernetes cluster, such as pods, services, and replication controllers. Some of the most essential commands to understand are described below.

**kubectl create**
Used to create a resource, like a deployment or replicaset, from either a YAML file or directly from the command line. If the command is run with the same input a second time it will not work, since there already exist a resource with that name.

**kubectl apply**
Used to either create a resource from a YAML file or stdin, or apply the changes if a resource with the same name already exists in the namespace. For this reason it is recommended to use it over the `kubectl create` command.

**kubectl get**
Used to get useful information about the resources on the cluster. It can target either a specific kind of resources like nodes or deployments, or an overview of the whole cluster. Often used with the `-o wide` flag to get additional information.

**kubectl label**
Used to create labels on resources. Updating already existing labels is done via the `-overwrite=true flag`.

**kubectl delete**
Used to delete resources from the cluster based on file name or their name together with either resource type or labels.

**kubectl rollout restart**

Used to rollout updates to a resource, like a deployment. The command makes sure that the cluster maintains availability by never taking down a amount of pods below a user specified threshold. For example, if the deployment has 4 pods and threshold of 75%, then it first takes down a pod and the creates a pod. It continues doing this until all the pods has been replaced. There is a similar threshold of the maximum number of allowed pods, which could be reached if the deletion process is slow. The restart command in combination with rollout will simply restart the resource while still having it available.

### 2.4.5 Kubernetes API

The Kubernetes API is a way to talk to the Kubernetes cluster [30]. It can be used to do things like create new pods and services or get information about what's going on in the cluster. The API is divided into several parts, each with a different purpose. The API server is the component that actually handles the requests from clients. It then forwards the requests to the appropriate controller. The controllers are responsible for making sure that the cluster is in the desired state. There are many different types of controllers, each with a specific purpose. For example, there is a ReplicationController which is responsible for making sure that there is the correct number of replicas of a given pod. The API is however not intended for use by end users. It is meant for use by tools and scripts that need to automate the management of a Kubernetes cluster.

The CLI tool kubectl is built on top of the Kubernetes API. It can be used to do all the things that can be done with the API, and more. It is also possible to write programs that use the Kubernetes API directly. For example, there are both official and third-party libraries available that implement the Kubernetes API in different programming languages. However, many such commands that kubectl performs such as a rollout restart is not part of the API and must be implemented in the program that utilizes the API if such a feature is desired. There are however libraries that mimics some of the kubectl commands, but not all of the commands as can be seen in the official Kubernetes Java Client.

## 2.5 Network Load Balancing

A network load balancer is a device that helps distribute the workload of traffic across different computers or servers [31]. This is helpful because it can make sure that no one computer is too overloaded and that all of the computers are doing an equal amount of work. They can also detect if a server is down and stop sending traffic to it.

Nginx and HAProxy are two examples of programs that can act as load balancers among other things. In Kubernetes, there are two types of load balancing: internal and external [11]. Internal load balancing is when the load balancer is within the cluster and only distributes traffic to services within the cluster. External load balancing is when the load balancer is outside of the cluster and can distribute traffic to services both inside and outside of the cluster.

Kubernetes services can be exposed in a number of ways. One way is to use a service type called “LoadBalancer”. When using a “LoadBalancer” within a cloud system, Kubernetes will automatically create and configure an external load balancer for the service. This is useful because it means that there is no need to manually set up and configure a load balancer. Another way to expose a service is to use NodePort. NodePort exposes the service on each node in the cluster at a specific port. Any traffic that is sent to this port on any of the nodes will be forwarded to the service.

Kubernetes also has the ability to do load balancing at the application layer. This is done with Ingress resources. An Ingress resource defines a set of rules that can be used to route traffic to different services within the cluster. This is useful because it allows for more control over how traffic is routed and it can make it easier to change the routing in the future if needed.

## 2.6 Security

There are many concepts within cyber security, this section aims to provide some general knowledge about the concepts used within this thesis. Some common defense systems will be introduced to provide an understanding of the differences between them and the Moving Target Defense that is experimented on in this thesis and how they could be used together. This section also provides some basic information about fingerprinting techniques, vulnerability scanning, exploitation, and how security systems can be evaluated.

### 2.6.1 Defense Systems

Static Code Analysis and Dynamic Code Analysis are two methodologies for code reviews [32]. Static Code Analysis is performed without running the code and it can be done using tools that analyze the code for potential problems. This can be helpful because it can find problems that would be difficult to find otherwise. However, it can also produce false positives, which are when the tool reports a problem but there is no actual problem. Dynamic Code Analysis is performed by running the code and observing its behavior. This can be helpful because it can find problems that are not easily found by static analysis. However, it can also be difficult to set up and it can be slow.

A Firewall is a system that controls traffic between networks [33]. Firewalls can be hardware, software, or a combination of both. They are used to protect networks from external threats. A firewall works by inspecting traffic and comparing it to a set of rules. If the traffic matches a rule, then the firewall will allow or deny it. There are many different types of firewalls. Some common types are stateful firewalls, application-level firewalls, and network-level firewalls. Stateful firewalls keep track of the state of connections and only allow traffic that is part of an existing connection [34]. Application-level firewalls such as web application firewalls (WAF) inspect traffic at the application layer [35] and can make decisions based on the application that is being used. Network-level firewalls inspect traffic at the network layer and can make decisions based on the source and destination of the traffic.

A Network Intrusion Detection System (NIDS) is a system that monitors network traffic and looks for signs of intrusion [36]. NIDS can be used to detect attacks as they are happening or after they have happened. They work by inspecting traffic and comparing it to a set of rules. If the traffic matches a rule, then the NIDS will generate an alert. NIDS can be either host-based or network-based. Host-based NIDS are installed on individual hosts and can only monitor traffic to and from that host. Network-based NIDS are placed in the network and can monitor all traffic that passes through them.

A Network Intrusion Prevention System (NIPS) is a system that monitors network traffic and looks for signs of intrusion [36], similar to a NIDS. NIPS can be used to detect attacks as they are happening and block them before they can do any damage, in contrast to a NIDS which only generates an alert. NIPS work by inspecting traffic and comparing it to a set of rules. In contrast to a network firewall, NIPS and NIDS perform packet inspections in order to detect malicious payloads or activities.

Note that all of the defense systems mentioned in this section complement each other in various ways and could thus be used in conjunction with each other. A moving target defense is yet another defense system and does not replace or prevent any of the other systems from being used. Usually within the perspective of security, the goal is to harden a system as much as possible within a reasonable scope. Which means that a simple system such as a network firewall is almost always used since the overhead of such a system can be very light.

### 2.6.2 Fingerprinting and Vulnerability Scanning

Fingerprinting is the process of identifying the operating system, software, and hardware of a device [37]. Fingerprinting can be used for many different purposes. For example, it can be used to find out if a device is running an outdated or vulnerable version of the software. It can also be used to target attacks against specific devices. There are many different ways to fingerprint a device. Some common methods are examining the headers of network traffic, looking for known vulnerabilities, and running system-specific commands. Fingerprinting is therefore often one of the first steps that an attacker will take.

A vulnerability is a flaw in a system that may be exploited. to gain access to the system or its data. Vulnerabilities can be found in software, firmware, and hardware. They can be found in operating systems, web applications, mobile apps, and devices. Vulnerabilities can be discovered through manual testing or automated scanning [38]. Many vulnerabilities are reported to the vendor and some are made public. Some common types of vulnerabilities are buffer overflows, cross-site scripting, and SQL injection.

Vulnerability scanning is an automated process of identifying vulnerabilities in a system [39]. Vulnerability scanners can be used to scan for known vulnerabilities and verify that patches have been applied. Such scanners can also be used by adversaries or malicious software to find vulnerabilities that can be exploited. There are different types of vulnerability scanners. Some common types are web application scanners, network scanners, and database scanners.

Later in this thesis, a scanner called OpenVAS Greenbone Security Assistant [40] will be used to scan the moving target defense system for fingerprints and vulnerabilities. OpenVAS Greenbone Security Assistant is a free and open-source vulnerability management system. It is used to scan for vulnerabilities, track remediation, and generate reports.

Another tool that will be used is Nmap. Nmap is a free and open-source network exploration tool and security scanner [41]. Nmap can be used to scan for hosts and services on a network. It can be used to identify hosts that are up, down, or have specific ports open. Nmap can also be used to fingerprint devices to find out what operating system and software they are running.

### 2.6.3 Exploitation, CVE - Common Vulnerabilities and Exposures

A CVE, or Common Vulnerabilities and Exposures [42], is a rule that defines a class of vulnerabilities. A CVE can be used to identify which devices are affected by a particular vulnerability. CVEs are assigned by the MITRE Corporation. The CVE website includes a searchable database of CVEs.

To exploit a system means to take advantage of a flaw in the system to gain access to it or its data. There are many different types of exploits. Some common types are buffer overflows, cross-site scripting, and SQL injection. To find out if a system is vulnerable to an exploit, one can check the CVE database to see if there are any CVEs that apply to the system.

In this thesis, the CVE-2014-0160 [43], also known as The Heartbleed Bug [44], will be used as an example. The Heartbleed Bug is a security flaw in the OpenSSL cryptography library. The flaw allows an attacker to read the memory of a system that is using a vulnerable version of OpenSSL. This can be used to obtain sensitive information such as passwords and private keys. The Heartbleed Bug was first made public in April 2014. The exploitation of this bug works by sending a specially crafted heartbeat request to a system that is using a vulnerable version of OpenSSL. The response from the system will include sensitive information from the system's memory. The response is at most 64 kilobytes in size and contains random data from the memory. This means that an attacker can send multiple requests and get different pieces of data from the memory each time, in hope of finding the desired data.

A free open source penetration testing framework called Metasploit [45] can be used to make penetration testing easier. Metasploit comes with a large number of exploits and payloads. It can be used to launch exploits amd payloads against vulnerable systems. In this thesis, the Metasploit module named “auxiliary/scanner/ssl/openssl_heartbleed” will be used to exploit the CVE-2014-0160, The Heartbleed Bug, as later described in chapter 6.

### 2.6.4 Backdoors

A backdoor is a way to bypass security measures and gain access to a system. Backdoors can be placed on systems by manufacturers, developers, or attackers. They can be used to gain access to systems for legitimate purposes or for malicious purposes. Typically, an adversary may upload a backdoor (a piece of software) to a system in order to gain access to it at a later time.

Backdoors can be prevented by using security measures such as access control lists, intrusion detection systems, anti-virus, and so on [46]. Access control lists can be used to restrict access to systems and data. Intrusion detection systems can be used to detect and respond to unauthorized activity and anti-virus may scan the file system or memory for detected malicious software such as a backdoor. However, if the backdoor is unknown and slips through these defenses, it can be very difficult to find and remove since the host may not even know it exists [46].

# 3

# Related Work

There are several research papers on the subject of Moving Target Defense. For example, Roy *et al.* [47] proposes Moving Target Defense as a technique to combat adversarial machine learning. The authors propose to alternate between different algorithms in a given software and are introducing randomness into its parameters. Zeitz *et al.* [48] goes into the design of a Micro-Moving Target IPv6 Defense for Internet of Things (IoT). The authors propose protocols, modes of operations and a possible simulation for Cooja (a network simulator for Contiki OS [49]).

However, our project takes a different approach than the two aforementioned papers [47] and [48]. Firstly, our project involves building a Moving Target Defense system that moves the critical application between different nodes. Secondly, our project aims to contribute with a proof of concept and the necessary details so that researchers and developers in the field can replicate and adapt the project to a real-world situation.

A third approach by Ahmed and Bhargava [50], proposes a framework for Moving Target Defense for distributed systems. The project seems to have been implemented and tested in practice by the authors, but the demo is not publicly available and the paper is not focused on the security aspects and benefits.

Al-Shaer *et al.* [51] and Caroll *et al.* [52], have investigated methods of Moving Target Defense that involve randomizing or shuffling the network addresses of hosts. Their methods are different from our project but could in practice achieve a similar result. They have studied the idea with simulations and tested it with small-scale prototypes with a feasible outcome.

Another paper by Tian *et al.* [53], which was influenced by Al-Shaer *et al.* [51], looked into the possibility of using MTD to thwart stuxnet-like attacks. These attacks work by targeting the control signals and manipulating the measurements on the system to mask the attack. Their results concluded that MTD could work as potential security measure, in large part because reconnaissance of the system is important to the attack which MTD is designed to handle.

Kenney A. Torkura *et al.* [54] wrote a paper that explains that Microservice Architectures (MSA) tend to use common images from public repositories such Docker Hub, which makes the services more vulnerable to attack since the same type of service is running on many machines. They propose Moving Target Defense mechanisms to overcome the security implications of homogeneous microservices by using different types of scoring metrics and by automatically altering the container images to accomplish diversification. These mechanisms are possibly something that could be combined with the moving target defense explored in our thesis.

There is a company named Polyverse Corporation that claims offer a Moving Target Defense Suite [55], based on a similar idea to our project. However, since it is not an open-work/open-source project, it is difficult to tell how effective the solution is.

Furthermore, Kaiyu Feng *et al.* [56] wrote a paper on preventing SQL injections with the help of Moving Target Defense and suggested a potential strategy with containerization technology combined with an orchestration tool such as Kubernetes, which is similar to our idea. They did however not implement the strategy and could therefore not test if it works in reality.

A method within the field of radio technology, known as Frequency-hopping spread spectrum (FHSS) [57], is already widely used in practice to prevent for example interference and eavesdropping. FHSS is an inspiration for this project and we hope to design a Moving Target Defense based on similar ideas.

To the best of our knowledge, there is no project or paper that has published details, examples or demonstrations with our suggested Moving Target Defense idea.

# 4 Design

This chapter describes our process of designing a moving target defense system. The chapter is organized as follows: first, we describe our definition of a moving target defense. Second, we enumerate the requirements and goals that guided our design process. Third, we describe the design choices and processes that we followed. Fourth, we detail the infrastructure design. Fifth, we discuss the algorithm that we used for switching targets.

## 4.1 Our Definition of a Moving Target Defense

In order to create a moving target defense, we first had to establish the concept and what we mean when using the term moving target defense. To do that, we will describe what static and moving targets are and how they differ.

A static target is an object that does not move. In the context of computer security, a static target is an object that is not changed or updated frequently. This could be a server, a website, or even an individual file. Static targets are typically easier for attackers to find and exploit because they do not change.

In contrast, a moving target is an object that is constantly changing. In the context of computer security, a moving target is an object that is frequently updated or changed. This could be a server, a website, or even an individual file. Moving targets should be more difficult for attackers to find and exploit because they are constantly changing. A moving target defense is thus a system that handles moving targets.

## 4.2 Requirements and Goals with the System

The moving target defense should be designed to work in conjunction with other security measures, such as firewalls and intrusion detection systems. The goal is to make it *more difficult* for an attacker to find and exploit vulnerabilities by constantly changing the system configuration. The moving target defense is not supposed to be a bulletproof defense system, hence traditional defenses are still required in conjunction.

## 4.3 Design Choices and Process

To implement a moving target defense, we first needed to identify some potential attack vectors that could be used against our systems. We then designed the moving target defense to address these threats. There are many different ways to design a moving target defense [58]. The most important thing is to ensure that the defenses are constantly changing and that they are effective against the identified threats.

Some of the threats that we identified that a moving target defense could potentially protect against include:

- Attacks that exploit static targets: A moving target defense can make it more difficult for attackers to find and exploit vulnerabilities in static targets.
- MITM attacks: A moving target defense can make it more difficult for attackers to intercept and modify communications between two systems.
- DoS attacks: A moving target defense can make it more difficult for attackers to overload a system with requests, making it unavailable to legitimate users.
- Physical attacks: A moving target defense can make it more difficult for attackers to physically access a system.

We decided to address these types of threats by constantly changing certain aspects of the system. We have come up with two defense mechanisms that should address the above threats. Each of these defense mechanisms is described in more detail below.

**Defense 1: Change physical location and IP address frequently**

One way to make it more difficult for attackers to physically access a system is to change the physical location of the system frequently. Changing the IP address of the system in combination with changing the physical location, makes it more difficult for attackers to intercept and modify communications between two systems. This makes it more difficult for attackers to find and target the system.

**Defense 2: Change software stack frequently**

Another way to make it more difficult for an attacker to find and exploit vulnerabilities is to change the software stack that the system uses frequently. This should make it more difficult for an attacker to fingerprint the system and makes it more difficult to exploit any vulnerabilities.

The two mentioned defenses above should individually help with the identified attack vectors, and in combination, improve the difficulty even more. More things can be done to further improve the difficulty, but those are out of scope for this thesis.

We also needed to consider the tradeoffs when designing the moving target defense. For example, we needed to balance the need for security with the need for availability and performance. Thus, in our design, we have chosen to focus on three key areas:

- **Security**: We wanted our defense to be effective against the threats that we have identified.
- **Availability**: We wanted our defense to be effective, without impacting the availability of our systems.
- **Performance**: We wanted our defense to have minimal impact on the performance of our systems.

To accomplish the two defense mechanisms outlined above, while also considering security, availability, and performance, we had several options to consider. Intuitively, with the most basic thought process, one could simply automate the process of changing the physical location, IP address, and software stack. This could be done by for example having a script that changes these things on a regular basis. However, this is not very practical as it would require a lot of effort to keep track of all the changes and ensure that everything still works as intended. A more practical solution would be to look at setting up hypervisors running virtual machines and then provisioning different images. These images could have different configurations, running with different software stacks. The IP address and physical location would be controlled by some kind of orchestrator.

However, starting a virtual machine and moving images around can be both heavy on resources and slow. This might not be the best solution in a production environment where speed and efficiency are important. We decided to go with a third option which is using container technology. Containers have several advantages over virtual machines. They are much lighter on resources, can be started and stopped very quickly and can be moved around more easily. This makes them a good candidate for our use case.

There are several container technologies available, but we decided to go with Docker.

The reason we have chosen Docker is that it is a very popular container platform and has good support for all types of applications. It also has a large ecosystem with many tools and services that can be used with it. The next step was to look at orchestrating the containers to different locations because Docker by itself is meant to run on a single machine. Instead of reinventing the wheel, we looked at existing orchestration tools such as Kubernetes, Apache Mesos, and Docker Swarm. We decided to go with Kubernetes because it is open-source, backed by a large company (Google), is very popular, and is widely available in most cloud providers. It also has good documentation, and a large ecosystem, and gives us the granular control that we need to design our moving target defense.

Kubernetes is a container orchestrator that can be used to manage a large number of containers across multiple machines. It can be used to automate the process of starting, stopping, and moving containers around. Kubernetes also has many other features such as load balancing, monitoring, and self-healing. All of these features are important for our use case. The next step was to look at how we could use Kubernetes to design our moving target defense.

The first thing that had to be done was to create a Kubernetes cluster. A Kubernetes cluster is a group of machines that are used to run containers. We could create a Kubernetes cluster on our local machines, or we could use a cloud provider that provides pre-configured Kubernetes clusters such as Google Cloud Platform, Amazon Web Services, Microsoft Azure, or Digital Ocean. We decided to use the official Minikube tool for local development and testing on our local machines. We were also given access to a cloud provider that is meant for researchers, called SNIC (Swedish National Infrastructure for Computing). We, therefore, used SNIC when we needed to do larger tests that couldn't be done on our local machines. However, SNIC does not come with a preconfigured Kubernetes cluster, so we had to set one up ourselves. More details about the configuration and setup are provided in chapter 5. We also decided to use DigitalOcean during some experiments to achieve realistic latency between different countries and continents, which is described in more detail in chapter 6.

The next step was to look at how we could automate the process of starting, stopping, and moving containers around. The common way is to use the Kubernetes CLI called Kubectl. However, this is a very manual process and is not suitable for our use case where we need to move containers between nodes automatically based on certain criteria. We, therefore, looked into writing a script that would execute Kubectl commands, however, we decided that it was more reliable and flexible to write a program that uses the Kubernetes API. There are several open-source libraries available for the Kubernetes API, in different languages. We decided to go with the official Kubernetes Java client because it is well-documented and likely to receive continuous development and support in the future. It also had support for most of the Kubernetes API resources that we needed to use.

## 4.4 Design of the Infrastructure

This section will give a general overview of the design of our infrastructure and then the actual hardware specifics will be detailed in section 5.1 about the cluster setup.

Kubernetes runs with a Master Node and a set of Worker Nodes. The Master Node is responsible for the management of the Worker Nodes. The Worker Nodes are where the containers are actually running. In our design, we have one Master Node and multiple Worker Nodes. The number of Worker Nodes can be increased or decreased depending on the needs of the system and shall remain flexible. The moving target defense will then communicate with the Master Node in order to determine which Worker Nodes the containers should be running on. An illustration of the design is displayed in Figure 4.1.

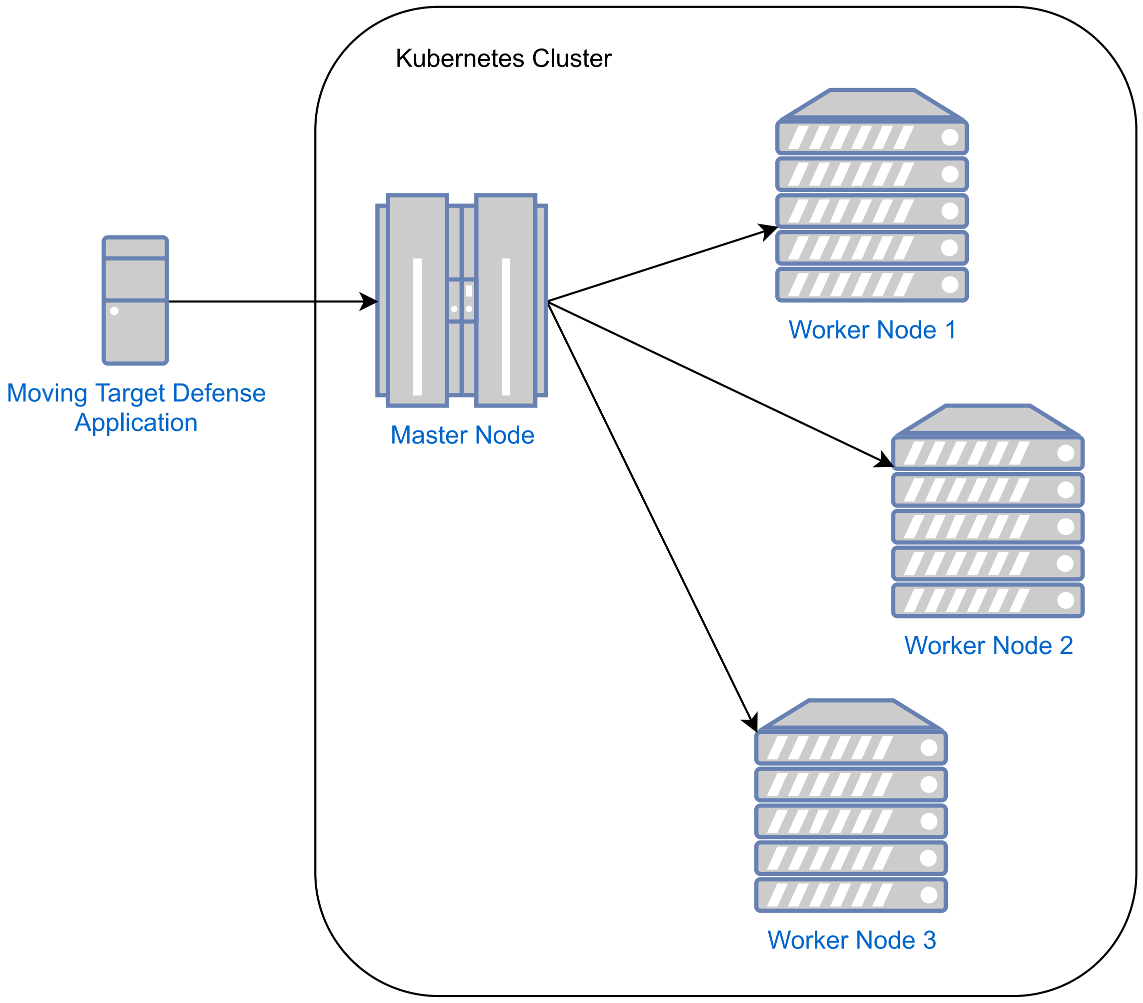

**Figure 4.1:** Infrastructure of a Kubernetes cluster with a moving target defense application.

In Figure 4.1, an adversary could be located in between any of the three connections between the worker nodes and the master node or target one of the nodes directly.

However, the figure shown is sort of a black box since there are no connections to the worker nodes other than the master node. This means the worker nodes are not providing anything useful. We have therefore come up with two more designs that extend this one. The first design is displayed in Figure 4.2.

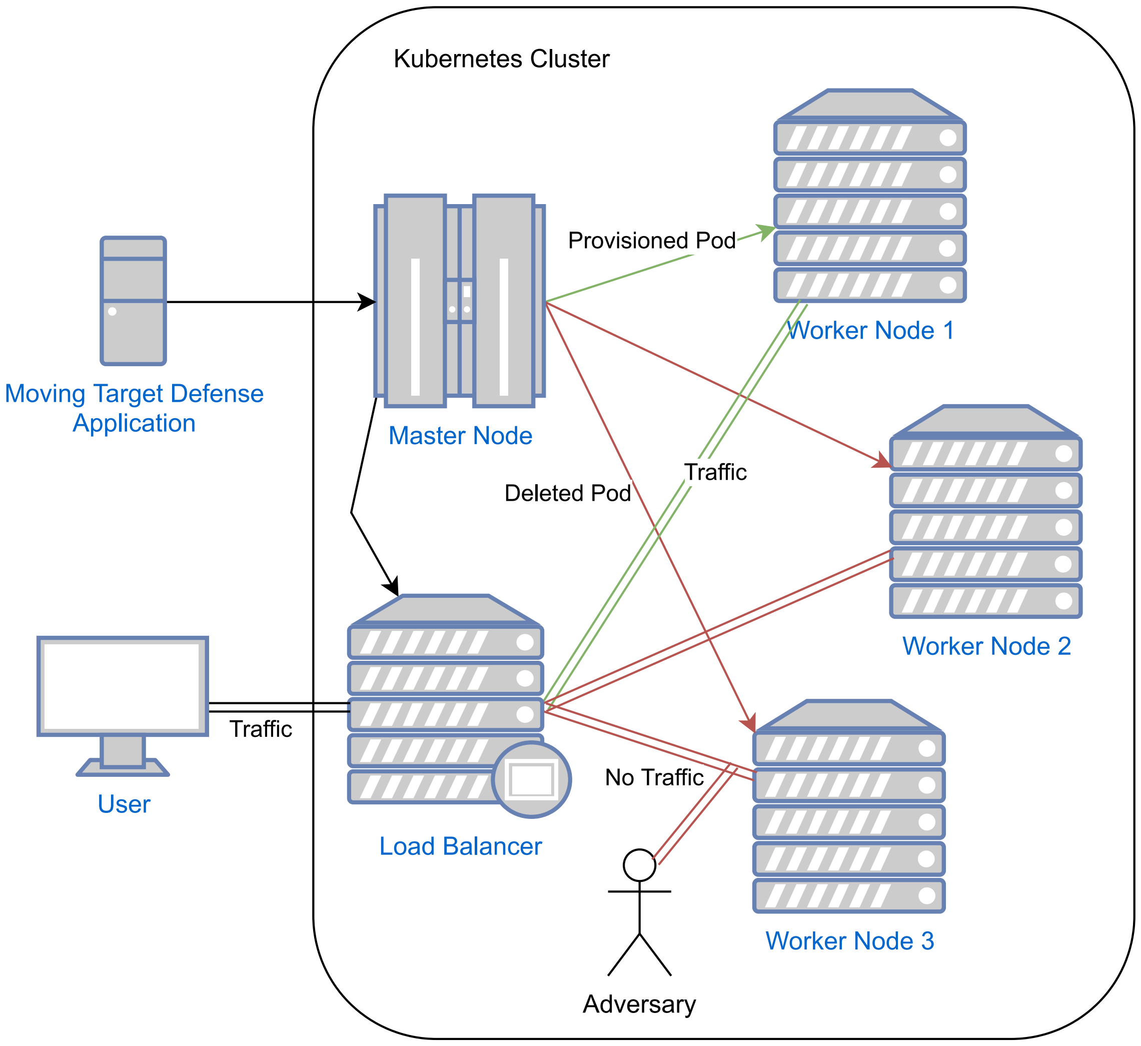

**Figure 4.2:** Infrastructure of a Kubernetes cluster with a moving target defense application and a Load Balancer.

The design in Figure 4.2, includes a load balancer that acts as a public endpoint to the Kubernetes cluster. It redirects traffic to all the available pods. The green arrow in the picture illustrates that the master node has provisioned a pod to the targeted worker node. While a red arrow indicates that the master node has deleted any pod if any, on the targeted worker node. The double-striped line indicates active traffic between the load balancer and the worker node since that node has an active pod running. Whilst the doubled-striped red line indicates no active traffic because the targeted worker node has no pod running. An adversary is also illustrated between the Load Balancer and Worker Node 3, to show that if that pod is not running, the adversary should not be able to intercept the traffic nor be able to attack that inactive node. However, a potential flaw is that the adversary may be able to target the load balancer directly and thus have any attack redirected to the active node, or if having internal network access to the load balancer, may be able to intercept all the traffic passing through it. This is why we have come up with one more design shown in Figure 4.3, that will show a different use case.

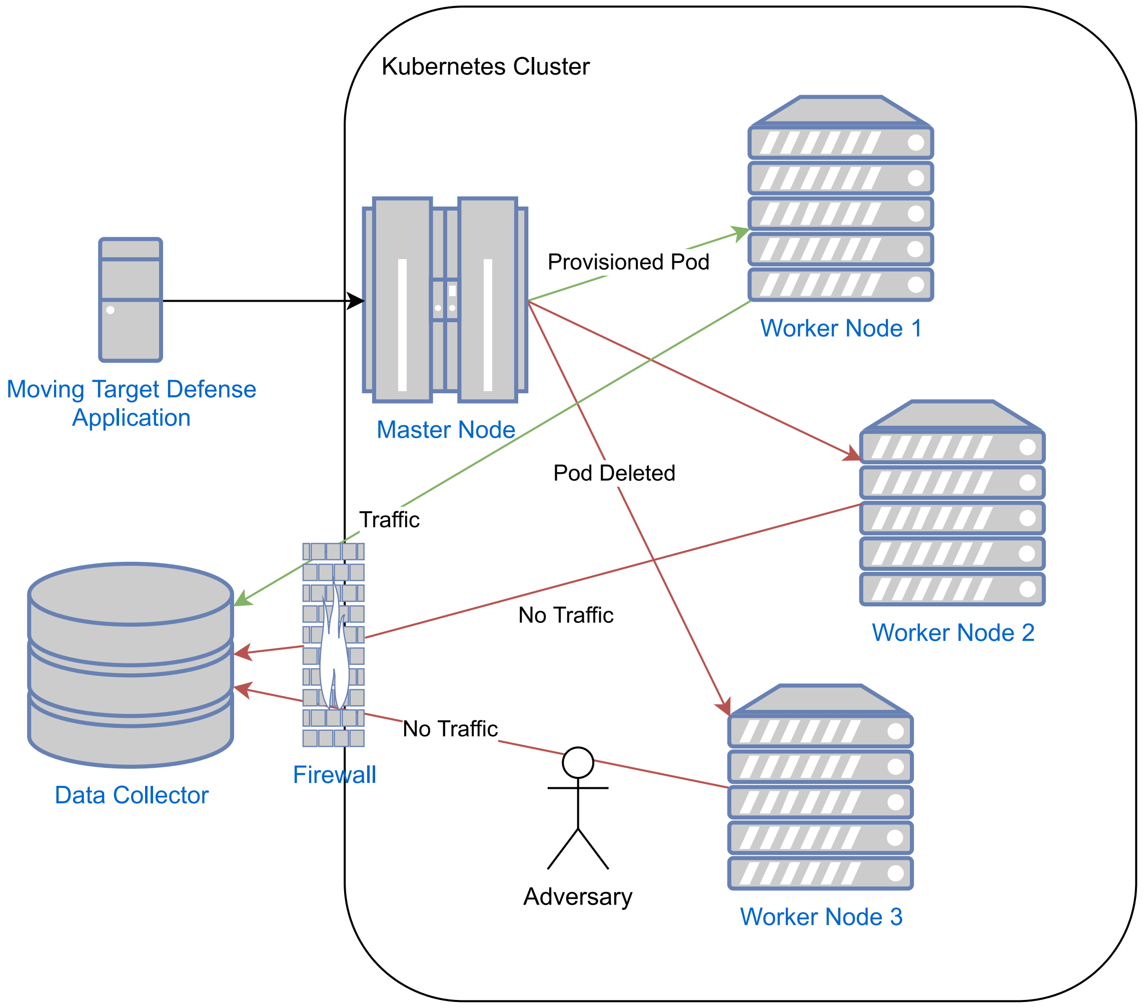

**Figure 4.3:** Infrastructure of a Kubernetes cluster with a moving target defense application and an external data collector.

As seen in Figure 4.3, there is a similar design to the one shown in Figure 4.2, however, in this design, there is no load balancer and the worker nodes merely act as data producers that send their data to an external data collector. This makes it so that there is no public endpoint to the cluster, thus no load balancer redirecting traffic to the active node, meaning it should be difficult for an adversary to track which node is currently active in order to attack it. In addition, since there is no public endpoint, the attacker must be located on the internal network of the node in order to attack it, since there is also a firewall that only allows traffic to flow in the outgoing direction of the cluster. Another thing that should be noted is that each node within the cluster could be located physically anywhere, for example in different cities, countries, and continents which would make it harder for an adversary to attack multiple nodes, especially when internal network access is required for an attack as in the Figure 4.3 design.

## 4.5 Switching Algorithm

The main goal of the algorithm is to change the attack surface of the system in an unpredictable way. To achieve this, the algorithm utilizes randomization on a couple of different parameters. The first one is on which of the worker node the active pod lives on. The second one is which deployment file to apply to the system, there may be multiple deployment files with different configurations. The active worker nodes and selected deployment files are randomized upon each iteration of the algorithm as seen on row 4 and 6 in Listing 4.1.

In order to achieve the swapping of pods between nodes, the algorithm uses the labeling system that Kubernetes provides. A deployment file can include a `nodeSelector` field which looks at the available nodes on the cluster and only deploys pods on nodes which has a label matching what is in that field. Outside of this algorithm, this matching is usually only used when the system is first starting up or if the deployment file is changed manually to update the system configuration. However, the algorithm is designed to take advantage of the fact that it is possible to change the labels programmatically. For it to work, the deployment files given to the application needs to have an entry in the `nodeSelector` field to be `mtd/node=active`. Then, on each iteration of the algorithm, it will randomly choose a new node to label `mtd/node=active`. When the randomly chosen deployment file is applied, it looks for a node with an `mtd/node=active` label on it and deploys the pod to that node.

We have come up with two versions of the algorithm, Listing 4.1 and Listing 4.2 below. The main reason for this was the need to update the functionality around the load balancer. This was because the first version had a slight gap in availability to the system, which resulted in lost requests when we tested the system. The difference start from row 7 in the listings above and mainly differ in that version 3 of the algorithm starts a second deployment so that two run simultaneity for a short time. The algorithm then waits a second for the load balancer to notice the new pod, and only after that is the old deployment and by extension pod deleted. This ensures at least in theory that there is now downtime from when the old deployment is deleted to when the new one is applied. The implementation of version 3 is shown in chapter 5 and the results of both versions are shown in chapter 6.

```
Remove previous deployments with the same name
Remove active labels
Create a list of available worker nodes, and if there is a
    currentNode, remove it from the list so it cannot be chosen as
    the next node
Choose a node from the list at random
Put an active label on the chosen node
Choose a random deployments from the settings
Apply or restart the chosen deployment, depending on if its a
    different or the same deployment file as the previous iteration,
     respectively
Wait for the pod to become ready before continuing
Wait a specified amount of time before starting a new iteration
Go back to line 2
```

**Listing 4.1:** MTD Algorithm V2

```
Cannot remove deployments from previous runs of the application
    since their names depends on how many iteration it ran for
Remove active labels
Create a list of available worker nodes, and if there is a
    currentNode, remove it from the list so it cannot be chosen as
    the next node
Choose a node from the list at random
Put an active label on the chosen node
Choose a random deployments from the settings
Start a second pod on the randomly selected node in (4)
Add a number to the end of the deployment name equal to the
    iteration count. This is done because we need to be able to
    distinguish between the two deployments currently applied
Wait for the pod to become ready before continuing
Remove active label from currentNode
Wait for 1 second so that the load balancer has had enough time to
    detect the new pod on currentNode
Delete the old deployment
Repeat from line 2
```

**Listing 4.2:** MTD Algorithm V3

# 5
# Implementation

This chapter describes the implementation of our moving target defense, which essentially is a program that automates the process of starting, stopping, and moving containers around in a Kubernetes cluster based on the settings and algorithms that we have defined. The implementation is built around four main parts, the Kubernetes cluster, the deployments, the code that makes up our application, and finally the CLI which controls the application. These parts are described in the following four sections.

## 5.1 Kubernetes Cluster Setup

In this section, we describe how we set up our own Kubernetes cluster on the SNIC cloud. As mentioned in the previous section, we decided to use SNIC for our larger tests that couldn't be done on our local machines. SNIC allowed us to set up virtual machines with different CPU, memory, and storage configurations. We also had the ability to add more machines to our cluster if needed.

The first thing that we had to do was to create a virtual machine that would be used as the Kubernetes master node. The master node is responsible for managing the other nodes in the cluster and orchestrating containers. We decided to go with the smallest machine size that SNIC offers that also satisfies the minimum requirements of Kubernetes, which is 2 CPUs and 2GB of RAM. We also created four additional virtual machines that would be used as Kubernetes worker nodes, with the same hardware configuration. The operating system running on the virtual machines was Ubuntu Server 20.04 LTS.

After the virtual machines were created, we had to set up a Kubernetes cluster on them. We then had to install Kubernetes on our master node, using the Kubeadm tool. There are different methods and configurations as to how to set up a cluster, so we will detail our configuration below for replication purposes. Listing 5.1 shows the configuration that is required on all nodes (Master and Workers), further more each step is commented in the listing.

```
# Update the host OS.
sudo apt update
sudo apt -y upgrade
```

```
# Install Docker runtime environment
sudo apt install -y docker.io

# Allow iptables to see bridged traffic.
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo sysctl --system

# Reboot the entire server.
sudo reboot now

# Check the status of the netfilter.
lsmod | grep br_netfilter

# The output should look like
br_netfilter           28672  0
bridge                176128  1 br_netfilter

# Add the kubernetes repository. Begin with installing curl &
   dependencies.
sudo apt-get install -y apt-transport-https ca-certificates curl

# Download and save the Kubernetes key rings.
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
    https://packages.cloud.google.com/apt/doc/apt-key.gpg

# Add Kubernetes to the repository.
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring
   .gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo
   tee /etc/apt/sources.list.d/kubernetes.list

# Install kubeadm and kubectl
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl

# Configure Docker's cgroup driver and restart the docker service.
sudo su
cat > /etc/docker/daemon.json <<EOF
{
    "exec-opts": ["native.cgroupdriver=systemd"],
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "100m"
    },
    "storage-driver": "overlay2"
}
EOF
systemctl restart docker.service
```

**Listing 5.1:** Configuration that is running on every kubernetes node (both master and workers).

Now that the basic set up of Kubernetes has been completed, the master node is configured as shown in Listing 5.2

```
# Start cluster, replace IP of master (api) in this command.
kubeadm init --pod-network-cidr 10.244.0.0/16 --apiserver-advertise
    -address=129.16.123.61

# If logged in as root user, do 'logout' or 'exit' if in a SSH
    session and make sure to be logged in as non-root.

# As a regular user, copy the kubeadm config so that it can be used
     by kubectl.
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

# Wait for master node to become ready. Check with.
kubectl get nodes

# Install Weave NET
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(
    kubectl version | base64 | tr -d '\n')"
```

**Listing 5.2:** Configuration that is running on the Master node only.

In addition, worker nodes were set up by following a similar procedure, except only the master node needs to setup networking (Weave NET) and instead of using the kubeadm init command, we used the kubeadm join command, providing it with the cluster join token and hash that was given from the master node's kubeadm init command. After this process was completed, we had a fully functioning Kubernetes cluster consisting of one master node and four worker nodes in different locations. The worker node configuration is shown in Listing 5.3.

```
# Every worker node run,
kubeadm join <IP> --token <token> --discovery-token-ca-cert-hash <
    discovery token>

# If nothing is happening, generate a new token.
kubeadm token create --print-join-command

# Check if the node has joined the cluster, run on Master node.
kubectl get nodes
```

**Listing 5.3:** Configuration that is running on the Worker nodes only.

The configurations in listing 5.1, 5.2 and 5.3 are derived from the official kubernetes documentation [59], [60], [61].

## 5.2 Code Design and Structure

This section describes the code design and structure of our moving target defense application. The application is written in Java and makes use of the Kubernetes Java client library. We will first describe the design of the application and then go into detail about the different packages and files that make up the codebase.

### Application Design

The goal of our application is to provide a way to automate the process of starting, stopping, and moving containers around in a Kubernetes cluster. We also knew from the start that we wanted a convenient way to start and stop the application, as well as a way to easily configure the settings. Therefore, we decided to make our application into a command line interface (CLI) application. This meant that we would have to parse command line arguments in order to figure out what the user wanted to do. For that reason, it made sense to use the Model-View-Controller (MVC) pattern for our application [62]. The MVC pattern is a way of organizing code into three different parts, the model, the view, and the controller.

The model is responsible for containing the data and business logic of the application. In our case, this includes things like the settings that the user has configured, as well as the functionality that we call upon to move containers around. The view is responsible for displaying information to the user. The controller is responsible for connecting the model and view together, as well as handling any input from the user. The Kubernetes client came with useful functionality but it did not provide the exact functionality that we needed out of the box, thus we had to take bits and pieces and combine them to create the features that we needed. We also wanted the client library to be replaceable in the future, hence we wrapped all the features and abstracted away the code so that we got the exact code that we needed.

### Package, File Structure, and Application Flow

The code for our application is organized into different packages, with each package containing a specific group of related classes. The main packages in our application are: model, view, controller. The view package contains MenuView, MtdView, and SettingsView. The model package contains Settings and a package we call kubernetes. The kubernetes package contains our own classes that controls kubernetes but they make use of the official client library. Finally, the controller package contains our MTD algorithms and the corresponding controller classes for the views, MenuController, MtdController, and SettingsController. The structure is illustrated in Figure 5.1.

As illustrated in Figure 5.1, the application flow starts at MtdMain where the application reads any input parameters. If there are no input parameters, MtdMain executes MenuController which in turn displays a menu with the help of MenuView. The MenuController then waits for user input until further action is taken, then

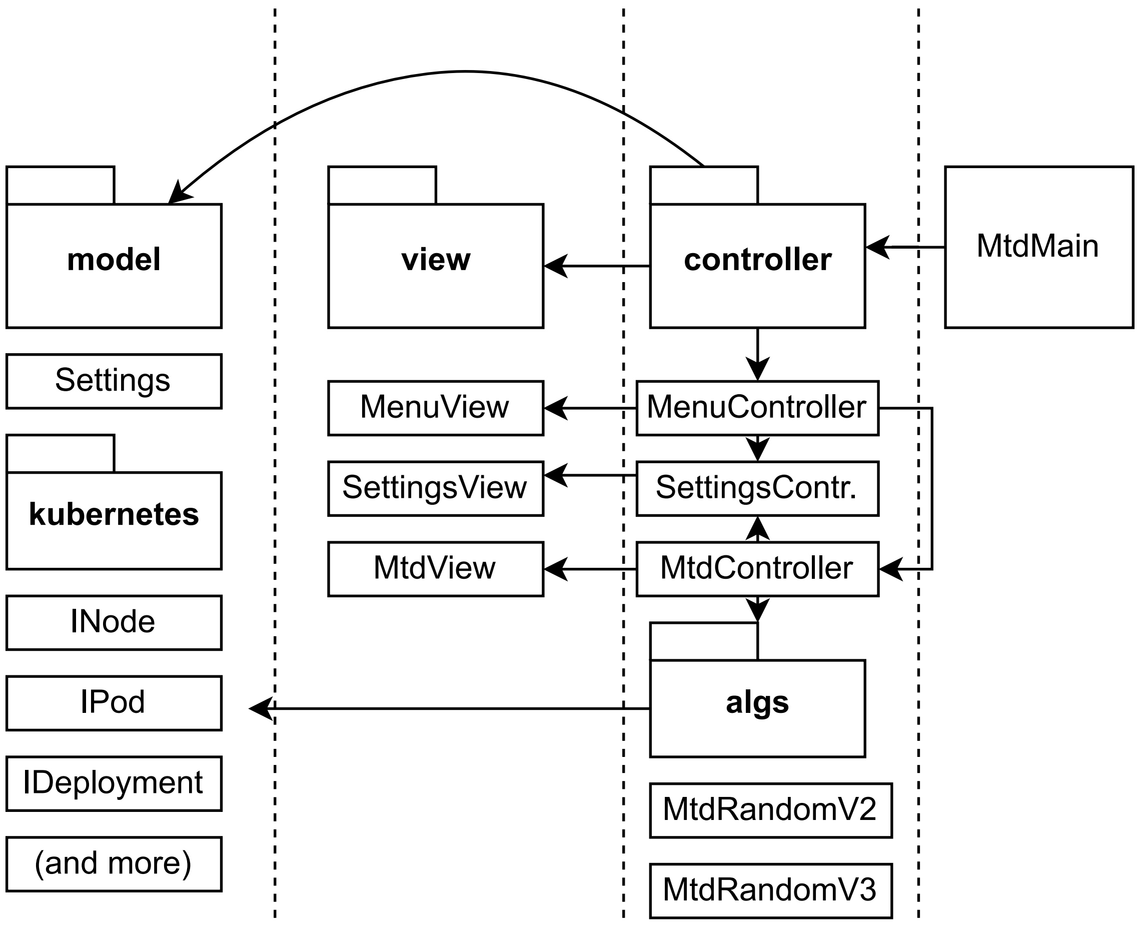

**Figure 5.1:** Showing the code structure and how the application flows, starting from MTD main.

finally creates or loads settings via the SettingsController, which is then used in the MtdController to start the Mtd algorithm. The system is designed such that it can be extended to choose different algorithms based on settings. Finally, the Mtd algorithm is using our kubernetes classes to control the MTD. Since there are interfaces encapsulating the kubernetes classes, it is possible to easily replace the client library with a different one or a different version, if that need arises in the future.

We have also provided unit tests which can be found in the test directory. These tests are executed automatically as part of the build process and they provide good coverage for the codebase. They have also proven useful while creating the MTD algorithms as it allows for running them without the overhead of the entire application.

The entire application (excluding dependencies) amounts to just over 3300 lines of code. This includes comments, blank lines, and unit tests, but excludes javadocs comments. The code is well organized and it should be easy to follow. We have tried to stick to a consistent coding style throughout the codebase. The code is also fully documented with javadoc comments.

### Algorithms, Pseudo-Code and Complexity

We have created and implemented two different algorithms that accomplish very similar things, as mentioned in the design chapter. The more recent version is what we call V3 and the description of this algorithm is shown in the design chapter, Listing 4.2. The implemented algorithm is provided in Listing 5.4 below, as pseudo-code and has been simplified to include only the most essential parts.

```
oldDeployment.delete();

for (node : nodeList)
    if (node.getLabels().containsKey(LABEL_KEY))
        node.deleteLabel(LABEL_KEY);

while ()
    nodeList = NodeTools.getWorkerNodes();

    if (currentNode != null) {
        nodeList.removeIf(tmpNode -> tmpNode.getName().equals(currentNode.getName()));
    }

    oldNode = currentNode;
    currentNode = nodeList.get(random.nextInt(nodeList.size());
    currentNode.addLabel(LABEL_KEY, LABEL_VALUE);

    currentDeployment = deployments.get(random.nextInt(deployments.size()));
    currentDeployment.apply(deploymentCounter);

    while (!(currentNode.getPods().size() == 1 && currentNode.getPods().get(0).getPhase().equalsIgnoreCase("running"))) {}

    currentNode.deleteLabel(LABEL_KEY);
    sleep(1000);

    if (!oldDeploymentName.isEmpty()) {
        oldDeployment = new Deployment(oldDeploymentName, "default");
        oldDeployment.delete();
    }

    oldDeploymentName = currentDeployment.getName();
    deploymentCounter++;

    sleep(timeBetweenSwap);
```

**Listing 5.4:** pseudocode for the algorithm

The pseudo-code algorithm amounts to 34 lines of code as shown in the figure, but the real implementation in Java amounts to 177 lines of code. The algorithm runs forever until stopped, so the worst-case time complexity for one iteration of the algorithm is $O(n)$ where $n$ is the number of nodes or the number of provided deployments, whichever is largest. Since the number of nodes is typically relatively few (not in the millions) the time to execute the algorithm is negligible. However, there are delays in terms of I/O inside the algorithm such as sending commands to Kubernetes and waiting for pod deployments. Which in reality, means that one full iteration (with complete node swap) takes at minimum between 3-5 seconds.

## 5.3 Building, Installing, and, Using the MTD Application

This section describes the requirements of the MTD application, how to build it and how to run it.

In order to use the MTD application, a Kubernetes cluster must be set up. This can be done by following the information provided in the previous sections. Once the cluster has been set up, the MTD application must be installed on a machine that can run uninterrupted, preferably near the cluster. The application could also be containerized and run from within the cluster. The installation instructions can be found in this section. After the installation is complete, a user can run the MTD application by using the command line interface (CLI). The CLI provides all of the options for running and managing the MTD application.

**Requirements**

The MTD application must be run on a machine with Java 11 or later installed. The application is a standalone program and does not require any additional dependencies as they are all embedded inside the JAR. In addition, the Kubernetes cluster must not have any other pods running under the default namespace, although they can run under different namespaces.

**Building**

The MTD application is built using Maven. To build the application, simply run `mvn package` in the project's root. This will create a JAR file in the `target` directory named `MtdJava-1.0-SNAPSHOT.jar`. To install the application, simply move it to a folder from where it should be run, preferably on a server on the same network as the Kubernetes master node to reduce latency and potential network interruptions. The server must also have Kubectl installed with a cluster config since the MTD application is using this config to connect to the cluster.

**How to Use**

After the application has been built and installed, the next thing that needs to be done is to create a Kubernetes deployment file, which will be used by the tool to provision pods. The deployment must include a `nodeSelector` with the label `mtd/node=active`, since this label is used to direct the pod to a specific node. Also, the current version of algorithms only supports 1 replica and must use the default namespace. An example configuration file is shown Listing 5.5:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  revisionHistoryLimit: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
        mtd: pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
      nodeSelector:
        mtd/node: active
```

**Listing 5.5:** Example deployment file with minimum fields required for the MTD application to work.

After the configuration is created, the tool can be run in two ways. The first way is to run it with a CLI interface using the following command:
`java -jar MtdJava-1.0-SNAPSHOT.jar`

It will bring up a menu that lets the user choose whether to create a new MTD configuration or load and run an existing one, as shown in Listing 5.6.

```
Moving Target Defense
=====================
Menu Options
1. Create & Run Settings File
2. Load & Run Settings File
3. Exit
---------------------
Make a selection (number):
1
Name your settings file (including file ending .yaml or .yml):
MTDSettings.yaml
```

**Listing 5.6:** The CLI for the application

The second method is to run the application with a parameter which is the file path to an existing MTD settings file. Doing so immediately starts the MTD. This is more suitable for example when setting up a cronjob, running the MTD from a script, or simply a convenience rather than having to start the interface. The syntax for running with a parameter is as following:
`java -jar MtdJava-1.0-SNAPSHOT.jar path/MTDSettings.yaml`

The MTD settings file is a YAML file that can be edited in a text editor. The file contains all the information needed to run the MTD on a given deployment or multiple deployments. It is therefore important to follow the existing structure of the file to avoid any potential errors. An example MTD settings file can be seen in Listing 5.7.

```
name: "MTDSettings.yaml"
serviceEnabled: true
serviceFileName: "LoadBalancerService.yaml"
deploymentFileNames:
- "DeploymentGVMTest.yaml"
- "DeploymentGVMTest2.yaml"
logToConsole: true
logToFile: false
```

**Listing 5.7:** Example settings file with load balancer and two different deployments that will randomly alternate.

Once the tool is running, it will start to move the containers around in the Kubernetes cluster, based on the settings defined in the configuration file. The tool will output information about what it is doing to `stdout`, which can be redirected to a file if needed. An example output is shown in Listing 5.8.

```
Starting MTD alg.
Randomly selected node: MTDNode-02, adding active label.
Randomly selected Deployment: DeploymentGVMTest.yaml
Applying deployment with name: gvm-deployment1
Trying to find the new pod...
Did not find new pod, waiting 1 second.
Did not find new pod, waiting 1 second.
Deleting active label on node: MTDNode-02
Swap finished, waiting 10000ms before next iteration.
============= Iteration Done ============
```

**Listing 5.8:** The CLI for the application

The MTD is designed to be run as a continuous process, and will therefore keep moving the containers around until it is stopped. It can be stopped by pressing CTRL+C in the terminal where it is running or by killing the process.

If the user runs the application without a parameter which brings up the menu and then selects the first option “Create & Run Settings File”, it will start a wizard that guides the user through the process of creating an MTD settings file. An example of the wizard is shown in Listing 5.9. The MTD immediately starts after the settings file has been created and will give an output like the one shown in Listing 5.8.

```
--- MTD Wizard ---
Do you want load balancing? (y/n)
y
Type your Service YAML file name (including .yaml):
LoadBalancerService.yaml
Do you want logging to file? (y/n)
n
Do you want logging to console? (y/n)
y
Type your Deployment YAML file name (including .yaml):
DeploymentGVMTest.yaml
Do you want to add another Deployment? (y/n)
y
Type your Deployment YAML file name (including .yaml):
DeploymentGVMTest2.yaml
Do you want to add another Deployment? (y/n)
n
```

**Listing 5.9:** The CLI for the application

The first question in Listing 5.9 is whether or not the user wants the MTD application to set up a load balancer. If answering yes, then the user will be prompted to give a name for a load balancer settings file, also known as a Service within Kubernetes. This file must be placed in the same directory as the MTD settings file and will be used by Kubernetes when setting up the load balancer. An example load balancer file is shown in Listing 5.10.

```
apiVersion: v1
kind: Service
metadata:
  name: lb-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30008
```

**Listing 5.10:** Example Kubernetes Service file that acts as load balancer within the MTD

The following questions are the choices of logging, either to a file, console, or both. The last input needed is the Kubernetes Deployment files that the application needs to be able to run. Using multiple different deployment files can be used to alternate images and configurations. The MTD system will randomly select a deployment file from the list during each node swap. For example, one deployment file could use an Apache image while another uses an Nginx image, and thus provide the same service to the end user, but potentially alternate vulnerabilities.

# 6

# Tests and Results

In this chapter, we will give a high-level overview of the experiments that we conducted in order to test our moving target defense. The goal of our experiments was to see if we could successfully move a container from one node to another, without any downtime or user-visible effects. We also wanted to see the performance of moving a container multiple times in quick succession while still maintaining availability. Lastly, we wanted to see if the moving target defense had any positive effect on securing a system against various types of attacks.

## 6.1 Performance and Availability

The goal of the experiment was to determine how well our system performed while the moving target defense was active. We did this by looking at three main metrics: CPU utilization, memory usage, and requests per second (RPS). We also wanted to see if there were any negative side effects of using the moving target defense, such as an increased latency or errors in the system. We also conducted the tests with the swapping algorithms running at different frequencies to see if that made a difference.

As part of our performance analysis, we wanted to monitor the CPU and memory usage of our nodes during different activities. We used the Prometheus and Grafana tools for this purpose as they are commonly used in such monitoring setups.

Prometheus is an open-source system for monitoring and alerting. It scrapes metrics from configured targets at given intervals and stores the data in a time-series database. Grafana is a visualization tool that can query Prometheus (or other data sources) and create dashboards to visualize the data. We set up Prometheus to scrape metrics from our Kubernetes cluster at a 1-second interval. We configured it to scrape metrics from nodes in our cluster, as well as from the Kubernetes API server. We also installed the node_exporter on each of our nodes to expose machine-level metrics such as CPU and memory usage.

Grafana was then used to create dashboards with graphs visualizing the data collected by Prometheus. These dashboards were used to monitor the resource usage of our nodes during different activities.

## Testbed

As mentioned in section 5.1, we set up a Kubernetes cluster on the SNIC cloud with one master node and four worker nodes. The master and worker nodes had 2 CPUs and 2GB of RAM. We also created six additional virtual machines that would be used as Kubernetes worker nodes, with the same hardware configuration, giving us a total of ten worker nodes.

For the initial testing of the uptime of the system, we first used a browser to send HTTP requests during each node swap, by simply refreshing the page frequently. Once we saw that it was working, we moved on to more reliable testing, so we wrote a Python script shown in Listing 6.1. It sends a request to the load balancer every 200ms to test to availability of the system while running the MTD algorithm.

```python
from time import sleep
import requests
from datetime import datetime

wait_time = 0.2
ip = ""
port = ""
url = "http://" + ip + ":" + port + "/"

while True:
    try:
        res = requests.get(url, timeout=5)
        print("{} {}".format(datetime.now().time(), res.text))
        sleep(wait_time)
    except requests.exceptions.ConnectionError:
        print("{} Failed to establish connection".format(datetime.
    now().time()))
        sleep(wait_time)
    except requests.exceptions.ReadTimeout:
        print("{} Read timeout".format(datetime.now().time()))
        sleep(wait_time)
```

**Listing 6.1:** requestSender.py

For the final test of our system, we used the ApacheBench tool to generate load on our system. ApacheBench is a tool for measuring the performance of HTTP servers. It can be used to simulate different numbers of users accessing a website or application. We ran ApacheBench from a separate machine, in order to more accurately simulate real-world traffic. We configured ApacheBench to make 100000 requests to the system for the small container images and 2000 requests for the system with larger images.

We also increased the size of the containers that we were using, in terms of CPU usage, memory and storage. We did this because we wanted to see if the moving target defense would still be effective when the containers were larger and had more resources allocated to them. The larger image is a WordPress image that was $\sim$ 600MB in size, we chose this image because it somewhat reflects a real world application in terms of CPU, memory, I/O, database connections and so on.

## Test 1 - Performance

For the performance test, we ran the MTD for 1 minute on 3 nodes to measure the hardware usages of the system. The algorithm was setup to make swaps with an interval of 10 seconds to make it clear in the graphs when the swaps occurred. We also ran them under simulated load to test if the MTD still functions. Lastly we tested the system to see if running MTD on a system decreases the amount of requests it can handle.

### Results

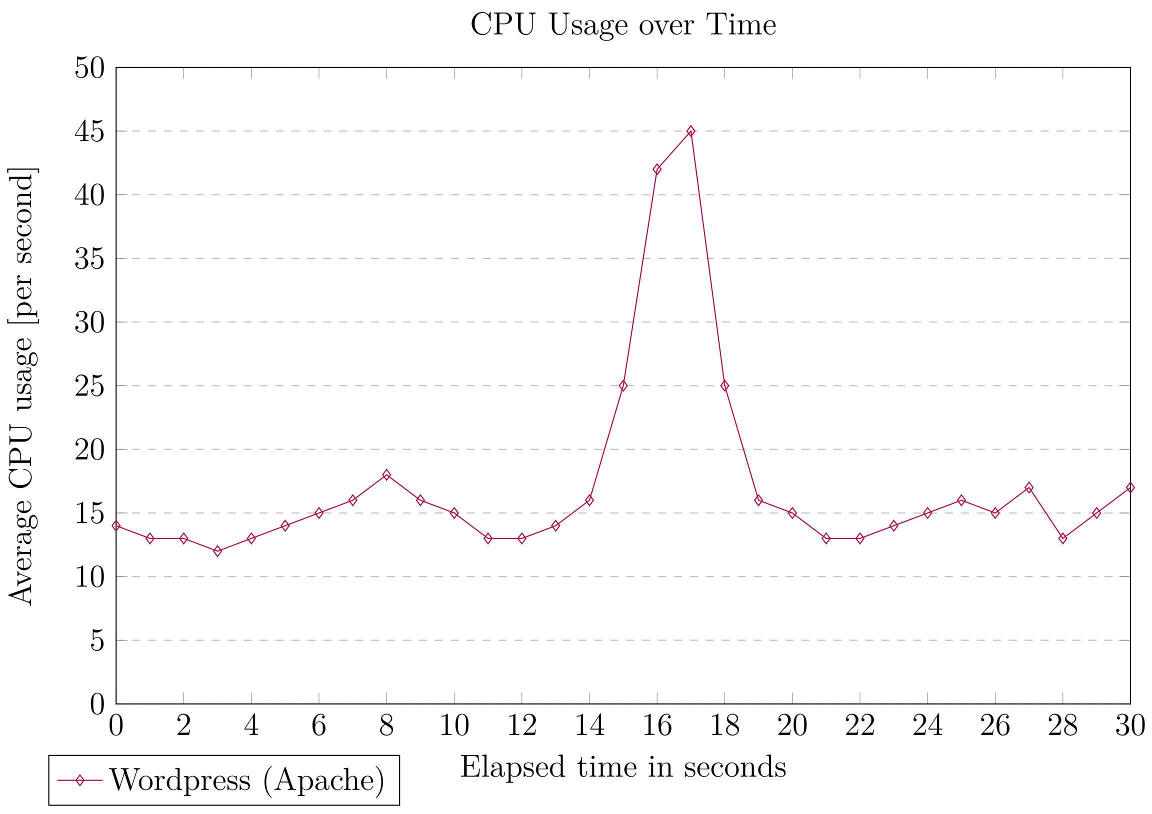

**Figure 6.1:** CPU usage of one node during the swapping of a Wordpress image under no load.

Figure 6.1 shows the CPU usages of one of the nodes when running the MTD with the WordPress image under no simulated load. The moment a swap occurs can clearly be seen by the sharp spike. Looking at the graph, we can confirm that a swap took around 3 seconds from start to finish which validates the results we got from the output logs of the application. The increase in CPU usage during a swap comes out to 29%.

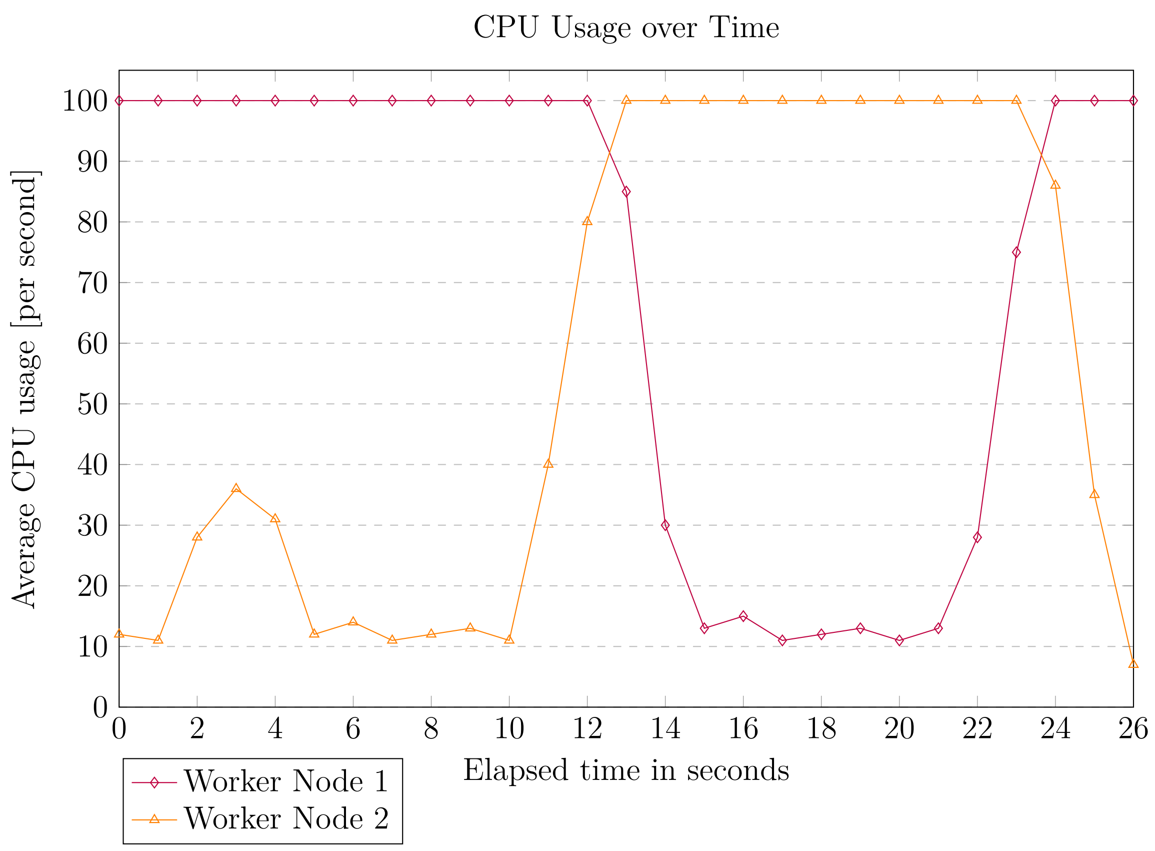

**Figure 6.2:** CPU usage of two nodes swapping a Wordpress image under load from ApacheBench. For example, the CPU usage shown on second 12 is the average usage between second 11 and second 12.

Figure 6.2 shows the CPU usage of two of the worker nodes when running the MTD with the WordPress image with a simulated load from ApacheBench. The two lines are overlaid on top of each other to make it clear how the timing works when a swap occurs. As we can see, the peaks of the two lines (running at 100%) are very near each other and almost overlap. The reason they do not overlap perfectly on the graph is because the CPU usage is measured as an average over the past second, which means that on second 12 in the graph, the average CPU usage between second 11 and second 12 is indeed 100%, which is a perfect overlap.

| Performance Tests with ApacheBench | | |
|---|---|---|
| Image Name | RPS without MTD | RPS with MTD |
| Nginx | 2834.44 | 2887.66 |
| httpd | 2756.12 | 2633.62 |
| WordPress | 11.97 | 13.06 |

**Table 6.1:** Results from stress tests when MTD is on versus when it is off

As seen in Table 6.1, there are some small fluctuations in the measured RPS, but this can be caused by a number of things besides the design and implementation of the MTD system. For example, there may be background processes running on the same machines that are impacting the performance of the system, or network congestion, running the test several times gave small random variations. In general, the results show that the MTD system does not have a significant negative impact on the performance of the system.

## Test 2 - Availability

The goal of the availability test is to show if the MTD system significantly decreases the uptime of the system. We measure the uptime by monitoring how often requests to the system are successful. We also used three different methods to test the uptime and generate load on the system, manual testing, our requestSender.py, and ApacheBench, using the same configurations specified in the earlier sections. We also used the three different types of container images, small (Nginx and Apache), and large (WordPress running on Apache).

We also tested how frequently the MTD algorithm can swap nodes before it causes significant downtime. We did this by starting the algorithm with a setting of 0 seconds and gradually increasing it to a level that was satisfactory.

### Results

The first test with the browser showed that the V2 algorithm caused occasionally timed-out requests during node swaps, which is why we created the V3 algorithm. The same test on the V3 algorithm showed no dropped requests. Furthermore, the rest of the tests are only performed on the V3 algorithm since it works better.

The following test with requestSender.py showed us that the load balancer randomly redirects packets between two active nodes during a swap, and that occasional TCP packets are timed out for a very brief moment after the older pod goes offline, since the load balancer does not update the status of the pod instantly. However, due to this shortened window, a typical HTTP implementation such as in a web browser, tries to resend the HTTP request in quick succession. Due to the shortened time window, it takes for the load balancer to get updated in algorithm V3, HTTP requests still reach their target despite some packets being timed out, but with some added latency in such rare cases.

| Availability Tests with ApacheBench | | | | |
|---|---|---|---|---|
| Image Name | # of Requests | Requests per second (RPS) | # of Failed Requests | Time Elapsed (sec) |
| Nginx | 100000 | 2887.66 | 0 | 34.630 |
| httpd | 100000 | 2633.62 | 0 | 37.971 |
| WordPress | 2000 | 13.06 | 0 | 153.098 |

**Table 6.2:** Results of Availability tests on three different docker images

Table 6.2 shows that all three tests with ApacheBench showed zero failed requests, which means that the kubernetes cluster had 100% availability. The MTD algorithm handled the potential downtime when the swaps happened and the loadbalancer managed to send and receive requests to the right node.

Furthermore, we tested what the highest swapping frequency setting was. From a performance perspective the algorithm does not have to wait at all before being able to start the next iteration. However, if we measure the amount of failed requests the system gets as you lower the swapping frequency, the minimum achieved wait time between a swap before it started to cause issues was less than 1 second. We also wanted to see if added latency between the worker nodes and the master node had any effect on the system as in how fast it can execute or if it would cause any inconsistencies in terms of availability. We performed that test by hosting the nodes in different regions on the cloud, thus achieving a natural latency of $\sim 90ms$ (instead of $\sim 1ms$). The MTD algorithm performed similarly as to the previous tests and we were unable to find any issues due to the increased latency.

## 6.2 Security – Fingerprinting and Vulnerabilities

The moving target defense system has two defense mechanisms as mentioned in the design chapter. This section aims to verify and analyze the security of each mechanism by actively fingerprinting and exploiting the system. Each test is detailed below along with the result of the defense mechanisms.

### Test 1 - Physical tampering

To test if the system helps defend against physical tampering such as side-channel attacks, we simply monitored the activity on each node such as the resource usage before and after a pod was deployed and deleted from the nodes.

#### Results

Our tests confirmed that the system was changing the physical location of the containers as it moved them between nodes and that the process and RAM activity were restored on the idle nodes, which means that there was no activity to tamper with once the pod has been swapped to another node. If an attacker were to physically tamper with a node, they would only be able to do so for a short period of time before the node was moved and their access was cut off. Although, depending on the setup, the pod will be redeployed to a given node eventually and allow the attacker to try again at a later stage.

However, the container image remains in the cache even after the pod is deleted from the node. This means that if an attacker could somehow gain access to the node, they would be able to find and tamper with the cached image in order to deploy a malicious container. To mitigate this, the system could be configured to delete the cached images after a certain period of time or when the pod is deleted. Although, that would increase the overhead of the system as it would have to fetch the images from the registry more often.

## Test 2 - Physical destruction

To test how the system handles destruction, we simply simulated destruction by switching off the power to one of the nodes in two different cases. The first case was when switching an idle node off to see if the algorithm tries to deploy to the "crashed" node. The second case was when we switched off an active node that was running a pod.

**Results**

Case 1, Kubernetes automatically marked the node as "NotReady" after a few seconds at which point it is no longer considered a valid node by our MTD algorithm, and therefore never deployed to it.

Case 2, Kubernetes automatically detects the failed pod after a couple of seconds and tries to schedule the pod to another node. This results in expected behavior however, since the deployment targets nodes labeled as active, the scheduling will be waiting for that label to appear on a different node. The current version of the MTD algorithm would also have to check the status of the crashed node and apply the active label to a different node in order for the rescheduling to work. If the algorithm implements this suggestion, there is an expected downtime that will last about 5-10 seconds during a node crash, about 5 seconds for the node crash to be detected, and another 3-5 seconds for redeployment.

## Test 3 - Fingerprinting, Vulnerability scanning and Exploiting a Specific Node

We have tested the system against active exploitation, fingerprinting, vulnerability scanning, port scanning, and man-in-the-middle attacks. In these tests, we assume that the attacker is located in a situation where they have network access only to a specific node.

The MTD was set up with a container image that was vulnerable to Heartbleed [44] in order to conduct meaningful tests. The MTD was then running with a swapping interval of 10 seconds.

Case 1, was to test of exploitation was to run Metasploit's Heartbleed module against the system.

```
use use auxiliary/scanner/ssl/openssl_heartbleed
set VERBOSE true
set RHOSTS ip-address
set RPORT 8443
set action DUMP
set leak_count 2000
run
```

**Listing 6.2:** Metasploit Heartbleed configuration

Case 2, was fingerprinting and Vulnerability scanning which was conducted with OpenVAS Greenbone Security Assistant, using all the NVTs and scanning options.

Case 3, The man-in-the-middle attack was conducted by sniffing the packets with Wireshark and to measure the amount of data retrieved.

Case 4, The port scanning test was conducted by running Nmap against the system.

**Results**

Case 1, The Heartbleed attack ran successfully for less than 10 seconds and was then canceled due to the pod being moved. Since the pod is no longer on the same node, the attacker can no longer exploit it. However, the goal with the Heartbleed exploit is to extract sensitive information such as private keys or passwords, this however takes time as the data continuously changes in memory.

Case 2, During the fingerprinting, it was observed that only the open ports on the system were being scanned and reported back. The fingerprinting did reveal the service and its vulnerabilities but only if it managed to perform the scans while the chosen node was active. Hence, timing is critical for the attacker. If the attacker's goal is to exfiltrate data, they need to have access to the system at the same time as the pod is deployed on the node. With two worker nodes running, OpenVAS was successful 50% of the time. Hence more worker nodes reduce the probability of the scan being successful.

Case 3, Like in the other cases, the man-in-the-middle attack was only successful while the pod was deployed on the node. This was due to the fact that, when the pod is redeployed, the traffic is rerouted and the attacker no longer has access to it. However, the packet sniffing continued as the pod was once again were deployed to the compromised node.

Case 4, The Nmap scan revealed the open port on the system including the service name, however, like the other tests, it was only successful while the pod was deployed on the node.

We also performed the same tests as in case 1-4, but by accessing the service via a load balancer which redirects all the traffic to the active node. In this situation, the Heartbleed attack and all the other tests were successful 100% of the time. This is due to the fact that, even though the pod gets redeployed, the load balancer still points to the same node. However, due to the nature of the Heartbleed attack, dumping memory takes time, and since the memory is random, it takes time for the attack to retrieve anything meaningful. With our setup, dumping 100 MB of memory took about 1 minute, at which point the MTD had swapped nodes 6 times. This means, the memory dumped is from different nodes and the chance of retrieving useful memory is lower than normally, partly because there are different nodes, but also because the memory of the deployed pod is fresh during each swap thus less likely to contain sensitive data.

## Test 4 - Fingerprinting, Vulnerability scanning and Exploiting via Load Balancer

This leads us to the testing of Defense 2, “change of software stack frequently”. In this test, we used an old vulnerable version of Apache and the latest version of Nginx with no known vulnerabilities. These two pieces of server software were used in the same moving target defense system and swapped between each other at random. We assume the position of going via the load balancer in order to get meaningful test results other than what was already shown above. In which case the traffic is always redirected to the active node. The same tools (OpenVAS and Nmap) were used to scan the system for fingerprints and vulnerabilities.

### Results

OpenVAS and Nmap both reported the fingerprint randomly, that is either Nginx or Apache, depending on which server was active during the conducted test. The same goes with the vulnerability scan which showed the given vulnerability only if the vulnerable deployment was running at the time of the scan. Conducting the tests 10 times resulted in the vulnerability being shown 50% of the time, as well as the fingerprint 50% of the time.

The mentioned attacks against a specific node gave us the probability of success given by $1/n$ where $n$ is the number of nodes. Or in the case of attacks that were mitigated by swapping software stack, $1/d$ where $d$ is the number of deployments. If the software stack swapping is combined with node swapping and the adversary is only able to attack a specific node then $1/n * 1/d$. For example, executing an attack against the vulnerable Apache image while only having access to 1 of 10 nodes and with 5 different software stacks (4 of which are not vulnerable), would give $1/10 * 1/5 = 2\%$ chance of success given the attacks tested.

## Test 5 - Backdoor Mitigation

During the development of the moving target defense, we discovered that there are different types of storage in Kubernetes. The two types are called “emptyDir” and “persistentVolumeClaim”. We found that the “emptyDir” storage is not persisted after the pod is deleted, while the “persistentVolumeClaim” storage is. This means that an attacker could create a backdoor on the target application and it should be created inside the corresponding volume type. If it is created inside an “emptyDir” volume and it should be wiped out when the pod is deleted. However, an attacker could create a backdoor in a “persistentVolumeClaim” volume and it should remain after the pod is deleted. We, therefore, wanted to test if our moving target defense mitigates backdoors in either of the two types of volumes.

In order to test our backdoor mitigation strategy, we uploaded a simple PHP shell in a “persistentVolumeClaim” volume as well as in an “emptyDir” volume while running the MTD algorithm.

**Result**

The MTD algorithm deletes the pod that contained the PHP shell and created a new pod on a different node. We found that the PHP shell was still present in the “persistentVolumeClaim” volume, while it had been completely removed from the “emptyDir” volume. The backdoor was on the persistent volume was accessible once again when the pod was redeployed on the compromised node.

This result means that while using non-persistent storage, an attacker would have to re-create their backdoor each time the pod is deleted and recreated. However, if an attacker were to use persistent storage, they would be able to retain their backdoor even after the pod was deleted and recreated by the MTD algorithm.

# 7

# Discussion and Evaluation

This chapter will discuss and evaluate the experiment of this thesis which was to investigate whether the moving target defense is an effective defense strategy against attackers as well as performant enough to be used in production environments. With that said, the developed product is a prototype and is not meant to be used in production environments as is. To further enhance the prototype and possibly make it stable enough for production, a future work section is included at the end of this chapter.

## 7.1 Performance Evaluation

The goal of the experiment was to mainly develop and test the "moving target defense" to see if the system could achieve a positive security benefit, and secondary to achieve high availability and performance to evaluate whether it could be used in a production environment.

The performance tests conducted in the previous chapter show that the average CPU and memory usage is increased significantly while the MTD algorithm is running compared to running a static pod on a single node. This is due to the fact that the system is constantly creating and deleting pods, and thus using more resources. For example, measuring the CPU's average usage over a longer time period increases the average compared to measuring the CPU usage over a shorter time period. This is because of how average CPU usage is measured. The average is calculated by summing up the usage over a certain time period and dividing it by the number of time periods. This means that when the system is running at 100% CPU usage for half of the time, and 0% CPU usage for the other half, the average will be 50%. However, if we were to measure the average over a shorter time frame, the average would be closer to 100% during its peaks and closer to 0% during its lows. So during a more careful inspection with a shorter scraping of the average CPU usage (1 second intervals), we can clearly see that the peaks are at 100% while the system is creating and deleting pods. However, immediately once the pod is ready and finished swapping by the MTD system, the usage goes down to normal levels (same as a pod without the MTD). This means that the CPU and RAM are only being taxed when the system is actively creating and deleting pods, hence the active pod within the MTD system does not perform worse than an identical pod outside of the MTD system. As a result, the usage graphs will misrepresent depending on what time frame they are averaged over, so it is important to be aware of that when analyzing the usage.

## 7.2 Availability Evaluation

The fact that the moving target defense algorithm is constantly running and swapping pods has an effect on availability. If we would want to use the system in production, we would need to have a way of ensuring high availability. In other words, the system should be able to continue running and providing service even if one or more of its components fail. It should also not drop any requests during a node swap.

We have developed two different algorithms to accomplish the same idea, swapping nodes. However, they go about it slightly differently. The reason for this is that the first version goes under the assumption that the integrated load balancer in Kubernetes (NodePort) will automatically detect when a pod is taken off a node and redirect the traffic to the new node. This in fact worked quite well during the initial testing. However, with more thorough testing and much faster requests, we were able to see that there were some issues with this approach. Namely, the fact that the traffic was not being properly redirected to the new node when a pod was taken off of a node. This resulted in some requests timing out, as they were being sent to a node where no pod existed. We solved this problem by creating a second version of the algorithm that controls node swapping more granularly, rather than letting Kubernetes handle the swap automatically. We could then hold off the pod deletion until the load balancer has caught up with the change, thus the old pod will still continue to serve requests for a short period after the new pod is ready (about 1 second). This basically eliminated any request timeouts, however, there were some discrepancies. For example, once the load balancer detects that both nodes are available. There is a tiny time window where it randomly redirects requests between the two active pods (about 100ms) and then yet another small time window before it detects that the old node has gone offline. This resulted in much fewer packets getting dropped, but due to the reliability in TCP and how applications implemented HTTP (automatically retrying), we could still see that all requests got through. So even though we did not see any timeouts with this new method, there were still some inconsistencies in the request handling.

From these availability evaluations, we can see that the MTD system does not have any significant negative effects on availability compared to running a static pod on a single node. In fact, it might even improve availability in some cases, as crash handling could easily be implemented and prevent a single point of failure.

## 7.3 Security Evaluation

Test 1 from section 6.2 shows that the system does provide an improvement in security against physical tampering since the pods are only running for a limited time, thus hardening the attack surface. Test 2 shows that the system could partially handle physical destruction, in the event of a node going offline before the pod is deployed to it. In the event that the node the active pod is running on goes offline, the system in its current state was unable to relocate the pod. However, with a minor improvement to the algorithm, that problem would be solved, and thus the system would only suffer a couple of seconds of downtime before it comes back online.

The security benefits of using a moving target defense system are numerous. The fact that the pods are constantly being created and deleted makes it harder for an attacker to gain access to the system. Even if an attacker is able to compromise a pod, the pod will only be active for a short period of time before it is taken offline and replaced by a new pod. This makes it much harder for an attacker to maintain access to the system and also limits the amount of damage that can be done, as seen in Test 5 where the backdoor is removed upon every redeployment, given that the storage is not persistent. We can also see this in Test 3, that ongoing attacks such as Heartbleed can be cut off during a node swap. In addition, we can see in Test 4 that fingerprinters and vulnerability scanners may get different results depending on the timing. If the scanner is able to fingerprint a system before it is swapped, it may get different results than if it were to fingerprint the system after the swap. This would make it more difficult for an attacker to gain information about the system and fingerprint it correctly.

The overall effectiveness of the moving target defense is increased if the attacker only has access to a specific node or a limited set of nodes rather than accessing it via a load balancer. The more nodes that are in the system, the harder it is for an attacker to keep track of which pod is running on which node. The more different configurations/deployments and software stacks that are running in the MTD also increase the difficulty to time an attack or a vulnerability scan.

For example, there may be applications that are not accessible via a load balancer and/or has no public endpoints, and whose sole job is to report to an instance with information such as sensor data. Such an application will benefit much more from using our moving target defense system than for example a website with a public load balancer as an endpoint that any adversary could access. On the other hand, if the goal of an attacker is to attack a specific node, and the attacker is going via the load balancer, then the attacks will be redirected to different nodes which defeats the purpose of attacking a specific node.

The security benefits of a moving target defense are not a silver bullet or a one size fits all solution. It depends on the environment in which it is used, what type of system it is used on, and what the goals of the attacker are. In conclusion, we found that generally, the system works well against attacks that take a longer time to complete than the amount of time it takes for the pod to get replaced on a different node.

For example, if an attacker did successfully time fingerprinting and a vulnerability scan on a vulnerable pod before it is replaced, as well as successfully exploiting it in time. The attacker would only have a few seconds to collect sensitive data before the connection is cut off since the pod is getting replaced constantly. The attacker would then have to wait until the next opportunity to perform the attack again and gradually collect the sensitive data. This could potentially slow down an ongoing attack so that an intrusion detection system may alert the responsible party in time for a manual response to the attack.

## 7.4 Application Constraints

Our target moving defense was designed with stateless applications in mind. Stateless applications are those where the data is not stored on the server. This type of application can be easily replaced with a new one without any data loss. However, stateful applications store data locally on the server. This type of application would not be able to take advantage of our system since the data would be lost every time the pod is replaced. Statefulness could however be achieved by storing the state externally, outside the cluster, for example in a database.

Additionally, our system is not well suited for applications that need to maintain a connection for an extended period of time. For example, if an application needed to keep a video stream open for an hour, the stream would get interrupted since the pod would likely be redeployed at least once during that time frame. The application would need to be able to reconnect to the new pod after it is deployed. Such a use case could be solved with for example a buffer so that the playback is not interrupted during the swap. Thus, handling swaps in such use cases must be implemented in the application.

Container images are downloaded from a container registry when a pod is deployed for the first time. The image is then cached on the node. When the pod is swapped, the pod will be deployed on a different node that may not have the cached image. The image would then need to be downloaded again from the registry, adding some delay to the swapping process. This however only happens the first time a new pod is deployed on each node. On consecutive deployments, the cached image is used instead. The result of this is that the very first iteration of the moving target defense is slowed down and limited by the network bandwidth between the nodes and the container registry which under our conditions typically added about a 20-30 seconds delay depending on the image size. Once the image was cached on all nodes, this delay was removed. This may bring a concern about whether the cached image has been tampered with. A solution to that is to set the “imagePullPolicy” to “Always” which will compare the cached image digest to the digest in the repository. Kubernetes then redownloads the image if the repository digest is not present in the cache. However, for that to work, the repository must always be available. If the system were to be used on a network with limited bandwidth, the nodes could be prepared with cached images beforehand to avoid the above-mentioned initial delay.

Clients that require fast responses from the server should not be impacted by the swapping procedure since the clients are automatically redirected to the new pod by a load balancer. Especially if our suggestions mentioned in 7.6 Future Work, about improved management of the load balancer are implemented.

## 7.5 Ethics and Sustainability

From an ethics perspective, there are not any concerns that using a moving target defense creates. It is a way to make it more difficult for an attacker to successfully attack a system and does not have any negative consequences for the user. However, the information we have provided about the moving target defense system may help enable a malicious actor to understand the system and thus provide them with information on how to circumvent it. For example, if the defense is unknown to a malicious actor, they may be confused as to why the system is changing properties, but reading this thesis will help them understand what is going on.

The methods used to analyze the security in this thesis such as The Heartbleed Bug and vulnerability scanners could be used by malicious actors to attack real targets. However, the methods we used have already been publicly known and patched for several years so there should be no negative impact.

From a sustainability perspective, the moving target defense system is using more energy and resources than a traditional system. The main use of energy is from the nodes that are used to constantly create new pods and replicas. The system also uses more resources in terms of storage and network bandwidth as it needs to constantly pull down images and updates for the pods. There are also additional nodes that need to stay on standby even when they are not used, thus using energy while idling.

## 7.6 Future Work

There are several ways in which the moving target defense system could be improved. The most obvious and immediate improvement would be to implement crash detection on a currently running pod and apply an active label to a different node so that Kubernetes can handle the rescheduling of the crashed pod properly.

Another improvement is to include management of the load balancer inside the MTD algorithm, such that it can redirect the traffic and ensure zero lost packets. As seen in our tests and section 7.2, we have achieved zero dropped requests but there may still be individual TCP packets that are dropped during a node swap on rare occasions, leaving the implementation on the requester's side to solve it by retrying to send a request. The improvement we suggest by controlling the load balancer involves redirecting the traffic to the newly deployed pod only once it has been confirmed as ready, instead of randomly sending packets between the old and the new pod for a brief moment.

One way to improve further is to make the system more adaptive so that it can automatically detect when an attack is happening and respond accordingly. For example, if the system notices that a lot of packets are being sent to a specific node, it could automatically add more nodes to the pool or redirect traffic to a different node. Similarly, the system could be scaling up the number of replicas when there is high legitimate traffic to the system. For example, the system could use a load balancer to detect when there is high traffic and only then create more replicas. This would save on resources when the system is not under a heavy load.

Another way to improve the system is to make it more scalable so that it can be used on larger systems. For example, the system could achieve high availability by scheduling multiple replicas of a pod on different nodes. This would ensure that even if one node is taken down, there are still other replicas that can handle the traffic with less downtime.

Finally, the system could use more rigorous testing to ensure that it is effective in practice. For example, the system could be tested against more sophisticated attacks such as DDoS attacks. Additionally, the system should be tested on a variety of different systems to ensure compatibility, such as edge devices with limited resources. As well as long-term tests to ensure that there are no performance degradation over time or special cases of bugs that have not yet been discovered.

Moving target defense is a promising area of research that has the potential to improve the security of systems. The methods used in this thesis are just a few of the many ways in which the system can be improved. With further research and development, the moving target defense system has the potential to become a key component in securing systems against attacks.

# 8

# Conclusion

The goal of the thesis was to explore the moving target defense concept and implement it on a real system to see if it is viable in practice in terms of performance and availability, as well as to see if it has any real security benefits. We started by surveying the literature to get an overview of the current state of the art in moving target defense. We then looked at different ways of implementing moving target defense that had not already been done and selected Kubernetes as our platform. We implemented a prototype system on top of Kubernetes and ran it on a testbed of ten nodes. We evaluated the system in terms of performance, availability, and security and found that the system was able to provide an increased level of security with only a small impact on performance and availability. In conclusion, we believe that moving target defense is a viable approach to securing systems against attacks and that it has the potential to be further developed into a key component of securing systems in the future.

# Bibliography

[1] The Kubernetes Authors. (2022). “Deployments | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 05/03/2022).

[2] 451 Research, “Digital Transformation Opportunity for Service Providers: Beyond Infrastructure,” in *2017 Microsoft Cloud and Hosting Summit*, Microsoft, 2017, p. 22. [Online]. Available: https://download.microsoft.com/download/E/0/A/E0A16C07-CA08-4880-B434-5BC6F9B2F193/MCHS2017_MELANIE%20POSEY.pdf (visited on 05/05/2022).

[3] IDC (International Data Corporation). (2021). “Cloud Infrastructure Spending Increased in Third Quarter of 2021 with Overall Growth Expected for 2021, According to IDC,” [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS48776122 (visited on 05/06/2022).

[4] MarketsandMarkets. (2021). “Cloud Computing Market Size, Share and Global Market Forecast to 2026 | COVID-19 Impact Analysis | MarketsandMarkets,” [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html (visited on 05/06/2022).

[5] AppsRunTheWorld. (2022). “Top 10 Cloud Software Vendors, Market Size and Forecast 2020-2025,” [Online]. Available: https://www.appsruntheworld.com/top-10-cloud-software-vendors-market-size-and-market-forecast/ (visited on 05/06/2022).

[6] Juniper Research. (2014). “Cloud Services to be Adopted by 3.6Bn Consumers Globally by 2018, Juniper Research Finds,” [Online]. Available: https://www.juniperresearch.com/press/cloud-services-adopted-by-3bn-consumers-2018.

[7] Transforma Insights. (2020). “Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030, by vertical (in millions),” Statista, [Online]. Available: https://www.statista.com/statistics/1194682/iot-connected-devices-vertically/ (visited on 05/06/2022).

[8] A. Yadav. (2020). "Network design: Firewall, IDS/IPS," Infosec Institute, [Online]. Available: https://resources.infosecinstitute.com/topic/network-design-firewall-idsips/ (visited on 08/13/2022).

[9] G. Sharma, S. Bala, A. K. Verma, and T. Singh, "Security in Wireless Sensor Networks using Frequency Hopping," *International Journal of Computer Applications*, vol. 12, no. 6, Dec. 2010, ISSN: 09758887. DOI: 10.5120/1686-2247.

[10] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, Jan. 2011, ISSN: 0146-4833. DOI: 10.1145/1925861.1925869.

[11] The Kubernetes Authors. (2022). "Production-Grade Container Orchestration | Kubernetes," [Online]. Available: https://kubernetes.io/ (visited on 04/10/2022).

[12] Docker, Inc. (2022). "Get Started with Docker," [Online]. Available: https://www.docker.com/ (visited on 04/10/2022).

[13] K. Harrison and S. Xu, "Protecting Cryptographic Keys from Memory Disclosure Attacks," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, IEEE, Jun. 2007, pp. 137–143, ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.77.

[14] M. Hemmati and M. Ali Hadavi, "Bypassing Web Application Firewalls Using Deep Reinforcement Learning," *The ISC International Journal of Information Security*, vol. 14, no. 2, pp. 131–145, 2022. DOI: 10.22042/isecure.2022.323140.744.

[15] N. Manohar, "A Survey of Virtualization Techniques in Cloud Computing," in *Proceedings of International Conference on VLSI, Communication, Advanced Devices, Signals & Systems and Networking (VCASAN-2013)*, 2013, pp. 461–470. DOI: 10.1007/978-81-322-1524-0_54.

[16] Amazon. (2022). "Amazon EC2 FAQs," [Online]. Available: https://aws.amazon.com/ec2/faqs/ (visited on 07/14/2022).

[17] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005, ISSN: 00189162. DOI: 10.1109/MC.2005.173.

[18] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, pp. 24–31, 3 May 2015, ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51.

[19] The Containerd Authors. (2022). "An industry-standard container runtime with an emphasis on simplicity, robustness and portability | ContainerD," [Online]. Available: https://containerd.io/ (visited on 05/12/2022).

[20] CRI-O. (2022). "Lightweight Container Runtime for Kubernetes | CRI-O," [Online]. Available: `https://cri-o.io/` (visited on 08/13/2022).

[21] OCI. (2022). "About the Open Container Initiative | OCI," [Online]. Available: `https://opencontainers.org/about/overview/` (visited on 08/13/2022).

[22] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, pp. 42–48, 5 Sep. 2017, ISSN: 2325-6095. DOI: `10.1109/MCC.2017.4250933`.

[23] Docker, Inc. (2022). "Swarm mode overview | Docker," [Online]. Available: `https://docs.docker.com/engine/swarm/` (visited on 09/04/2022).

[24] Red Hat, Inc. (2022). "Red Hat OpenShift | Red Hat," [Online]. Available: `https://www.redhat.com/en/technologies/cloud-computing/openshift` (visited on 09/04/2022).

[25] The Apache Software Foundation. (2022). "Program against your datacenter like it's a single pool of resources | Apache Mesos," [Online]. Available: `https://mesos.apache.org/` (visited on 09/04/2022).

[26] The Kubernetes Authors. (2022). "Welcome! | minikube," [Online]. Available: `https://minikube.sigs.k8s.io/docs/` (visited on 04/28/2022).

[27] A. P. Ferreira and R. Sinnott, "A performance evaluation of containers running on managed kubernetes services," IEEE, Dec. 2019, pp. 199–208, ISBN: 978-1-7281-5011-6. DOI: `10.1109/CloudCom.2019.00038`.

[28] Stackpath. (2022). "What is YAML?" [Online]. Available: `https://www.stackpath.com/edge-academy/what-is-yaml/` (visited on 06/14/2022).

[29] The Kubernetes Authors. (2022). "Configuration Best Practices | Kubernetes," [Online]. Available: `https://kubernetes.io/docs/concepts/configuration/overview/` (visited on 09/18/2022).

[30] ——, (2022). "Client Libraries | Kubernetes," [Online]. Available: `https://kubernetes.io/docs/reference/using-api/client-libraries/` (visited on 08/13/2022).

[31] V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, pp. 28–39, 3 1999, ISSN: 10897801. DOI: `10.1109/4236.769420`.

[32] W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, and R. Tober, "Integration of Static and Dynamic Code Analysis for Understanding Legacy Source Code," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Oct. 2016, pp. 543–552, ISBN: 978-1-5090-3806-0. DOI: `10.1109/ICSME.2016.70`.

[33] The Editors of Encyclopaedia Britannica. (2022). “Firewall,” Encyclopedia Britannica, [Online]. Available: `https://www.britannica.com/technology/firewall` (visited on 06/03/2022).

[34] M. Gouda and A. Liu, “A Model of Stateful Firewalls and Its Properties,” in *2005 International Conference on Dependable Systems and Networks (DSN’05)*, IEEE, pp. 128–137, ISBN: 0-7695-2282-3. DOI: `10.1109/DSN.2005.9`.

[35] V. Clincy and H. Shahriar, “Web Application Firewall: Network Security Models and Configuration,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, Jul. 2018, pp. 835–836, ISBN: 978-1-5386-2666-5. DOI: `10.1109/COMPSAC.2018.00144`.

[36] A. Fuchsberger, “Intrusion Detection Systems and Intrusion Prevention Systems,” *Information Security Technical Report*, vol. 10, no. 3, pp. 134–139, Jan. 2005, ISSN: 13634127. DOI: `10.1016/j.istr.2005.08.001`.

[37] J. P. S. Medeiros, J. B. B. Neto, A. M. B. Júnior, and P. S. M. Pires, “Learning Remote Computer Fingerprinting,” in, 2014, pp. 253–283. DOI: `10.1007/978-3-319-05885-6_12`.

[38] N. Schagen, K. Koning, H. Bos, and C. Giuffrida, “Towards Automated Vulnerability Scanning of Network Servers,” in *Proceedings of the 11th European Workshop on Systems Security*, New York, NY, USA: ACM, Apr. 2018, pp. 1–6, ISBN: 9781450356527. DOI: `10.1145/3193111.3193116`.

[39] S. Rahalkar, “OpenVAS,” in *Quick Start Guide to Penetration Testing*, Berkeley, CA: Apress, 2019, pp. 47–71. DOI: `10.1007/978-1-4842-4270-4_2`.

[40] Greenbone Networks GmbH. (2022). “OpenVAS - Open Vulnerability Assessment Scanner,” [Online]. Available: `https://www.openvas.org/` (visited on 06/15/2022).

[41] G. Lyon. (2022). “Nmap: the Network Mapper,” [Online]. Available: `https://nmap.org/` (visited on 06/15/2022).

[42] The MITRE Corporation. (2022). “Overview - About the CVE Program,” [Online]. Available: `https://www.cve.org/About/Overview` (visited on 06/15/2022).

[43] Red Hat Inc. (2020). “NVD - CVE-2014-0160,” NIST, [Online]. Available: `https://nvd.nist.gov/vuln/detail/CVE-2014-0160` (visited on 06/16/2022).

[44] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, “The matter of heartbleed,” ACM, Nov. 2014, pp. 475–488, ISBN: 9781450332132. DOI: `10.1145/2663716.2663755`.

[45] Rapid7. (2022). “Metasploit - The world’s most used penetration testing framework,” [Online]. Available: `https://www.metasploit.com/` (visited on 06/14/2022).

[46] A. Kurniawan, B. S. Abbas, A. Trisetyarso, and S. M. Isa, “Classification of Web Backdoor Malware Based on Function Call Exectuion of Static Analysis,” *ICIC Express Letters*, vol. 13, no. 6, pp. 445–452, DOI: `10.24507/icicel.13.06.445`.

[47] A. Roy, A. Chhabra, C. A. Kamhoua, and P. Mohapatra, “A moving target defense against adversarial machine learning,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ACM, 2019, pp. 383–388, ISBN: 9781450367332. DOI: `10.1145/3318216.3363338`.

[48] K. Zeitz, M. Cantrell, R. Marchany, and J. Tront, “Designing a Micro-Moving Target IPv6 Defense for the Internet of Things,” in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, ACM, 2017, pp. 179–184, ISBN: 9781450349666. DOI: `10.1145/3054977.3054997`.

[49] B. Thébaudeau. (2019). “An Introduction to Cooja,” [Online]. Available: `https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja` (visited on 05/13/2022).

[50] N. O. Ahmed and B. Bhargava, “Mayflies,” in *Proceedings of the 2016 ACM Workshop on Moving Target Defense - MTD’16*, ACM Press, 2016, pp. 59–64, ISBN: 9781450345705. DOI: `10.1145/2995272.2995283`.

[51] E. Al-Shaer, Q. Duan, and J. H. Jafarian, “Random Host Mutation for Moving Target Defense,” in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, 2013, pp. 310–327, ISBN: 9783642368820. DOI: `10.1007/978-3-642-36883-7_19`.

[52] T. E. Carroll, M. Crouse, E. W. Fulp, and K. S. Berenhaut, “Analysis of network address shuffling as a moving target defense,” in *2014 IEEE International Conference on Communications (ICC)*, IEEE, 2014, pp. 701–706, ISBN: 978-1-4799-2003-7. DOI: `10.1109/ICC.2014.6883401`.

[53] J. Tian, R. Tan, X. Guan, Z. Xu, and T. Liu, “Moving target defense approach to detecting stuxnet-like attacks,” 1, vol. 11, 2020, pp. 291–300. DOI: `10.1109/TSG.2019.2921245`.

[54] K. A. Torkura, M. I. Sukmana, A. V. Kayem, F. Cheng, and C. Meinel, “A cyber risk based moving target defense mechanism for microservice architectures,” in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 2018, pp. 932–939. DOI: `10.1109/BDCloud.2018.00137`.

[55] J. Tanner. (2017). “Polyverse Announces Moving Target Defense Suite for Container Environments,” [Online]. Available: `https://www.businesswire.`

com/news/home/20170628005638/en/Polyverse-Announces-Moving-Target-Defense-Suite-for-Container-Environments (visited on 05/13/2022).

[56] K. Feng, X. Gu, W. Peng, and D. Yang, “Moving target defense in preventing sql injection,” in *Artificial Intelligence and Security. ICAIS 2019*, X. Sun, Z. Pan, and E. Bertino, Eds., Cham: Springer International Publishing, 2019, pp. 25–34, ISBN: 978-3-030-24268-8. DOI: 10.1007/978-3-030-24268-8_3.

[57] D. Torrieri, *Principles of Spread-Spectrum Communication Systems*, 4th ed. Cham: Springer International Publishing, 2018, ISBN: 978-3-319-70568-2. DOI: 10.1007/978-3-319-70569-9.

[58] J. Xu, P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu, “Comparing different moving target defense techniques,” ACM Press, 2014, pp. 97–107, ISBN: 9781450331500. DOI: 10.1145/2663474.2663486.

[59] The Kubernetes Authors. (2022). “Container Runtimes | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/ (visited on 05/03/2022).

[60] ——, (2022). “Installing kubeadm | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/ (visited on 05/03/2022).

[61] ——, (2022). “Createting a cluster with kubeadm | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ (visited on 05/03/2022).

[62] MDN contributors. (2022). “MVC | mdn web docs,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/MVC (visited on 08/13/2022).

# A

# Appendix 1

The entire implementation and source-code is available on our GitHub repository under the GPLv3 license.

**GitHub repository**

`https://github.com/ptibom/Moving-Target-Defense-with-Kubernetes`