Beyond the First Layers of Cloud Defenses

An Ensemble Solution for Improving Container Runtime Security Using Metrics

Master Thesis

Group ds102f21

Aalborg University
Technical faculty of IT and Design

The Technical Faculty of IT and Design
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Beyond the First Layers of Cloud Defenses - An Ensemble Solution for Improving Container Runtime Security Using Metrics

Theme:
10th Semester project - Master Thesis

Project Period:
Spring Semester 2021

Project Group:
ds102f21

Participant(s):
Domantas Astrauskas
Fruszina Vivienne Spence

Supervisor(s):
Danny Bøgsted Poulsen
René Rydhof Hansen

Copies: 1

Page Numbers: 74

Date of Completion:
June 10, 2021

Abstract:

An important topic in cloud computing is security. Security in the cloud is more akin to a journey rather than a destination. Securing a cloud platform is a complex task that has to be implemented on multiple layers. One of these layers is the container layer. This thesis focuses on the container layer. It concentrates on container security during run time. The intention of the thesis is to improve on the results of an open-source runtime security tool, Falco. Falco's biggest weakens is the amount of alerts it sends out. In many cases an actual attack alert can be buried by the number of alerts Falco sends out if their priority level is low. By implementing a tool that uses simple algorithms to detect malicious behaviour in the containers we aim at improving the priority level of those Falco alerts that have an underlying attack as source. The selected algorithms look at container metrics, such as CPU and memory usage and identify outliers in their usage attempting to pinpoint when an attack is happening. If the algorithms detect an attack at the same time as Falco does the priority level of the Falco alerts is increased thus giving the alert more significance.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

List of Figures

List of Tables

Preface

This thesis was created at Aalborg University under the CS-IT programme in collaboration with Keysight Technologies. It is a 10th semester project, the Master's Thesis. The report was done by ds102f21.

The project group is thankful to each of the members for their contribution during the semester. Also a heartfelt thank you to our supervisors from AAU Danny Bøgsted Poulsen and René Rydhof Hansen and our contacts from the company Andrea Cattoni, Lars Mikkelsen and Dragos Brezoi.

The project is divided into chapters: Chapter 1 introduces the problem area and defines the underlying hypothesis, Chapter 2 discuss related works, Chapter 3 presents theoretical considerations about the cloud computing paradigm, Chapter 4 talks about containers and virtual machines, Chapter 5 presents the current issues with container security, Chapter 6 discusses the experiments presenting their setup, the necessary tools and describes these, Chapter 7 presents the results of the experiments, Chapter 8 discusses possible directions where future research could go and finally Chapter 9 concludes the thesis.

Aalborg University, June 10, 2021

Domantas Astrauskas
<dastra19@student.aau.dk>

Fruzsina Vivienne Spence
<fbenke11@student.aau.dk>

Chapter 1

Introduction

The popularity of cloud computing is on the rise since the 1990s. With the release of the Elastic Compute cloud (EC2) in 2006 by Amazon a new era of cloud computing started and its popularity experienced exponential growth. An ever increasing number of companies are moving their current software over to the cloud and start developing cloud native applications. The popularity of cloud computing is not surprising as it promises low maintenance costs, accessibility, scalability, mobility, unlimited storage capacity, back-up and restore data, automatic software integration and mobility among others. The cloud is not without fault however and the biggest concern, to this day, being security [41, 9].

Securing a cloud environment is challenging and it has to be a continuous effort rather than a one-time occurrence. Security in general, and cloud security in particular, is a journey not a destination. In the cloud, security has to be assured on multiple layers. More often than not, cloud security tactics use a defense-in-depth strategy, meaning that there are multiple security measures in place on different levels [26]. One of these level is the container level. This is the level that is the focus of this thesis.

In 2013, the emergence of Docker made the popularity of containers increase exponentially. Containers are a standard unit of software that bundle applications, their dependencies and configuration in a systematic manner so that the application can run quickly and reliably indifferent from the computing environment. In cloud computing, containers have initially emerged as lightweight versions for virtual machines [30, 38]. They facilitate isolation of resource procedures and allow users to work with applications in this manner. In the cloud containers are used to build blocks that in turn create operational efficiency, version control and abstract away from the underlying environment [38].

As security in the cloud is still a major concern, using containers in the cloud adds an extra layer that needs to be taken into consideration when securing it. In many cases container security is exhausted at the image, host and OS levels. Scanning container images before pushing them into a registry and digitally signing them at build time are good practices, they are just not enough. Problems can emerge at runtime and suddenly an exposed process can appear behind the defence lines [13]. Runtime container security requires the analysis of all activities within a container application environment. This entails keeping an eye on all container and host activities and monitoring the protocols and payloads of network connections. Given the dynamic nature of container environments traditional security practices, like hardening attack surfaces or vulnerability scanning, prove insufficient in providing a complete runtime protection [8].

In this thesis we focus on the container level, more precisely we concentrate on container runtime security in the cloud. The goal is to create proof of concept for a tool that applies different anomaly detection algorithms to container metrics such as CPU and memory usage, which then can be used together with a popular open-source runtime security tool to improve the results produced by this.

1.1 Problem Area

As previously stated, security measures must be implemented on multiple levels in the cloud. An overview of the different levels can be seen below:

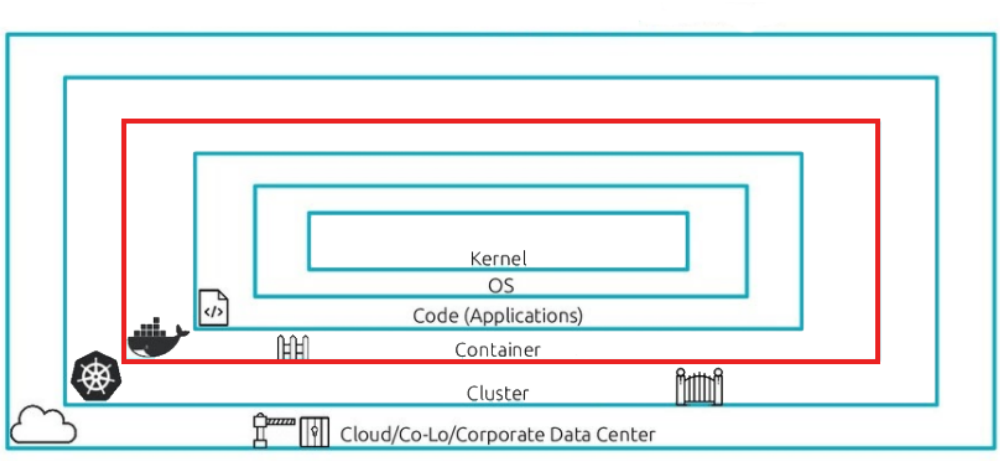

Figure 1.1: Layers of a cloud infrastructure - based on: [10]

In a previous research [3] we investigated security in the cloud on the application level, thus a natural next step was to continue our research by looking at the next layer, the container layer. This layer has a two-fold security requirement to make sure that everything is as safe as it possibly can be. On one hand, the container images need to be scanned and signed before deployment, and on the other, security measures have to be taken to assure their security during runtime. To further narrow down the scope of this thesis we chose to look at container security during runtime.

There are multiple commercial and open-source tools available that provide runtime cloud security. In this thesis we are working with one such tool, called Falco. Falco is an open-source cloud-native runtime security project designed specifically for detecting threats in a Kubernetes cluster. It is capable of detecting unexpected application behaviour and alerts on threats at runtime [37].

While running some initial test on a private cloud with Falco we noticed that it sends a lot of alerts, sometimes a couple every second, depending on the size of the cluster and the number of active containers in it. These alerts have different priority levels ranging from debug to emergency. During this initial experiment phase we simulated a couple of attacks and observed the incoming alerts. What we noticed was that the alerts that came through had a very low priority level, *notice*, which is the third lowest priority. Since notice level alerts can be pretty common in a cloud with multiple deployed containers these might get ignored by system admins and allow malicious users to exploit this.

This got us thinking about a possible way of improving threat detection during runtime that involves container metrics. The underlying idea was to monitor container metrics, such as CPU and memory usage during runtime, and see if we can detect some anomalies around the same time frame as the alerts from Falco signalled. If there are anomalies in the container metrics the priority level of the Falco alert will be raised to a higher priority level, drawing the attention of a system admin to take action.

We implemented the **Container Overseer**, a tool that receives Falco alerts via a messaging system and also monitors container metrics at runtime. It uses different anomaly detection algorithms to monitor metrics and flags any abnormal behaviour. Our hope is that the Container Overseer combined with Falco can become an ensemble solution that has the capability of improving runtime container security.

1.1.1 Problem Statement

Based on the information above the underlying hypothesis of this thesis follows:

> **The runtime security of cloud based containers in a private cloud can be improved by combining container metrics monitoring using simple anomaly detection algorithms with an open-source runtime security tool.**

In order to prove the above hypothesis, this thesis dives into related topics; integrates Falco, the runtime security tool to an existing cloud infrastructure; implements and deploys the Container Overseer, on the same cloud, that is capable of intercepting alerts from Falco, monitor container metrics and detect anomalies in these, and increase the priority level of the Falco alerts before they are sent if relevant. The goal is to improve the results of Falco by looking at anomalies in the container metrics.

The main focus areas of this thesis are:

- Cloud computing and containers.
- Container runtime security and its importance in the cloud.
- Set up and use of a runtime container security tool in a private cloud.
- Implementation and deployment of a tool to a private cloud, that can analyze container metrics and detect anomalies in these, using different anomaly detection algorithms.
- Investigation of the results of the runtime container security tool, Falco, and the metrics analyzing tool, Container Overseer, to improve the results from Falco.

Chapter 2

Related Work

In order to prove the hypothesis presented in the previous chapter the first step was to take a thorough look at the literature about cloud and container security and solutions. Furthermore, an investigation into anomaly detection algorithms was also required so the literature regarding these was also closely examined.

Cloud and container security are topics of interest in research circles because of the increasing popularity of the technologies. However, market research shows that cloud and container security are some of the main concerns when it comes to actually using them [36]. This is why there are a number of research papers that focus on surveying the literature regarding cloud and container security and currently available solutions.

A literature survey by Ali et al. presents the opportunities and challenges of cloud computing [1]. The writers present a detailed analyses of security issues arisen from the very nature of cloud computing and the available solutions. They also touch upon security vulnerabilities in mobile cloud computing. The findings of this research show that cloud computing has certain security issues that carry over from traditional computing but it has its own unique challenges too. These unique challenges come from virtualization and multi-tenancy, where different users have access to the same resources. The research also points out the various legal challenges that can accompany the geographically distributed nature of cloud computing. As far as solutions go, the paper divides these in two categories: counter measures for communication and counter measures for architectural issues.

In their paper with the title *Container Security: Issues, Challenges and the Road Ahead*, Sultan et al. [36] presents a comprehensive overview of the current state of container security and solutions. They distinguish four generalised use cases that potentially

cover all security requirements within the host-container threat landscape: protecting a container from applications inside it, inter-container protection, protecting the host from containers, and protecting containers from a malicious or semi-honest host. They found that in the case of the first three use cases the security solutions are software based while in the last use case the solutions are hardware based.

When talking about anomaly detection in the cloud many research papers use machine learning techniques. One such paper written by Islam and Miranskyy [18] talk about anomaly detection in cloud components using machine learning. They look at resource utilization and metrics to detect anomalies in various multi-dimensional time series. They observe that while data centers have been deploying advanced monitoring systems to oversee the health of cloud components these systems rely on statistics and heuristics based resource utilization thresholds. Given the nature of cloud applications this approach is not effective enough. The authors use machine learning techniques and tools to find anomalies in the multidimensional cloud resource utilization time-series data. They created a dimension-independent neural network model to detect outliers in components of cloud platforms [18].

Our belief is that anomaly detection in cloud components does not have to use machine learning tools and techniques in order to provide good results. In this thesis we are exploring alternative ways for anomaly detection that can improve the rate of detecting outliers in container metrics.

Another paper that focuses on anomaly detection in the cloud centers around Docker containers and attempts to create a Docker container anomaly monitoring system based on optimized isolation forest [42]. In this paper, Zou et al. proposes an online container anomaly detection system by monitoring and analysing multidimensional resource metrics of the containers based on the optimised isolation forest algorithm. Their system is capable of identifying abnormal resource metrics and it can automatically adjust the monitoring period to reduce delays and system overhead. It can also locate the cause of the anomalies by analyzing container logs. The authors found that such a system is very effective in both simulated and real cloud environments [42]. This paper inspired our approach in this thesis and it prompted us to further investigate anomaly detection algorithms that could be used in a cloud settings.

An additional paper that focuses on anomaly detection algorithms is *A Comparative Evaluation of Unsupervised Multivariate Data* [11]. In this paper Goldstein and Uchida evaluate 19 different unsupervised anomaly detection algorithms on 10 different datasets from multiple application domain. The research inspects the anomaly detection performance, computational effort, the impact of parameter settings and outlines the global/local anomaly detection behaviour. They find that nearest neigh-

bour based algorithms perform better then clustering algorithms but clustering algorithms have a lower computation time. The findings indicate that nearest neighbour based methods like k-NN should be used for global tasks and local outlier factor (LOF) for local tasks instead of clustering based methods [11]. Based on this paper we decided to include LOF as one of the algorithms in the Container Overseer.

Other researches apply statistical algorithms to detect anomalies. In their paper Siris and Papagalou [34] consider an adoptive threshold algorithm and a particular application of the cumulative sum (CUSUM) algorithm to detect the most common type of denial of service attack. They investigate detection probability, false alarm ratio and detection delay. The authors look into how are these metrics affected by the parameters of the algorithm and the characteristics of the attack with the goal of offering guidelines to effectively tune the parameters of the detection algorithm to achieve specific performance requirements. The findings show that the adoptive threshold algorithm produces satisfactory results for high intensity attacks but it does not perform well against low intensity attacks. CUSUM was found more effective across a range of different attacks [34].

The threshold algorithm, also known as Welford's algorithm is mentioned by Lobo in his article with the title *Detecting real-time and unsupervised anomalies in streaming data: a starting point*. He proposes some modifications to the algorithm by introducing an upper and lower limit. Another algorithm explored by Lobo in the same article is a quartiles-based solution. This algorithm can help identifying outliers by the way it represents the distribution of a datased based on a set of number summaries: the minimum, the first quartile, the median, the third quartile and the maximum [24]. All data point above the maximum or below he minimum are considered outliers. Lobo suggests using sliding window when implementing this algorithm for streaming data. In this thesis we work with both the threshold algorithm and the quartiles-based solution in the Container Overseer.

As it can be seen from this overview of the literature, cloud and container security are seen as important topics, however most of these researches look at cloud and container security from a static angle, identifying and attempting to mitigate issues before deployment. Few research considers streaming data. Runtime container security is still an area that could use additional research, thus we choose to focus on this. Some research suggests anomaly detection for improving cloud and container security. We tried to select those researches that do not use machine learning algorithms because we wanted to investigate how do simpler algorithms perform. The aforementioned papers and articles provided a great starting point for the research conducted in this thesis.

Chapter 3

Cloud Computing Paradigm

Chapter 1 introduced the main topic of this thesis, namely runtime cloud security on the container level. To better understand the underlying technology this chapter takes a closer look at cloud computing, introducing the necessary terminologies, talking about what is cloud computing and the different deployment and service models. Furthermore, the chapter discusses the top security threats of cloud computing and presents a brief overview of existing cloud security strategies.

3.1 Cloud Essentials

Cloud computing has been around since the 1960s but it has seen a great increase in popularity since 2006 when Amazon released its Elastic Compute cloud (EC2) as a commercial web service [3].

Cloud computing is an on-demand delivery of computing services over the internet. These services include servers, storage, databases, networking, software analytics and intelligence. Cloud computing offers faster innovation, scalable resources and economies of scale [27].

The strength of the cloud lays in virtualisation, it's ability to create virtual computing resources. This allows the use of different operating systems and applications on the same machine with the same hardware at the same time. Using virtualisation allows the cloud to offer low cost, hardware reducing and energy saving solutions. Moreover, virtualisation can turn one server into multiple servers and one data center into multiple data centers. It can be used for storage, memory and networking as well [3].

Cloud computing has multiple benefits, among which the following can be mentioned [27]:

- Cost - cloud computing reduces costs because users do not need to buy expensive hardware and software, do not have to be concerned about setting up their own data centers and do not have to worry about the accompanying costs.
- Speed - cloud services are on-demand and scaling up in resource usage is quick and simple.
- Global scale - cloud services allow for elastic scaling, providing the right amount of resources from the right geographic location.
- Productivity - since cloud providers handle a lot of the datacenter related work, IT teams can spend time on other, business critical tasks.
- Performance - using a cloud can greatly improve performance since cloud computing services are set up to run on a world wide network of source datacenters which provides reduced latency for application and greater economies of scale.
- Reliability - data can be mirrored so data backup, disaster recovery and business continuity is easier and less expensive to assure.

Many cloud providers mention security among the benefits of cloud computing because in essence clouds should be secure since the providers offer a broad set of policies, technologies and controls that strengthen the overall security posture. [27]. Security is however a controversial topic, and one of the main reasons why companies move so cautiously towards cloud adaptation [9]. Some other disadvantages of cloud computing include downtime, while rare it is always costly; vulnerability to attacks, because every component is online; limited control and flexibility; and vendor lock-in [23].

3.1.1 Cloud Deployment Models

A cloud deployment model, or cloud computing architecture, refers to the type of environment configuration based on proprietorship, size and access. It describes the way cloud services are deployed. There are three different ways of deploying cloud services: on a public cloud, on a private cloud or on a hybrid cloud. The three ways of deploying cloud services lead to four widespread cloud deployment

models: private cloud, public cloud, community cloud and hybrid cloud. These deployment models are defined by who controls the infrastructure and where is it located [3, 27].

A **private cloud** is not available for the general public as it usually belongs to a single company. The server of a private cloud can be hosted internally by the company or externally. The infrastructure of this type of cloud is maintained on a designated private network and the hardware and software are used only by the specific company. A private cloud provides opportunity for customisation of the infrastructure and tailor it to the company's needs. As such, the company has the possibility to safeguard mission critical operations and data. The benefits of this deployment model stem from its autonomy and it provides flexible development and high scalability besides the increased security, privacy and reliability. This deployment model can be expensive as the company has to secure the hardware, software and staff requirements itself [32].

A **public cloud** is available for the general public. Data is created and stored on third-party servers and the server infrastructure belongs to a service provider. This service provider manages the server and the resources and users do not have to worry about acquiring and maintaining hardware. This deployment model is usually offered both as free of charge and on a pay-as-you go basis. The privacy of data on this type of cloud is significantly lower than in the case of the private cloud. Public clouds provide hassle-free infrastructure management for the users, high scalability and availability and reduced costs in general. Some disadvantages of the public cloud are the reliability, data security and privacy and the lack of customization of service options [32].

The **community cloud** is very similar to a private cloud except instead of belonging to a single company it belongs to a group of companies that have similar backgrounds. These companies share the infrastructure and resources of the community cloud. The companies need to have uniform security, privacy and performance requirements. The costs of such a cloud are shared by all users thus it is cheaper than having a private cloud. Community clouds have improved security, privacy and reliability compared to a pubic cloud. Moreover, it makes it easy to share data and collaborate on projects. Community clouds have certain shortcomings that involve increased costs compared to the public deployment model, sharing of storage and bandwidth between multiple companies and it is not as widespread as the other models just yet so it is less mature [32].

The final deployment model is a **hybrid cloud**. This is a combination of the public and private clouds. This allows companies to combine the different aspects of the two models and create a deployment model that truly fits the company. In most

cases, companies identify their mission critical operations and data and secure these behind a private cloud while the rest of the operations and data can be stored on a public one. This allows companies to safeguard and control strategically important assets. Hybrid clouds provide increased security and privacy compared to the public cloud deployment model and has enhanced scalability and flexibility at a reasonable price. However, this deployment model only makes sense if the company is able to make the distinction between mission-critical an non-mission-critical data [3, 32].

3.1.2 Cloud Service Models

There are three basic types of cloud service models: infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS) and software-as-a-service (SaaS):

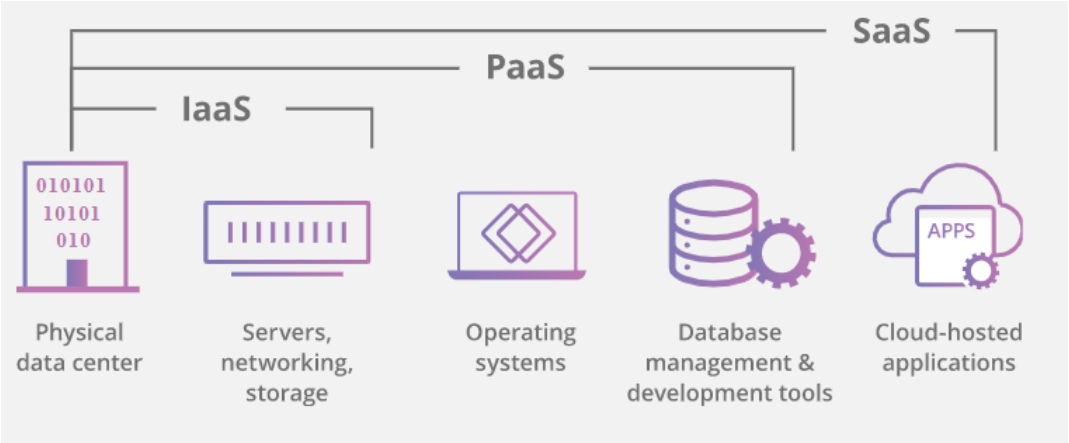

Figure 3.1: Cloud service models - source: [3]

As it can be seen form the figure above the three service models are overlapping and gradually cover more and more aspects of cloud computing. IaaS provides computing resources, such as physical data centers, servers, networking and storage. This model supplies the bare minimum allowing the users to use these as they see fit. This provides the biggest freedom for users among the three models, as they can use their own platforms and applications within the provided infrastructure. Moreover, the model is pay-as-you-go and has the possibility to scale based on the needs of the users. This leads to reduced up-front and maintenance costs and no single point of failure. Since the infrastructure is in the hands of the cloud provider there are some security risks involved together with limited customisation opportunities [3].

PaaS provides everything that IaaS does together with access to a cloud based environment. It is a cloud based runtime environment where users can develop, test, manage, build and deliver applications. It has a suite of pre-built development tools but provides the freedom for developers to customise and test their own applications too. Security, operating systems, servers and backups are all managed by the cloud provider in this model. PaaS is a good solutions where teams work on a project remotely. This model has reduced costs, improved time to market, continuous updates, scalability and freedom of actions. This model is however, vendor dependent and can have compatibility issues with existing infrastructures [3].

SaaS is a service model that provides everything that the previous two models do but in addition it also provides access to cloud hosted applications. This is the most comprehensive service model of the three, that makes it possible for the user to use applications remotely without the need of installing these locally. It uses a subscription model where the management of the software of the applications is handled by the provider. This service model has no initial set up and hardware costs, it has automatic upgrades, cross-device compatibility, scalability, ease of customisation and can be accessed from anywhere. However, SaaS provides the least amount of control among the three service models as most everything is handled by the service provider. It also has a limited range of solutions and dependency on connectivity to the internet [3].

3.2 Top Threats of Cloud Computing

The cloud being a complex technology with multiple layers that can be corrupted, it is no surprise that security in the cloud is taken seriously. In 2019 the Cloud Security Alliance has identified 11 threats, risks and vulnerabilities that are prominent in a cloud environment. Most of these issues are the result of the shared, on-demand nature of cloud computing [19]. These 11 issues, in order of significance are:

- Data breaches - This is an issue where sensitive, protected or confidential information is released, stolen or viewed. This can be caused by targeted attack, by simple human error, by application vulnerability or by inadequate security practices [19].
- Misconfiguration and inadequate change control - If computing assets are not set up correctly they can be left vulnerable to malicious activity. Misconfiguration of cloud resources, such as containers, can lead to data breaches and potential data loss or modification of resources and services. Most often the absence of an effective change control is the underlying reason [19].

- Lack of cloud security architecture and strategy - Migrating over to the cloud can expose data to different threats in lack of a security architecture and strategy. In most cases functionality and the speed of migration are more important than security which can leave organisations vulnerable to cyber-attacks [19].
- Insufficient identity, credential, access and key management - This is not a new issues and is present in traditional IT environments too, but it becomes more prominent in the cloud. Credentials and cryptographic keys must not be embedded in source code or distributed in public facing repositories. Identity management tools have to be able to scale to handle the lifecycle management of a lot of users. They must support immediate de-provisioning of access [19].
- Account hijacking - This is an issues where highly privileged accounts get abused. In the cloud the highly privileged accounts are the cloud service accounts or subscriptions. These accounts are vulnerable to phisihing attacks, exploitation of cloud based systems or stolen credentials [19].
- Insider threat - An insider can be a current or former employee, contractor or other trusted business partner who has or had authorised access and use this access maliciously or unintentionally to act in a way that negatively affects the organisation. These insiders are dangerous because they operate from within the company's security defenses. Insider negligence is the cause of most security incidents [19].
- Insecure interfaces and APIs - APIs and UIs are the most exposed parts of a system and as such they are very likely to be targets for attacks. This is why security by design and adequate controls protecting them is so important [19].
- Weak control plane - A control plane enables the security and integrity of data, the data plane, and provides stability and runtime of the data. A weak control plane means that the person in charge is not in control of the data infrastructure's logic, security and verification [19].
- Metastructure and applistructure failures - This issue is concerned with the understanding of how to implement proper cloud applications to fully utilise the cloud platform. If an application is not designed for the cloud it might be unable to take full advantage of it and include vulnerabilities that will affect its security [19].
- Limited cloud usage visibility - This issue refers to the ability of an organisation to determine if its cloud usage is safe or malicious. There are two key challenges here: unsanctioned app use and sanctioned app misuse [19].

- Abuse and nefarious use of cloud services - Malicious attackers can use cloud computing resource in an attack or store malware on cloud services. This malware can use cloud-sharing tools to further propagate itself. This issue can be mitigated by an incident response framework that can address the misuse of resources and provides a means for customers to report abuse [19].

3.3 An Overview of Cloud Security Strategies

Most of the above mentioned threats can be mitigated by having the proper security strategies in place. According to Ali et al. [1] security strategies can be divided into two groups: those that address communication issues and those that address architectural issues.

Communication and network is a vital part of cloud computing. To secure these, a combination of LANs, intrusion detection systems (IDS), intrusion prevention systems (IPS) and firewalls are recommended. These are all necessary to protect data in transit. These need to be set up with strict access management policies in order to provide the maximum protection [1].

Architectural issues need to be address from several different angles: virtualisation, data/storage security solutions, security solutions for cloud applications and APIs, identity management and access control, contractual and legal level solutions.

VM images require high security and integrity as they are the starting point for the virtual machine. These images can be used by different, unrelated users so if these images are vulnerable the whole system will be too. Each virtualized OS has to be secured with built-in security measures and third party security technologies. A VM at rest should be encrypted and patched with the latest fixes as soon as possible. Additionally, security vulnerability assessment tools and virtualization aware security tools should be in place [1].

Data and key management are another important aspect that cannot be overlooked. Key management should be kept in house if possible or done by a trusted cryptographic service but in any case, best practices should be followed. Often the best approach is to use an off-the-shelf technology. Standard algorithms should be prioritized over proprietary encryption algorithms [1].

Security for cloud applications and APIs is crucial to provide safe development and execution life cycle. These measures should not assume anything about the external environment. Security and privacy requirements need to be specified in accordance to the need of the cloud development and deployment. Cloud specific risk and

attack vectors have to be identified and the risk and attack models need to be continuously built and maintained. It is advised to re-use software components that are known to mitigate certain security vulnerabilities [1].

Identity and management control should strive to use open standards, such as SAML or OAuth. All entities should have an identified trust level and bi-directional trust needs to be ensured for secure relationship and transaction. Services should have import/export functions into standards [1].

When working with cloud systems security issues related to service-level agreements and geographical legalities also need to be taken into consideration. In many cases a web service agreement is used to define the syntax and semantics of publicizing the competences of the service providers. This is also used to create the template based agreements and to monitor obedience towards these agreements[1].

As the previous sections highlight, security in the cloud is a complex problem and needs to be addressed on different levels. That is one of the reasons why this thesis focuses on security in the cloud on the container level. Nowadays, containers are usually part of modern, cloud-native architectures because they can consistently run anywhere. Thus, cloud security on a container level became one of the areas of interest of cloud security. It is a newer area of cloud security and the focus has mostly been put on static security of containers. Container security during runtime is however equally important [13]. Since there are only a few available solutions, runtime container security is a research area that has clear place for improvement. The following two chapters go into detail about container essentials and runtime container security respectively.

Chapter 4

Containers and Virtual Machines

New technologies and improvements that are constantly emerging in the IT field create more and more pressure on businesses to be increasingly more agile and fast in delivering new functionality and products. However, achieving faster delivery and better agility in IT projects is a complex problem that makes businesses face many challenges. One of the trending concepts that help resolve some of the challenges mentioned is containerization [15]. This chapter looks at this technology, discussing containers and virtual machines, the benefits and drawback of containers and introduces a popular containerization platform, Docker.

4.1 Containerization

Containerization makes it possible to run software code on any infrastructure consistently by encapsulating the code together with all its dependencies [14]. An example of benefits brought by this technology is the fact that packaging up the code in this way allows it to run on different infrastructures without problems, avoiding possible bugs and issues that can occur when moving a software application to a different infrastructure [15]. Of course, this is just one of the benefits that results from this kind of application isolation. When talking about application isolation another technology that is often encountered is virtualization. This technology can be an alternative to containerization, or, in other cases, it can be used together with it in order to achieve better results, but more on the differences between the two technologies later.

Software code packaged together with its dependencies, as previously described,

is referred to as container. Here is a definition of container provided by IBM [16]: “Containers are an executable unit of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether it be on desktop, traditional IT, or the cloud”.

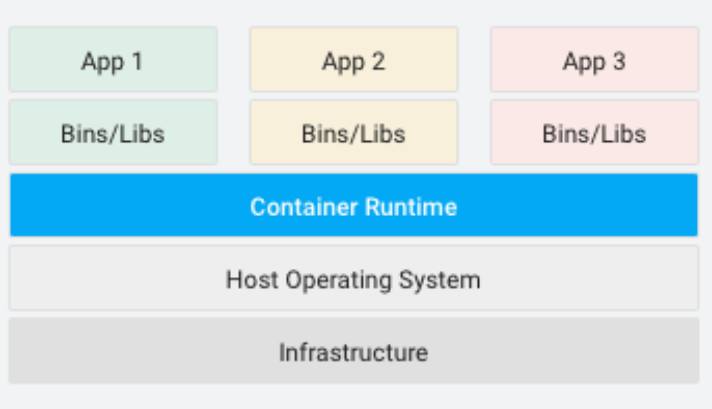

Figure 4.1: Containerization setup example - source: [40]

In the picture above, an example of containerization setup can be seen. In this example we can identify three separate containers which contain one application and its dependencies each. We can also see "Container Runtime" which is a software necessary for the creation and start-up of containers. Below this, is a "Host operating system" which is shared between the containers.

4.2 Virtualization

It was mentioned earlier that besides containerization, another technology was used to isolate applications, and that is virtualization. Before proceeding to compare the two it is important to take a closer look at this technology. Here is a definition of this technology: “Virtualization is a process whereby software is used to create an abstraction layer over computer hardware that allows the hardware elements of a single computer to be divided into multiple virtual computers” [17]. Looking at this definition a few other terms need to be highlighted, that are often used in this context. One is hypervisor, which is the software mentioned in the definition used to create an abstraction layer over the computer hardware. Another important concept is virtual machine, it is an independent virtual computer which is created in the process of virtualization.

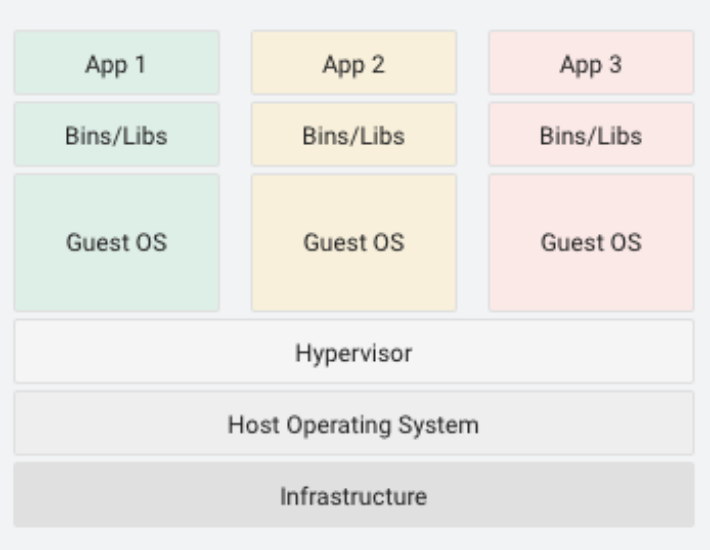

Figure 4.2: Virtualization setup example - source: [40]

Just like the example with the containers, the figure above shows a virtualization setup example. In this example we can see "Hypervisor" which is responsible for the creation and management of virtual machines (VM). In the example, three separate virtual machines can be identified, each containing a "Guest OS", an operating system that the application in the VM can use.

4.3 Containers vs. Virtual Machines

For application isolation one of the two technologies can be selected, containers or virtual machines. Although there are use cases where both technologies are used, each technology has its own purpose in such cases and are not used as a substitute for one another.

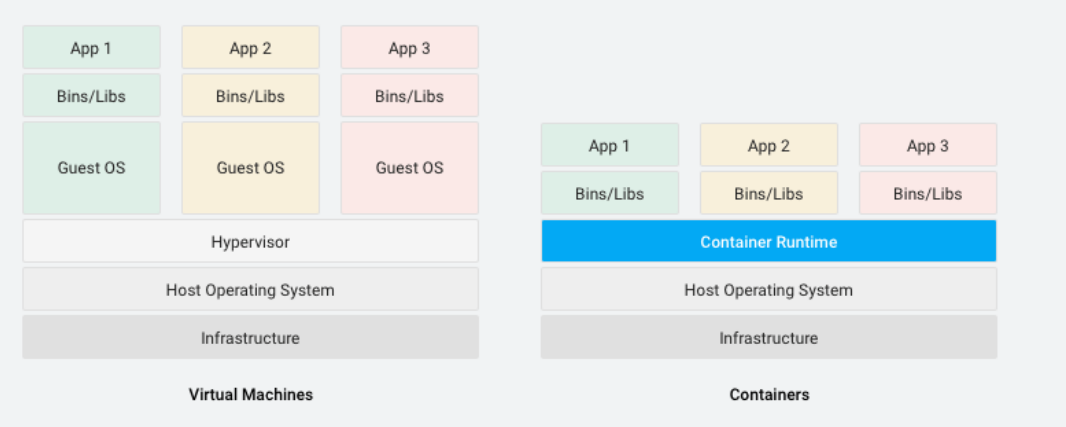

Figure 4.3: Comparison of Containers and Virtual machines setup - source: [40]

From the picture above, it is not difficult to identify the main difference between these two technologies. While virtual machines each have their own "Guest OS", containers all share the same operating system. In case of virtual machines, having separate operating systems provides better isolation and in turn better security, but comes at the cost of performance and computer resources. Containers, on the other hand, share the same operating system, which can mean worse isolation. But, at the same time, sharing a single OS makes containers more light-weight, resulting in faster performance and smaller memory footprint. [17]

4.4 Benefits of Containers

Having discussed the different technologies used to isolate applications and differences between them, it is important to look at a more complete list of benefits that these technologies bring. In particular for this project, the focus is on the benefits of the containers.

An article by IBM lists seven benefits of using containers [14]:

- **Portability** - as mentioned in the definition, containers contain all the dependencies of an application and therefore can be run anywhere.
- **Agility** - standards developed by open source projects such as Docker [6], allow for developers to use DevOps tools together with this technology.
- **Speed** - unlike virtual machines containers do not need to have separate operating systems, they share the same operating system kernel making them

faster and speeding-up start-times since there is no operating system to boot.

- **Fault isolation** - because applications are isolated from one another an issue in one of them does not affect others. This kind of isolation also makes detection of faults easier.
- **Efficiency** - similarly to "Speed", because containers share the same OS, they are smaller in size compared to VMs and faster, allowing for more containers to run on the same machine.
- **Ease of management** - because of container orchestration platforms, managing containerized applications becomes even easier since these platforms can automate tasks such as installation or scaling of containers.
- **Security** - the isolation of applications can prevent other applications from being affected if one of them is compromised.

4.5 Risks of Containers

While containers have many possible benefits and they appear to be very beneficial to software systems, there are several factors that should be considered as possible disadvantages to using containers [39]:

- **Security** - While also mentioned as a benefit, possible security risks remain a concern that need to be considered when adapting this technology. Since this is an important issue closely related to this project, these risks will be discussed in more detail in chapter 5.
- **Development difficulties** - Another concern relates to the addition of new technology to a project. Addition of another technology often brings considerable costs to the project. These costs might appear as time consumed to adapt to new technology or a need to modify the system, eventually the need to maintain the technology throughout the project lifespan should be considered.
- **Resource usage** - While correct usage of this technology can provide considerable benefits, negligence when using it can also negatively impact the system. Although containers can start-up and run faster (compared to VMs) this process still consumes computing resources and should not be neglected in order to avoid possibly reducing the benefits or even increasing the costs of the system.

4.6 Docker

One of the containerization technologies, Docker Engine, had a big impact on the adoption of containerization technologies in the industry [14]. This is also the containerization technology used in this thesis. Docker Engine is a client-server application which is composed of several parts [6]:

- Docker Daemon, which is the 'server' part of the application and is responsible for running client requests.
- Several APIs, used for communication between Docker Daemon and other applications.
- Command line interface (CLI).

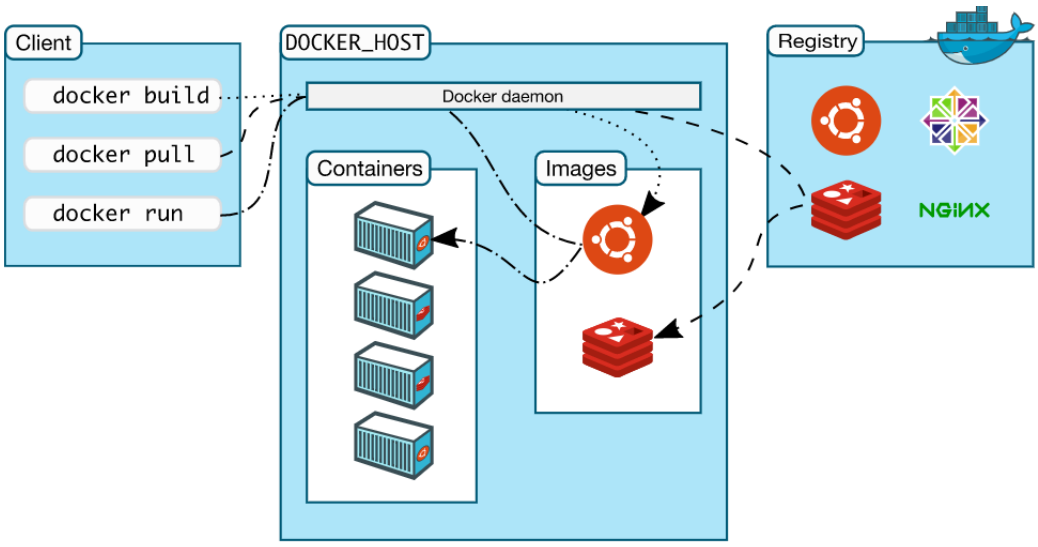

Figure 4.4: Docker architecture - source: [7]

As mentioned earlier, Docker uses client-server architecture, and in the picture above this can also be seen. Client in the picture is responsible for giving instructions to the Docker daemon, and while there are several different client options, a simple example is Command line interface [7]. In the picture we can see several command examples in the client, which can be used to perform different tasks. Docker host in the picture refers to a machine that hosts the Docker daemon, which is not necessarily the same machine where the client is located [7]. Finally, registry can be used for storing and retrieving container images, which will be further explain later.

From the picture, it is clear that Docker has several objects to containerize applications. There are several objects such as images, containers, networks, volumes and more, but here we will focus only on the ones directly used in the project: images and containers [7].

Image

In the documentation available about Docker, images are defined as follows - “An image is a read-only template with instructions for creating a Docker container" [7]. These images can be created by the users or downloaded from registries [7]. These objects can also be based on one another allowing them to be reused and expanded.

Container

Here container refers specifically to a Docker container. Following the definition of the images, in order to create a container an image has to be used. That means that a container is an executable instance of an image [7]. A client (for example CLI) can be used to create, delete, move, start, and stop the container [7]. Based on instructions provided by the image, a container can have several applications in it, which in turn can have a state saved in the persistent storage, and while a stop command, stops the container it does not lose its state and the container can be started again, unless it is deleted [7].

Chapter 5

Container Security

The previous chapter listed as security as one of the drawbacks of containers. This chapter takes a closer look at this topic discussing container threats, existing runtime container security strategies and runtime security tools.

As mentioned before, containers initially emerged as lightweight alternatives to virtual machines. Containers are pivotal to cloud computing and are considered the standard for microservice deployment [36]. More and more companies move towards a container infrastructure when running their applications and this growth is not going to slow down in the near future. According to Million Insights [12] the application container market is expected to grow to 8.2 billion $USD from the $1.5 billion registered in 2018. This growth is thanks to the fact that an increasing number of organisations are turning towards-cloud computing due to the features like huge data storage, low maintenance costs or the assistance in creating, upgrading or deploying solutions on a single OS kernel. Scheduling, scaling, monitoring and management are also easier in a cloud-environment which adds to the cloud's attractiveness for companies.

The biggest issue however is security. While many cloud providers promise increased security, companies are still sceptical about it and adding containers into the mix adds an extra layer of complexity when it comes to security. No matter what platform one chooses to run their containers on, security is one of the harder challenges. The reason for this is because of the multiple moving layers in a cloud native stack and the fact that some distributions of the provider may not be secure by default despite of what operators assume [10]. On one hand, operators make assumptions that are incorrect and do not focus on security early on. On the other hand, even when they are aware of the security needs, operators tend to only partially cover these needs, focusing mainly on image scanning or the host and OS

security. Container images should be scanned and signed before pushed into the registry and security profiles should be set between the container and the host kernel, however this is not enough. These are good practices but they cannot guarantee that malicious individuals won't find a back-door at runtime. For this reason a comprehensive cloud security strategy needs to include real-time, runtime detection of threats and violations [13]. The importance of runtime security is highlighted by the fact that applications deployed in a container will most likely be up and running for years so even a small bug in the code can cause runtime security vulnerabilities for a long time.

Information security in general, and container security in particular, is not a single, one time, action. Security is a process moving through different phases changing, adopting and straightening itself along the way [10]. Security processes and strategies can be grouped into three phases: prevention, detection and response.

Prevention is the fist step of the journey. It includes having proper access control, authentication and authorisation. The goal is to create defense-in-depth, securing each layer and making sure that when a layer fails it fails the safest way possible, returning to a known sate and/or raising an alarm [10]. It is important to know when a layer fails, and this is where **detection** comes in. Following the defense-in-depth principles, each layer should have some kind of detection mechanism in place in order to alert the administrators in due time that something is going wrong [10]. Once a threat has been detected the next step is **response**. In the case of containers this can be an action of killing or pausing the exposed container for example.

Software systems based on containers have three key components [28]:

- The execution environment that is the container itself
- The orchestration and scheduling controller
- The repository that holds the container images or the code

Technology platform aside, container security includes assuring the security of the underlying physical infrastructure, assuring the security of the management plane, properly securing the image repository and building security into the code running inside the container. The underlying infrastructure and its security is not so different from any other form of virtualisation but special attention has to be paid to the underlying operating system where the container's execution environment runs. The orchestrator and scheduler also have to be secured in order to provide comprehensive container security. The image repository is another point of interest and this needs to be secured in a safe location with proper access control. Proper access

control can help with loss or unapproved modification of container images and definition files. It also contributes to the prevention of leaks of sensitive information. Running vulnerable software inside a secure container is still a possibility and this is why it is important to secure the code that is running in a container. Code with vulnerabilities can expose the shared operating system or data from other containers or allow too much network access [28].

5.1 Container Threats

Container environments share many of the same threats as single OS server environments and virtualized environments. As such, containers are vulnerable to threats like [13]:

- Distributed denial of service (DDOS) attacks that seek to make a system unavailable by flooding the bandwidth or resources of the target system disrupting services [35].
- Cross-site scripting (XSS) attacks that are a type of injection attacks where malicious scripts are injected into web pages viewed by other users [31]
- Compromised containers trying to download malware
- Gaining access to vulnerabilities and weaknesses by compromised containers scanning other internal systems
- Unauthorised access across containers, hosts or data centers
- Container resource hogging to starve other containers of resources
- Live patching applications that can bring in malicious processes from hijacked DNS or other services
- Poorly designed applications can cause network flooding that can impact other containers

These threats can be exploited by malicious individuals if the proper security measures are not in place. For example, a malignant user can use an SQL injection attack to gain access to a database container and steal data. Other, well-known bugs like the shellshock bug or the heartbleed bug can also be used in an attack against containers. Shellshock is a family of security bugs affecting the Unix Bash shell that allows remote attackers to execute arbitrary code in the container. Heartbleed is an OpenSSL bug that causes containers to leak memory and allow malicious users to analyze that leak [33].

5.2 Runtime Container Security Strategies

As it was previously highlighted, container security is a continuous battle that needs constant attention and monitoring. As such it is important to have runtime security strategies in place because containers are likely to be "alive" for a long period of time once they have been deployed. Huang [13] proposes a comprehensive list of items that one needs to pay attention to when implementing runtime container security. These items are organised in three groups: preparing for production, basic runtime container security and advanced runtime container security:

1. Preparing for production [13]
    - Secure the OS - apply the latest security patches and remove all unnecessary files and modules.
    - Follow the best practices suggested by the container platform used.
    - Make sure to have authentication and authorisation in place.
    - Vulnerability scan containers in all registries.
    - Digitally sign or do integrity checks on container images.
2. Basic runtime container security [13]
    - Secure the data center at the gateway or entry point.
    - Keep unused containers under control by regularly cleaning them up.
    - Load application containers in read-only/non-persistent mode.
3. Advanced runtime container security [13]
    - Isolate containers into the minimum working zone.
    - Real-time attack monitoring.
    - Monitor container behaviour for violations.
    - Live scan the running containers for vulnerabilities.
    - Automate security policies.
    - Analyze past security events to correlate events and store forensic data for containers.

In this thesis the focus is on a couple of the advanced runtime container security strategies, namely: real-time attack monitoring and monitoring container behaviour for violations.

5.3 Runtime Security Tools

Given the popularity of containers there are a number of tools available today that are geared specifically towards container security. The majority of these tools are concerned with the security of the container images, which is a static approach to container security. While securing container images is important because they are the foundations that the containers will be built on, container security does not stop at securing these images. The security of running containers is equally important [13]. As it was stated earlier, our goal in this thesis is to take a closer look at one of the runtime security tools and improve on false negative results. To this end we selected **Falco** as the runtime security tool.

Falco is an open source runtime security tool created by Sysdig. Some of the main reasons behind the choice of Falco for this thesis were the fact that it is open source with a large community behind it and it was the first runtime security project that joined the Cloud Native Computing Foundation (CNCF) so it has a certain level of maturity.

Falco detects unexpected behaviour in the containers and sends out alerts at runtime. It requires a driver to listen to the Linux Kernel which allows it to have a deep visibility into all system call activities, such as security events [37]. It uses cloud audit logs and provides threat detection and alerts based on these. Falco allows its users to create detection rules to define unexpected behaviour. It strengthens container cloud security as it uses a common policy language to detect threats in and across containers and hosts. Given the nature of its immediate alert system it reduces risk significantly because it allows for immediate response. It leverages the most current detection rules and its out-of-the box rules alert about malicious activities, common vulnerabilities and exposures [37].

A comprehensive list of Falco's features can be seen in the list below [37]:

- **Deployment**
  - Licensing - Open source Apache V2 license
  - Installation - Daemonset via Helm, package manager, Docker container
  - Installation support - Community supported
- **Continuous Cloud Security Posture Management**
  - Threat detection based on cloud logs such as suspicious logins or file access
  - Context enrichment - Cloud, host, containers and Kubernetes labels

- **Compliance**
  - CIS Benchmarks, PCI controls, NIST 800-190 controls - Compliance rules can be created at runtime
- **Detection**
  - Runtime detection
  - Detection of anomalous behaviour on new logins, file access, network, system calls and storage writes
  - Detection of anomalous behaviour on Kubernetes API calls
  - eBPF (extended Berkeley Packet Filter) probe
  - Kernel module probe
  - Metadata context - Cloud, host, container and Kubernetes labels
- **Response**
  - Default notification channels - requires 3rd party components
- **Policy management**
  - Out-of-the-box rules library - Community created
- **Additional security**
  - Audit - A records of all commands executed on cloud accounts and assets can be built with own external database

Falco has an impressive list of features but what it essentially does, is constantly monitoring the container cluster that it is deployed in and based on its rule set it sends out alerts if it detects malicious behaviour. The bigger the cluster, the more containers Falco has to monitor, the more notifications are sent out. The alerts can have different level of severity. These are, in order of severity (from lowest to highest): debug, info, notice, warning, error, critical, alert, emergency.

We conducted some early experiments with Falco where we simulated two different attacks: a compromised container attack which tried to download malware and a resource hogging attack. These early experiments showed interesting results. While Falco was able to detect these attacks and send out alerts for them, the severity of these alerts were of "notice", which is the third lowest severity level. Since the severity level of these attacks were so low they could have been overlooked by an administrator allowing the malicious behaviour to continue. Falco ranks these attacks at such a low severity level because some of the activities can be the results

of normal usage. Falco has no way of differentiating between an actual attack and regular usage. This fact sparked the idea of a tool that would monitor container metrics, separately from Falco, and find anomalies in metric usage. We set out to create the **Container Overseer**, a tool that uses three different algorithms to detect anomalies in container metrics. If it finds certain anomalies it checks if Falco has also raised an alert and if it did, the severity level of the alert gets increased to a higher priority, giving it a higher chance to be noticed by a system administrator.

Chapter 6

Experiments

The following chapter talks about the experiment location and setup, the Container Overseer and the three algorithms it uses. Additionally it presents the experiments conducted with the Container Overseer, Falco and Falco Sidekick and describes the simulated attacks used to test the performance of the aforementioned tools.

6.1 Experiment Setup

This thesis is a collaboration with Keysight Technologies, an electrical and electronic manufacturing company, whose goal is to connect the world in a safe manner by exploring the edges of test and measurement science [3]. In 2018 Keysight joined the European (EU) project – 5G Vertical INNovation Infrastructure (5G-VINNI) as the only test, measurement and network visibility company. The aim of the 5G-VINNI project is to accelerate the acceptance of 5G across Europe [3].

In order to adequately support the needs of the 5G-VINNI project, Keysight set up an on-premise cloud to provide on-demand network access to a shared pool of resources that can be rapidly provisioned and released. Within the 5G-VINNI project Keysight is a cloud provider that provides testing capabilities for the partners of the project via SaaS. As it was described in chapter 3, SaaS is a type of service models which supplies everything that PaaS and IaaS does and in addition provides access to cloud hosted applications [3]. These applications run in containers thus container security is a topic of interest for the company.

In this thesis, we used this on-premise cloud as our test bed. The cloud is using OpenStack for virtual infrastructure management. This is a free, open-source cloud

computing infrastructure software for virtual machines, bare metal and containers. It provides a large pool of compute, storage, and networking resources, that are managed through APIs or a dashboard. Through OpenStack standard IaaS is available and on top of that, it has additional components that provide orchestration, fault management and service management amongst other services [29].

From the point of view of the hardware, Keysight's cloud is hosted on five physical nodes, some focusing on providing compute resources, while some focus on networking and storage while routing and switching is handled by OpenStack. In this cloud setup there are two compute nodes, one control node, one netwrok node and one storage node [2]. The nodes have the following characteristics:

	Compute	Control	Network	Storage
System	Dell PowerEdge R630	Dell PowerEdge R630	Dell PowerEdge R630	Dell PowerEdge R630
vCPU	32 cores @ 2.1 Ghz	16 cores @ 2.1 Ghz	16 cores @ 2.1 Ghz	16 cores @ 2.1 Ghz
CPU Model	Intel(R) Xeon(R) CPU E5-2620 v4	Intel(R) Xeon(R) CPU E5-2620 v4	Intel(R) Xeon(R) CPU E5-2620 v4	Intel(R) Xeon(R) CPU E5-2620 v4
Memory	96.0 GiB RAM	48.0 GiB	48.0 GiB	48.0 GiB
Storage	1x500GB (ssd)	1300.2 GB over 3 disks 2x500 GB (ssd) 1x300 GB (hdd)	1300 GB	3300.6 GB over 7 disks 6x500 GB (ssd) 1x300 GB (hdd)
Deployed OS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS
Kernel	bionic	bionic	bionic	bionic

Table 6.1: Characteristics of the physical nodes

For the purposes of our experiments we set up a virtual machine in this cloud. The underlying OS for this VM is Ubuntu 20.04.1. The experiments were ran on a machine with a 2 core CPU, 4GB of RAM and a total disk space of 20GB. Following this we installed Docker into this VM. Docker is a widely used PaaS product that uses OS level virtualization to deliver software in packages called containers [3]. Working with Docker containers makes it easy to handle if something goes wrong because it allows us to restart containers instead of tearing down and rebuilding the whole VM.

After this initial setup the next step was to bring our Container Overseer to the cloud and set up Falco and Falco Sidekick before we could simulate the attacks.

6.2 Container Overseer

The Container Overseer is the main contribution of this thesis. This is a C#, .NET Core application, that has two main goals:

- Monitor container metrics, such as CPU and memory usage, during runtime and identify suspicious behaviour based on these metrics.

- Receive the alerts from Falco and increase their severity level if the container metrics also show suspicious behaviour.

To achieve the first goal, the Container Overseer implements three algorithms to monitor container metrics. These are: local outlier factor, threshold or Welford's algorithm and quartiles-based algorithm. For the second goal we needed to set up Falco and Falco Sidekick on the cloud. The next sections discuss these in details.

6.2.1 Algorithms

The algorithms used in the Container Overseer have to fulfil several criteria in order to achieve the task described before. One of the requirements is capability to handle real-time stream data. Container resource metrics information which are being analysed are received with short intervals between every update to the metrics, so in order to select the appropriate algorithms to handle this kind of data it is important to understand the restrictions that are imposed by it. The article called *Detecting real-time and unsupervised anomalies in streaming data* [24] defines four constraints that need to be considered:

- Stream data often arrive one item at the time and can be read only once, which means that algorithms has to be able to decide if the data should be saved or discarded when it arrives.
- Only select, relatively small number of data instances can be stored in the memory.
- Processing time of data should be short.
- A model produced by an algorithm working on a data stream has to be equivalent to a model produced by an algorithm that works with data in batches (possibly historical data).

Looking at these requirements and understanding the difficulties of working with stream data, we selected several algorithms that fulfil these requirements. The selected algorithms are fast enough to finish processing before the next data instance arrives, work on a limited number of observations while continuously incorporating new data and, finally, since the main purpose of these algorithms is anomaly detection, the algorithms can classify/identify an anomaly whenever new data instance arrives, allowing the system to appropriately archive or discard the data.

Another requirement that was considered when selecting the algorithms was their simplicity and transparency. Although it is sometimes assumed that simple algorithms will produce worse results compared to more sophisticated algorithms, it is often much more difficult to analyse and understand the results of complicated algorithms and find the underlying problems that could cause bad results. This lead to a decision to work with simpler algorithms as a proof of concept and to better understand what might be the limitations and issues when trying to detect anomalies in the systems.

6.2.1.1 Threshold algorithm

The Algorithm referred to as the Threshold algorithm in the research paper *Application of anomaly detection algorithms for detecting SYN flooding attacks* [34] or a very similar algorithm called The Welford's algorithm [24] was the first algorithm that the group selected for the task. The main idea behind this algorithm is to calculate a mean over some period of time and at the same time a standard deviation in that same period. Then based on these calculated values establish a threshold which would indicate an anomaly when crossed. A more mathematical formula for this algorithm [24]:

$$L = OnlineMean + X * OnlineStd$$

In this simple expression, the purpose is to calculate limit L which represents the threshold. The threshold is calculated using three values. $OnlineMean$ is an average value over some period of time (which is updated with every new observation). $OnlineStd$ is the standard deviation calculated over the same period as $OnlineMean$, it is also updated with every new observation. Finally, X is a variable which is used to adjust the threshold, depending on the selected value, the false positive and false negative ratio can be adjusted [24].

6.2.1.2 Quartiles-based algorithm

Another algorithms which was used in order to detect anomalies in the observations was a solution based on quartiles calculations. Similarly to a previous algorithm this algorithm defines a threshold which is used to determine if an observation is an anomaly. For this algorithm it is important to calculate three different quartiles: 1st quartile (25th percentile), median (50th percentile) and 3rd quartile (75th percentile). In order to calculate quartiles it is important to have an ordered list of observation.

This list used for the algorithm consists of observations from the data stream over some period of time. In order to calculate the threshold value with this algorithm the following formula is used:

$$IQR = Q3 - Q1$$

$$L = Q3 + X * IQR$$

In the first expression the interquartile range, IQR, is calculated, which is later used in the calculation of limit L, the threshold. $Q1$ and$Q3$ refer to the values of 1st quartile and 3rd quartile respectively. In the second expression, the calculation for the threshold can be seen. Here X is a variable used to adjust the threshold.

6.2.1.3 Local Outlier Factor (LOF)

Finally, the third algorithm that we have selected is the Local Outlier Factor (LOF) algorithm. This algorithm was first introduced in the research paper *LOF: Identifying Density-Based Local Outliers* [4]. A simple explanation of this algorithm is that it looks at a number of nearest neighbors of a new observation and compares the density around the new observation with local density around those nearest neighbors, in order to determine whether the new observation belongs to a cluster or is an outlier. For a more formal definition, three different formulas will be discussed in order to explain the process of calculating LOF. First, in order to calculate LOF it is necessary to understand how to calculate reachability distance [4]:

$$\textit{reach-distance}_k(p, o) = \max\left(\textit{k-distance}(o), d(p, o)\right)$$

The formula above calculates the reachability distance of point (observation) p with respect to point o. Subscript $_k$ in this expression represents the number of closest neighbors that should be inspected. Function $d(p, o)$ represents distance calculation between two point p and o [4]. Another function seen in the formula, which need to be explained is $k - \textit{distance}(o)$, this function represents the distance from point o to its k^{th} nearest neighbor. In order to better understand this explanation of these functions in the formula, a visual representation from the original research paper [4] is presented below:

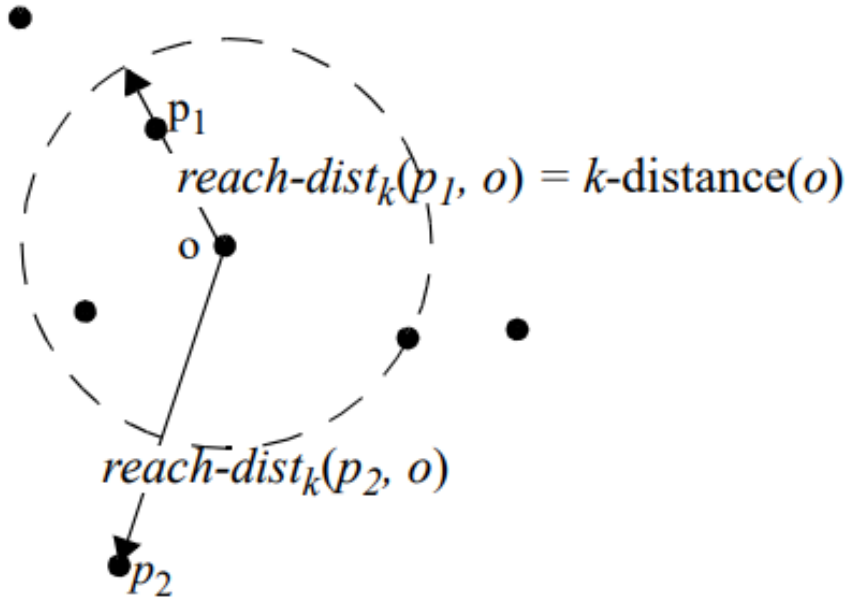

Figure 6.1: Reachability distance illustration (where k = 4) - source: [4]

With the knowledge of how to calculate reachability distance, the formula to calculate local reachability density (LRD) can be investigated, which is the next step necessary to calculate LOF.

$$LRD_k(p) = 1/(\frac{\sum_{o \in N_k(p)} reach\text{-}distance(p,o)}{|N_k(p)|})$$

In order to understand the expression above, it is important to explain what $N_k(p)$ means. This function returns a set of k nearest neighbors to point p. This in turn also means that $|N_k(p)|$ is the number of items in that set. With this information, it can be seen that local reachability density is an inverse of average reachability distance of p's k nearest neighbors.

Finally, the local outlier factor can be calculated using the formulas that where explained above as follows [4]:

$$LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{LRD_k(o)}{LRD_k(p)}}{|N_k(p)|}$$

The results of this calculation can be interpreted as follows [4]:

- If LOF ~ 1, it means that the new observation has approximately the same density as its neighbors and therefore belong to the cluster.
- If LOF < 1, it means that the new observation is in a denser region, as it has higher local density than its neighbors.
- If LOF > 1, it means that the new observation is in a less populated region, meaning that it is an outlier.

6.3 Falco and Falco Sidekick

Falco has already been described in chapter 5. It was set up in the cloud using a Docker image. The free, open-source version of Falco that was set up on the cloud has five outputs out-of-the-box for events: stdout, file, gRPC, shell and http. These outputs are convenient but limit Falco's integration with other components. Falco Sidekick is a simple daemon that extends the number of possible outputs [22]. Falco Sidekick takes Falco's events and forwards it do different outputs like: Slack, Teams, Datadog, Discord, ElasticSearch, Loki, NATS, RabbitMQ, Kubeless, Kafka, WebUI and others. Moreover, it provides metrics about the number of events and allows the users to add custom fields in events [22].

We chose to use Falco Sidekick because we wanted to take advantage of its capability to sent notification to NATS. NATS is a connective technology responsible for addressing, discovery and exchanging of messages that have a common pattern in distributed systems. NATS uses subjects for addressing and discovery instead of host names and ports which provides and abstraction layer between the application or service and the underlying physical network. A publisher sends out a message which is then received, decoded and processed by one or more subscribers. NATS is easy to deploy and works everywhere from bare metal, through VMs and Kubernetes clusters to devices and is secure by design. NATS experiences a widespread use for cloud messaging between services and for event and data streaming [5].

For the purpose of the experiments we deployed a NATS server in the VM described previously. For our purposes we used a single NATS server that was installed in the cloud via Docker. We implemented a publish-subscribe message distribution model where a publisher, in our case Falco, sends messages, Falco alerts, on a subject, and a subscriber, the Container Overseer, receives it. There is one thing one needs to be aware of when working with NATS and that is the fact that it offers the same level of message guarantee as TCP/IP. In other words, if there is no subscriber listening

for the message then the message will be lost [5]. Core NATS is basically a fire-and-forget messaging system but it suited our purposes perfectly for the experiments.

6.4 Simulated Attacks

With all the set up ready the next step was to simulate some attacks. The experiments started with attacks that impacted certain container metrics using the Mischief Simulator, a tool we created. Following these we experimented with attacks from Keysight's Threat Simulator to see how does the Container Overseer hold up against attacks simulated by this tool. In these experiments we focus on two metrics: CPU and memory usage.

6.4.1 Mischief Simulator

The Mischief Simulator is a small tool created by the writers of this thesis. This is a simple tool that, through the use infinite loops, busy waiting and other similar actions, creates two different states in the container. Normal performance state, where an application hosted in the container performs as it is expected and attack state, where the application simulates resource consumption which is to be expected if the container is attacked/misused in a computing resource intensive attacks. These two states allows to simulate several different scenarios where a container is compromised.

The Mischief Simulator successfully imitates CPU and memory hogging. By adjusting the intensity of the attacks (amount of resources consumed during attack) and how stable or varying the computing resource consumption is during normal performance state, it becomes possible to analyse which changes in the system affect the algorithms increasing or lowering the detection rates.

Using the Mischief Simulator we ran several experiments. To achieve the final goal, to raise the priority level of the alerts from Falco, we needed to determine the best settings for the algorithms discussed previously. Therefore, to better adjust the algorithms and get more accurate results several scenarios where defined in this application.

One of the simulations that Mischief Simulator was used for is creating an environment in a container where application continuously consumes a relatively constant amount of CPU power and after a set time it changes to an attacked state where the CPU power consumption increases sharply, but similarly to the previous state it

maintains a rather stable consumption of the computational power.

Another scenario which is used to test all the algorithms follows the same sequence, but instead of a very steady and constant resource usage, the resource usage still maintains a rather constant average consumption, but compared to before has a much higher standard deviation. This change applies to both simulated states.

Finally, a scenario specifically for the purpose of simulating attack that is heavily dependant on RAM usage was created, this scenario similarly to others simulates increased usage of RAM. Differently from previous two attacks, this attack tries to simulate a scenario where computer resource usage is increased more gradually and then, after the simulated attack, the resources used are released all at once.

6.4.2 Threat Simulator

Threat Simulator is a breach defense tool created by Keysight Technologies that automatically scans perimeter defenses, web application firewall and web policy engines with the goal of identifying any vulnerabilities. It is a cloud-based platform that uses a microservice architecture and is delivered as a SaaS solution [21]. Threat Simulator has three core components [21]:

- A user-friendly web based interface
- A dark cloud entity that is responsible for spinning up agents to simulate threat actors
- Agents, available in Docker container format, that act as simulated target or attackers inside the network

Threat Simulator is capable of simulating the entire kill chain, which is a chain of cyber intrusion activities: reconnaissance, weaponization, delivery, exploitation, installation, command and control, and actions on objectives [25]. Furthermore, it can analyze the detection and blocking capabilities of the the system and provide recommendations for actions that can be taken to further improve security. It also performs reassessments of the environment thus providing continuous validation [21].

The attacks simulated by Threat Simulator are safe because it never actually interacts with the production server. Instead it uses isolated software endpoints across the network. These are the endpoints the dark cloud connects to [21].

For our experiments we deployed a Threat Simulator agent in Keysight's cloud infrastructure. This agent identified services that the location can use to access systems outside the cloud and services that outside systems can use to access the cloud. This agent executes the audits that constitute a security assessment. The agent recreates the network behaviour between simulated adversaries outside and inside the organization, such as hackers, malicious domains, infected hosts and the like, together with their corresponding targets. The agent is a containerized application that runs on an Linux-based distribution. It is installed beside the existing production servers and user workstations [20]. For the purposes of our experiment we used an infrastructure-agnostic agent that provides generic methods to deploy the Threat Simulator agent on our infrastructure.

After this we set up a couple of scenarios that are assessments which run on specified agents [21]. We set up the following scenarios for our experiments:

- Suspicious User Behaviour - this assessment validates the security controls for Local Area Network users and devices. This is a collection of attacks and suspicious network traffic that simulates attacks like Monero Mining Traffic, Tor Darkweb Connection, OpenVPN connection and data or Bitcoin mining activity among others.
- Malicious Media file transfer - this assessment contains a collection of malicious media files that contain local exploits designed to alter the execution flow of the applications operating with these media files. It performs 15 different attacks using the drive-by compromise method, where the users browser is targeted for exploitation.
- Sunburst December 2020 Campaign - this is a highly-sophisticated supply chain attack that uses trojanized updates to get access to the system. It simulates the download of Sunburst Malware than it simulates the traffic that occurs after executing this malware. Within this scenario there are two attacks, the first installing the compromised update, the second taking over the control of the system.

Once a scenario is configured the assessment can start one or more audits, that aim to verify that the security control is capable of blocking the attack. If the audit passes the malicious attack was blocked. If an audit fails Threat Simulator provides a list of recommendations of actions that can be performed in order to improve security.

The results of the experiments with the both Mischief Simulator and Threat Simulator are discussed in the next chapter. To conclude this chapter the following diagram presents the complete experiments set up on the Keysight cloud:

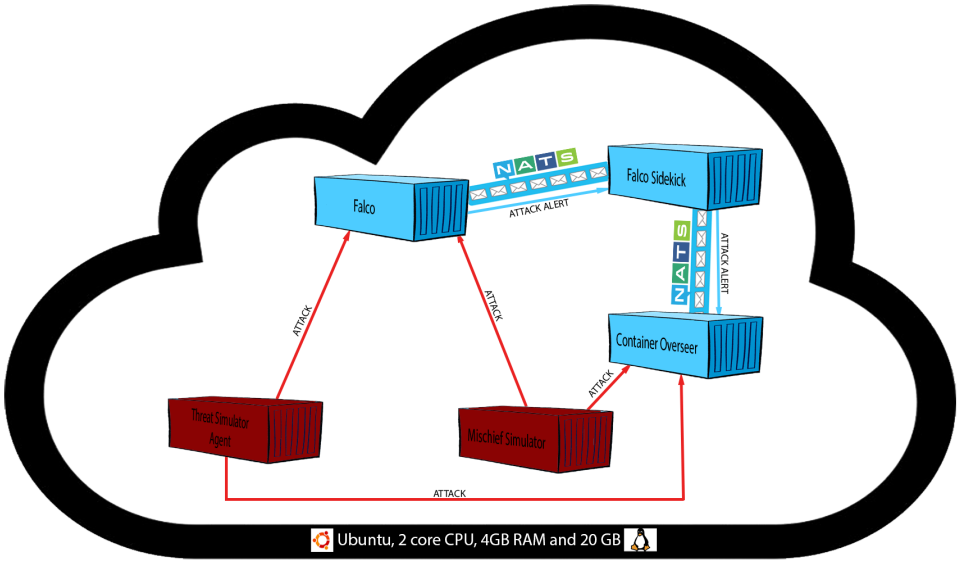

Figure 6.2: Cloud Setup - Own creation

In this setup the Threat Simulator Agent and the Mischief Simulator are the ones that are executing attacks. These attacks are picked up by the Container Overseer thanks to its algorithms monitoring the container metrics. The attacks are also picked up by Falco that uses NATS messaging to send out alerts via Falco Sidekick. The Container Overseer listens for these alerts on a NATS subject and acts whenever an alert comes in.

Chapter 7

Experiment Results

This chapter presents the results of the experiments that were ran on the private cloud. The first part presents experiments ran with the Container Overseer and the Mischief Simulator. These serve as proof of concept that the selected algorithms are suitable for the underlying problem. Here we started off with some simple experiments monitoring CPU usage. Following this, some more complex attacks were executed with the Mischief Simulator where CPU usage was monitored. Moreover, we ran experiments where instead of the CPU usage the focus was on memory usage. The second part of this chapter presents the experiments conducted with Keysight's Threat Simulator, Falco and the Container Overseer.

7.1 Experiments with the Mischief Simulator

As described in chapter 6, the Mischief Simulator is a simple tool, created to simulate suspicious behaviour. The simple attacks, performed with this tool, simulate a container that consumes a relatively constant amount of CPU but when an attack happens this consumption spikes. The more complex attacks have a little more erratic CPU usage.

7.1.1 Simple CPU Attacks

The simple attacks with the Mischief Simulator served as a proof of concept supporting the selection of the algorithms. With these experiments we tested the three algorithms with different setups. The base setup looked at 10, 20, 50 and 100 pre-

vious observations but where there was a reason we considered even more previous observations. For the threshold algorithm different standard deviation values were tested, ranging from 1 to 4 times the value of the standard deviation. For the quartiles-based algorithm the variables to adjust the threshold by were 0.5, 1.5, 1 and 2 respectively. Finally, for LOF we looked at the distance to the 2^{nd}, 3^{rd}, 4^{th} and 5^{th} nearest neighbour. These adjustments influence the placement of the threshold, placing it further away from the mean. The goal is to find a threshold that provides the lowest amounts of false positive and false negative answers.

7.1.1.1 Threshold Algorithm

In the case of the simple attacks with the Mischief Simulator, algorithm configuration only slightly affected the experiment results. The following figures show the results from the threshold algorithm when it looks at 10 and 100 observations respectively with a standard deviation modifier of 4:

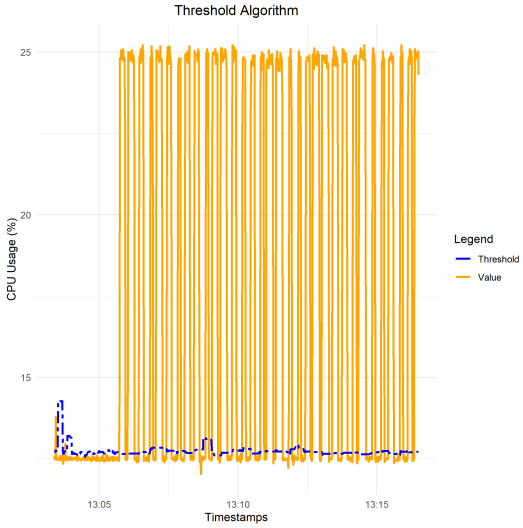

Figure 7.1: Threshold Algorithm with 10 memory and 4*std - Simple Attack Scenario

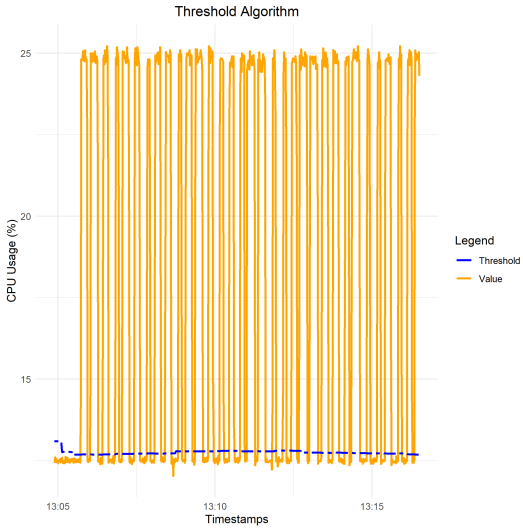

Figure 7.2: Threshold Algorithm with 100 memory and 4*std - Simple Attack Scenarios

s

The plots above show that the threshold algorithm produces a more even threshold line when it looks at 100 observations. The successful detection rates of the algorithm can be seen from the table below:

	Algorithm Setup	Succ*e*ssfulDetection	False*e*Positive	False*e*Negative
1	ThresholdAlgorithm-n1-mem10	0.876	0.124	0.000
2	ThresholdAlgorithm-n2-mem10	0.925	0.075	0.000
3	ThresholdAlgorithm-n3-mem10	0.948	0.052	0.000
4	ThresholdAlgorithm-n4-mem10	0.950	0.050	0.000
5	ThresholdAlgorithm-n1-mem20	0.879	0.121	0.000
6	ThresholdAlgorithm-n2-mem20	0.931	0.069	0.000
7	ThresholdAlgorithm-n3-mem20	0.950	0.050	0.000
8	ThresholdAlgorithm-n4-mem20	0.951	0.049	0.000
9	ThresholdAlgorithm-n1-mem50	0.884	0.116	0.000
10	ThresholdAlgorithm-n2-mem50	0.936	0.064	0.000
11	ThresholdAlgorithm-n3-mem50	0.948	0.052	0.000
12	ThresholdAlgorithm-n4-mem50	0.951	0.049	0.000
13	ThresholdAlgorithm-n1-mem100	0.885	0.115	0.000
14	ThresholdAlgorithm-n2-mem100	0.936	0.064	0.000
15	ThresholdAlgorithm-n3-mem100	0.950	0.050	0.000
16	ThresholdAlgorithm-n4-mem100	0.951	0.049	0.000

Table 7.1: Success rates of the Threshold Algorithm - Simple Attack Scenario

From the above table it can be seen that the threshold algorithm was ran in 16 different combinations, where the Mischief Simulator simulated attacks, to see which algorithm setup has the highest detection rate. The results show that all setups performed pretty well, the lowest detection rate being 87.6%. The best detection rate, above 95%, was achieved in 3 separate setups, looking at 20, 50 and 100 previous observations and all having 4 as the standard deviation modifier.

7.1.1.2 Quartiles-based Algorithm

The quartiles-based algorithm performed similarly to the threshold algorithm when tested with the Mischief Simulator. The results of this algorithm with 10 and 100 previous observations can be seen below:

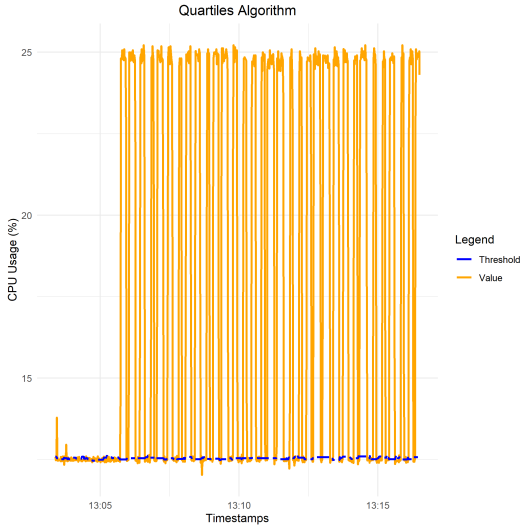

Figure 7.3: Quartiles-based Algorithm with 10 memory and a variable of 2 - Simple Attack Scenario

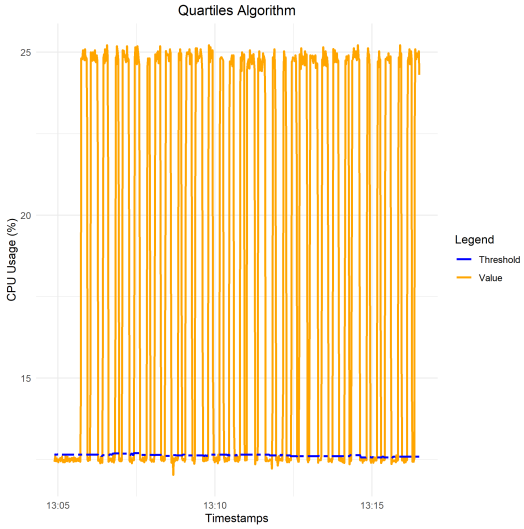

Figure 7.4: Quartiles-based Algorithm with 100 memory and a variable of 2 - Simple Attack Scenario

Both cases produced even threshold lines and showed high detection rates. To find the best performing algorithm setup for the quartiles-based algorithm we looked at 16 different combinations:

	Algorithm Setup	SuccessfulDetection	FalsePositive	FalseNegative
1	QuartilesBasedAlgorithm-iqr0.5-mem10	0.731	0.269	0.000
2	QuartilesBasedAlgorithm-iqr1-mem10	0.804	0.196	0.000
3	QuartilesBasedAlgorithm-iqr1.5-mem10	0.867	0.133	0.000
4	QuartilesBasedAlgorithm-iqr2-mem10	0.881	0.119	0.000
5	QuartilesBasedAlgorithm-iqr0.5-mem20	0.810	0.190	0.000
6	QuartilesBasedAlgorithm-iqr1-mem20	0.850	0.150	0.000
7	QuartilesBasedAlgorithm-iqr1.5-mem20	0.881	0.119	0.000
8	QuartilesBasedAlgorithm-iqr2-mem20	0.898	0.102	0.000
9	QuartilesBasedAlgorithm-iqr0.5-mem50	0.826	0.174	0.000
10	QuartilesBasedAlgorithm-iqr1-mem50	0.846	0.154	0.000
11	QuartilesBasedAlgorithm-iqr1.5-mem50	0.867	0.133	0.000
12	QuartilesBasedAlgorithm-iqr2-mem50	0.888	0.112	0.000
13	QuartilesBasedAlgorithm-iqr0.5-mem100	0.827	0.173	0.000
14	QuartilesBasedAlgorithm-iqr1-mem100	0.850	0.150	0.000
15	QuartilesBasedAlgorithm-iqr1.5-mem100	0.882	0.118	0.000
16	QuartilesBasedAlgorithm-iqr2-mem100	0.942	0.058	0.000

Table 7.2: Success rates of the Quartiles-based Algorithm - Simple Attack Scenario

From this table it can be seen that the highest detection rate, above 94%, was achieved when the algorithm looked at 100 previous observations and used 2 as the variable

to influence the placement of the threshold. It is noteworthy that all algorithm setups had a detection rate above 73%.

7.1.1.3 LOF

For these simple experiments LOF produced similar results to the previous two algorithms. The figure below is a partial result of the Container Overseer's log output for the LOF algorithm that shows that this also detected problems at the same time stamps as the other two algorithms and registered no issues where the other two registered no issues.

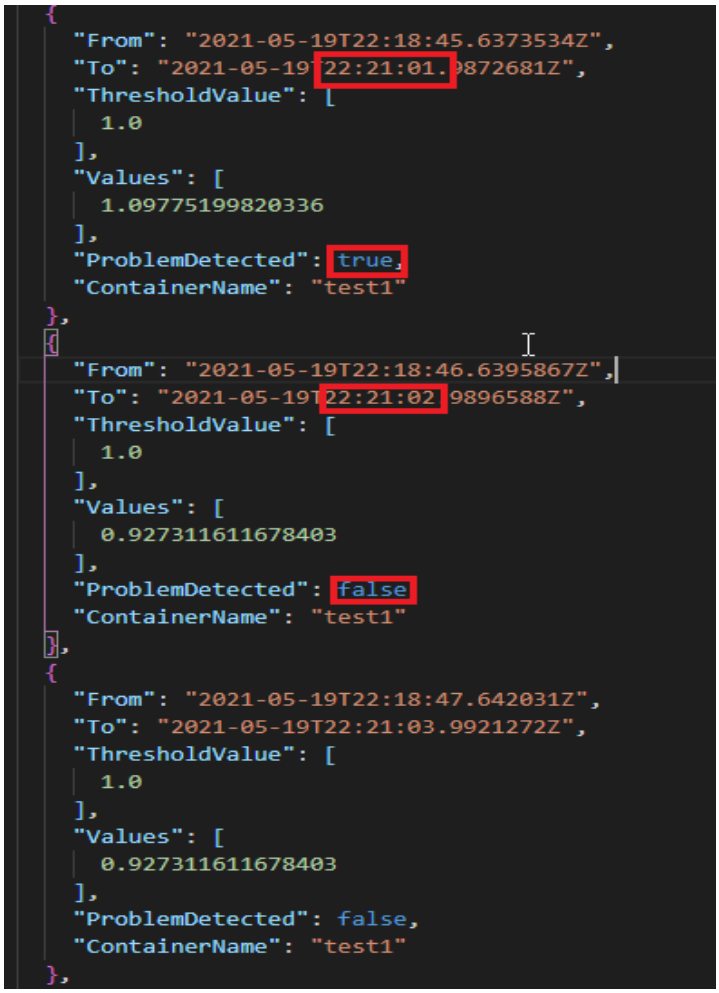

Figure 7.5: LOF Algorithm Simple Attack Scenario

Similarly to the previous two algorithms, we ran LOF in 16 different configuration setups to determine the best performing setup:

	Algorithm Setup	Succ*e*ssfulDetection	Fals*e*Positive	Fals*e*Negative
1	LOFAlgorithm-nn2-mem10	0.521	0.479	0.000
2	LOFAlgorithm-nn3-mem10	0.528	0.472	0.000
3	LOFAlgorithm-nn4-mem10	0.518	0.482	0.000
4	LOFAlgorithm-nn5-mem10	0.518	0.482	0.000
5	LOFAlgorithm-nn2-mem20	0.613	0.385	0.002
6	LOFAlgorithm-nn3-mem20	0.615	0.384	0.002
7	LOFAlgorithm-nn4-mem20	0.570	0.428	0.002
8	LOFAlgorithm-nn5-mem20	0.587	0.411	0.002
9	LOFAlgorithm-nn2-mem50	0.703	0.295	0.002
10	LOFAlgorithm-nn3-mem50	0.708	0.291	0.002
11	LOFAlgorithm-nn4-mem50	0.631	0.367	0.002
12	LOFAlgorithm-nn5-mem50	0.668	0.330	0.002
13	LOFAlgorithm-nn2-mem100	0.784	0.214	0.002
14	LOFAlgorithm-nn3-mem100	0.610	0.388	0.002
15	LOFAlgorithm-nn4-mem100	0.734	0.265	0.002
16	LOFAlgorithm-nn5-mem100	0.651	0.347	0.002

Table 7.3: Success rates of LOF - Simple Attack Scenario

The table shows that out of the three algorithms tested LOF had the worst successful detection rate, the highest being 78.4% where the algorithm looked at 100 previous observations and considered the distance to the 2^{nd} nearest neighbour. It is also interesting to note that out of the three algorithms only this one produced any false negative results.

These results prompted us to further test the performance of these algorithms with more complex attack scenarios from the Mischief Simulator.

7.1.2 Complex CPU Attacks

These attacks were also conducted by the Mischief Simulator but using the second scenario described in chapter 6.4.1, where the simulated attack is a bit more erratic. The algorithms had the same setups as in the case of the simple experiments. In the case of the threshold algorithm and the quartiles-based algorithm we looked at 10, 20, 50 and 100 previous observations while in the case of LOF this was extended to 200, 300, 400 and 500.

7.1.2.1 Threshold Algorithm

The figures below show the results obtained with the threshold algorithm with 10 and 100 observations respectively and 2 as the value of the standard deviation modifier:

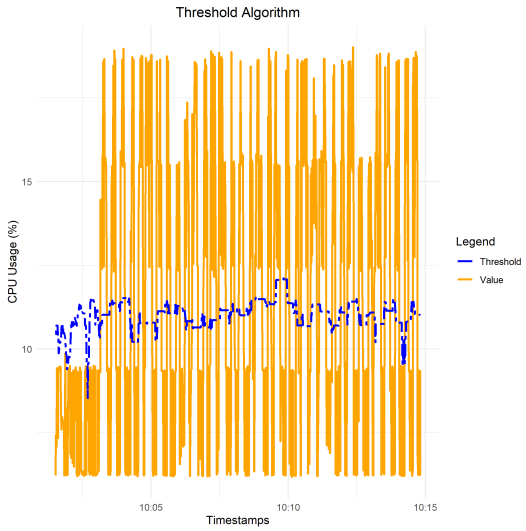

Figure 7.6: Threshold Algorithm with 10 memory and 2*std - Complex Attack Scenario

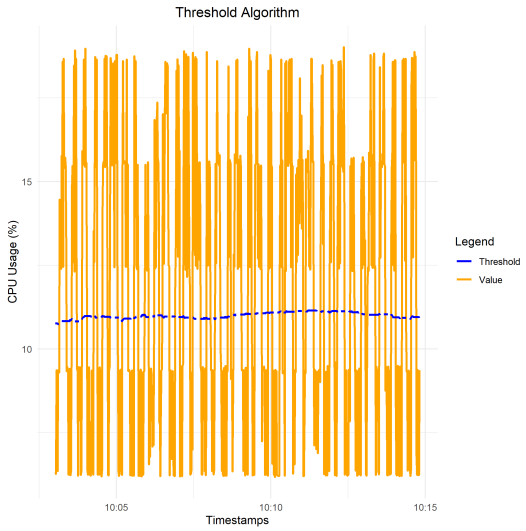

Figure 7.7: Threshold Algorithm with 100 memory and 2*std - Complex Attack Scenario

The figures above show that the successful detection rate of the threshold algorithm is not influenced by the number of previous observations it considers. The threshold is more consistent with 100 previous observations. This however, does not indicate the the algorithm performs better if it looks at the previous 100 observations since the successful detection rates with the two setups are identical:

	Algorithm Setup	SuccessfulDetection	FalsePositive	FalseNegative
1	ThresholdAlgorithm-n1-mem10	0.818	0.169	0.013
2	ThresholdAlgorithm-n2-mem10	0.949	0.031	0.020
3	ThresholdAlgorithm-n3-mem10	0.682	0.007	0.311
4	ThresholdAlgorithm-n4-mem10	0.555	0.001	0.444
5	ThresholdAlgorithm-n1-mem20	0.810	0.177	0.013
6	ThresholdAlgorithm-n2-mem20	0.948	0.033	0.020
7	ThresholdAlgorithm-n3-mem20	0.492	0.000	0.508
8	ThresholdAlgorithm-n4-mem20	0.487	0.000	0.513
9	ThresholdAlgorithm-n1-mem50	0.818	0.169	0.013
10	ThresholdAlgorithm-n2-mem50	0.949	0.031	0.020
11	ThresholdAlgorithm-n3-mem50	0.494	0.000	0.506
12	ThresholdAlgorithm-n4-mem50	0.489	0.000	0.511
13	ThresholdAlgorithm-n1-mem100	0.837	0.150	0.013
14	ThresholdAlgorithm-n2-mem100	0.949	0.031	0.020
15	ThresholdAlgorithm-n3-mem100	0.589	0.004	0.407
16	ThresholdAlgorithm-n4-mem100	0.489	0.000	0.511

Table 7.4: Success rates of the Threshold Algorithm - Complex Attack Scenario

The table above shows that the best performing algorithm setup used 2 as the standard deviation modifier and used 10, 50 and 100 previous observations. In all of these cases the successful detection rate was above 94% with 3.1% false positive and 2% false negative results. This can be explained looking at zoomed-in version of figures 7.6 and 7.7:

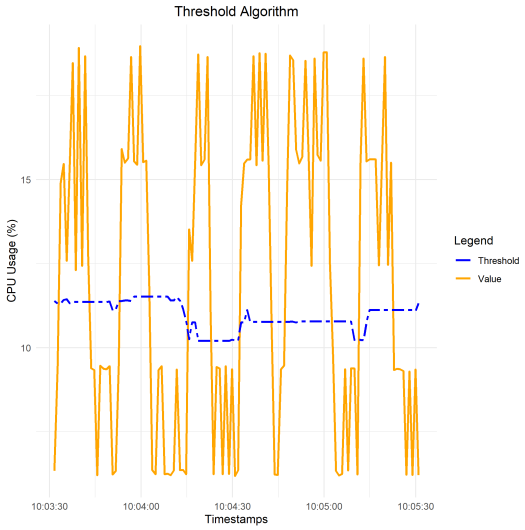

Figure 7.8: Zoom-in on Threshold Algorithm with 10 memory and 2*std - Complex Attack Scenario

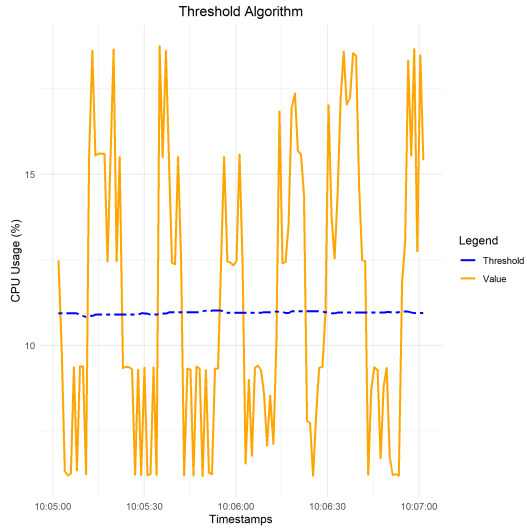

Figure 7.9: Zoom-in on Threshold Algorithm with 100 memory and 2*std - Complex Attack Scenario

From these figures it is clear that while the threshold is not as uniform if the algorithm looks only at the previous 10 observations, its fluctuation does not influence the outcome of the detection hence the two algorithm setups have the same detection rates. In fact, looking at only 10 previous observations will help the system to better adapt to fluctuating usage whereas the version that looks at 100 previous observations will have a harder time with that.

7.1.2.2 Quartiles-based Algorithm

The quartiles-based algorithm behaves very similarly to the threshold algorithm when tested against these more complex attacks from the Mischief Simulator. In this case the best performance regarding successful detection comes from looking at 20 or 100 previous observations with a variable of 0.5:

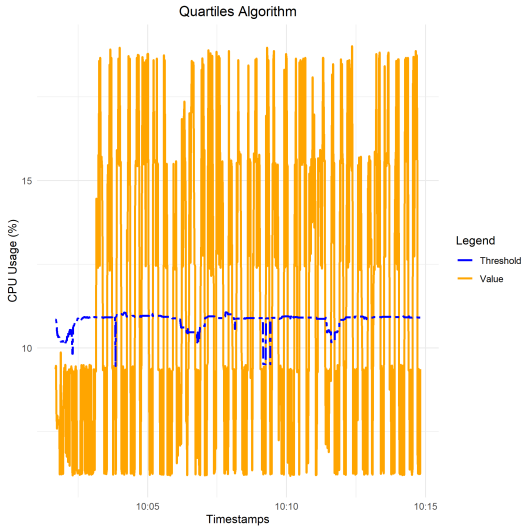

Figure 7.10: Quartiles-based Algorithm with 20 memory and 0.5*std - Complex Attack Scenario

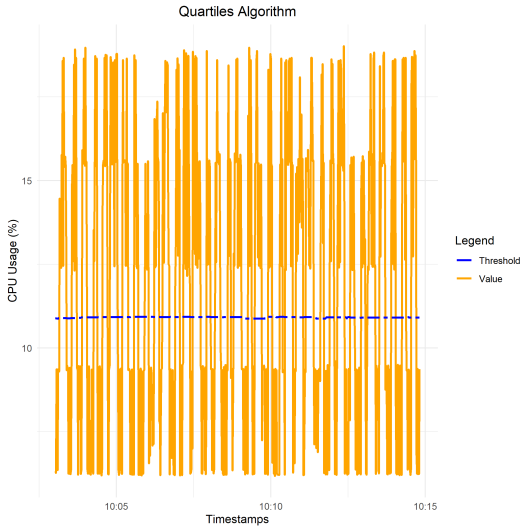

Figure 7.11: Quartiles-based Algorithm with 100 memory and 0.5*std - Complex Attack Scenario

Similarly to the threshold algorithm the quartiles based algorithm also produces a more even threshold line if it looks at the previous 100 observations but the fluctuations of the threshold does not influence the success rate of the detection when the algorithm looks at the 20 previous observations. This is also reflected in the table below:

	Algorithm Setup	SuccessfulDetection	FalsePositive	FalseNegative
1	QuartilesBasedAlgorithm-iqr0.5-mem10	0.570	0.427	0.003
2	QuartilesBasedAlgorithm-iqr1-mem10	0.591	0.404	0.004
3	QuartilesBasedAlgorithm-iqr1.5-mem10	0.604	0.391	0.004
4	QuartilesBasedAlgorithm-iqr2-mem10	0.640	0.356	0.004
5	QuartilesBasedAlgorithm-iqr0.5-mem20	0.949	0.033	0.018
6	QuartilesBasedAlgorithm-iqr1-mem20	0.591	0.004	0.404
7	QuartilesBasedAlgorithm-iqr1.5-mem20	0.504	0.000	0.496
8	QuartilesBasedAlgorithm-iqr2-mem20	0.485	0.000	0.515
9	QuartilesBasedAlgorithm-iqr0.5-mem50	0.949	0.033	0.018
10	QuartilesBasedAlgorithm-iqr1-mem50	0.851	0.020	0.129
11	QuartilesBasedAlgorithm-iqr1.5-mem50	0.579	0.001	0.420
12	QuartilesBasedAlgorithm-iqr2-mem50	0.488	0.000	0.512
13	QuartilesBasedAlgorithm-iqr0.5-mem100	0.949	0.033	0.018
14	QuartilesBasedAlgorithm-iqr1-mem100	0.641	0.007	0.352
15	QuartilesBasedAlgorithm-iqr1.5-mem100	0.590	0.003	0.407
16	QuartilesBasedAlgorithm-iqr2-mem100	0.502	0.000	0.498

Table 7.5: Success rates of the Quartiles-based Algorithm - Complex Attack Scenario

If the algorithm looks at 20, 50 or 100 previous observations it can detect an attack with 94.9% accuracy if the variable is 0.5. The reason as of why are these success rates the same in all three setups is the same as for the threshold algorithm.

7.1.2.3 LOF

When running the more complex attack scenario with the Mischief Simulator we noticed that the more previous observations the LOF algorithm had to work with the better results were obtained. As a consequence we decided to run some additional experiments where we extended the observation window of this algorithm to 200, 300, 400 and 500 observations. The figure below shows the outcome of these extended experiments:

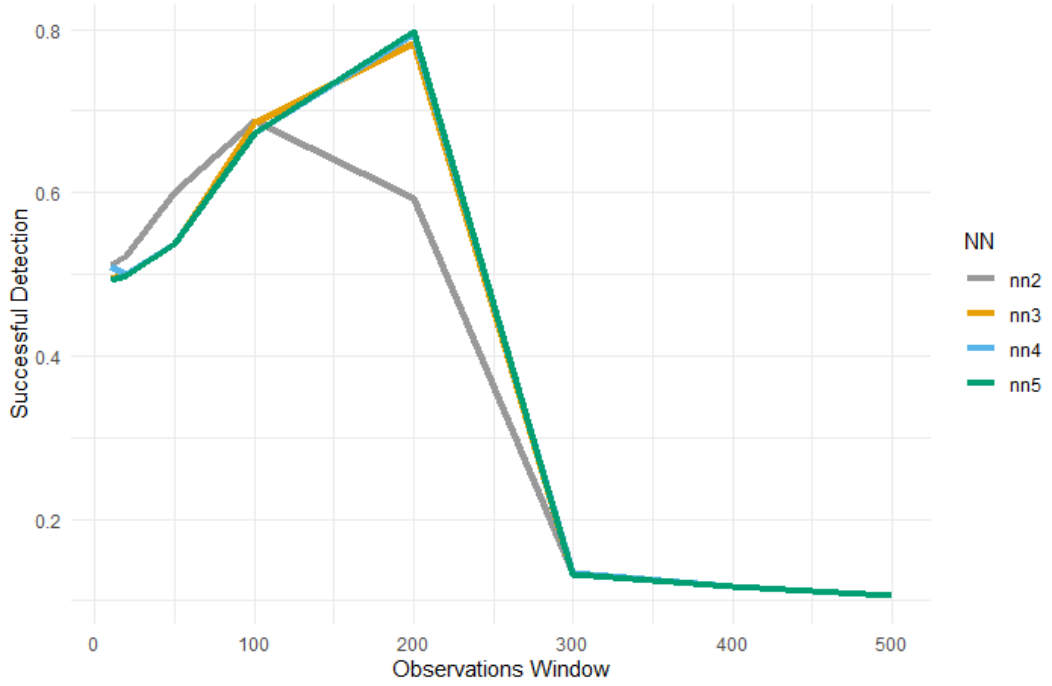

Figure 7.12: LOF Algorithm - Complex Attack Scenario

The figure shows that there is a steady climb upward for the accuracy of the attack detection between 10 and 100 previous observations. The results can be significantly improved with looking at 200 previous observations especially when looking at the distance to the 3^{rd}, 4^{th} and 5^{th} nearest neighbour. More accurate results can be seen in the table below:

	Algorithm Setup	Succ*e*ssfulDetection	Fals*e*Positive	Fals*e*Negative
1	LOFAlgorithm-nn2-mem10	0.511	0.487	0.002
2	LOFAlgorithm-nn3-mem10	0.495	0.505	0.000
3	LOFAlgorithm-nn4-mem10	0.510	0.490	0.000
4	LOFAlgorithm-nn5-mem10	0.492	0.508	0.000
5	LOFAlgorithm-nn2-mem20	0.523	0.477	0.000
6	LOFAlgorithm-nn3-mem20	0.500	0.500	0.000
7	LOFAlgorithm-nn4-mem20	0.500	0.500	0.000
8	LOFAlgorithm-nn5-mem20	0.498	0.502	0.000
9	LOFAlgorithm-nn2-mem50	0.601	0.395	0.003
10	LOFAlgorithm-nn3-mem50	0.538	0.462	0.000
11	LOFAlgorithm-nn4-mem50	0.538	0.462	0.000
12	LOFAlgorithm-nn5-mem50	0.538	0.462	0.000
13	LOFAlgorithm-nn2-mem100	0.688	0.297	0.015
14	LOFAlgorithm-nn3-mem100	0.686	0.301	0.013
15	LOFAlgorithm-nn4-mem100	0.673	0.307	0.020
16	LOFAlgorithm-nn5-mem100	0.673	0.317	0.010
17	LOFAlgorithm-nn2-mem200	0.594	0.225	0.182
18	LOFAlgorithm-nn3-mem200	0.782	0.135	0.083
19	LOFAlgorithm-nn4-mem200	0.793	0.129	0.077
20	LOFAlgorithm-nn5-mem200	0.797	0.128	0.076
21	LOFAlgorithm-nn2-mem300	0.134	0.502	0.364
22	LOFAlgorithm-nn3-mem300	0.132	0.502	0.366
23	LOFAlgorithm-nn4-mem300	0.134	0.502	0.364
24	LOFAlgorithm-nn5-mem300	0.132	0.502	0.366
25	LOFAlgorithm-nn2-mem400	0.118	0.486	0.396
26	LOFAlgorithm-nn3-mem400	0.118	0.486	0.396
27	LOFAlgorithm-nn4-mem400	0.118	0.486	0.396
28	LOFAlgorithm-nn5-mem400	0.118	0.486	0.396
29	LOFAlgorithm-nn2-mem500	0.105	0.480	0.414
30	LOFAlgorithm-nn3-mem500	0.105	0.480	0.414
31	LOFAlgorithm-nn4-mem500	0.105	0.480	0.414
32	LOFAlgorithm-nn5-mem500	0.105	0.480	0.414

Table 7.6: Success rates of LOF - Complex Attack Scenario

From the table it is clear the the best performing algorithm setup for LOF is when it looks at 200 previous observations with the distance to the 5^{th} nearest neighbour. In this case it detects attacks over 79.7% of the cases. This number is still lower than the successful detection rates of the threshold or the quartiles-based algorithms.

7.1.3 Memory Attacks

The previous experiments focused on a single container metric, CPU. The results proved that this metric is a viable option to consider when trying to detect runtime attacks. We wanted to see if the same stands for other metrics as well, so next step was to conduct some experiments with the Mischief Simulator that looked at memory usage in the containers.

7.1.3.1 Threshold Algorithm

In the case of the threshold algorithm the best results were produced when the standard deviation modifier was 2 and the algorithm looked at the 20 previous observations:

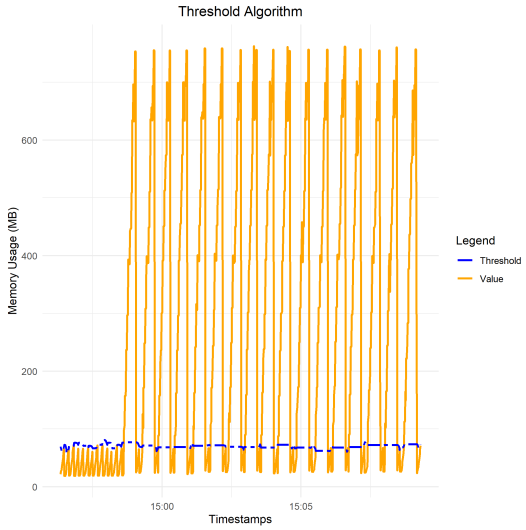

Figure 7.13: Threshold Algorithm with 20 memory and 2*std - Memory Attack Scenario

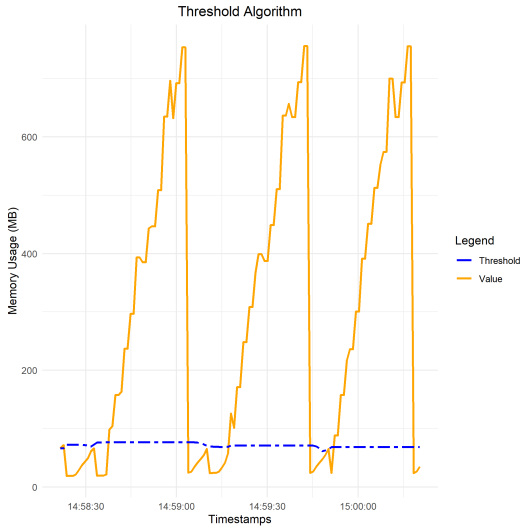

Figure 7.14: Zoom in on Threshold Algorithm with 20 memory and 2*std - Memory Attack Scenario

The figure on the left shows the performance of the algorithm over a longer period of time, while the one on the right is a more zoomed-in version that shows more precisely how the memory values change during the attack. It is interesting to note that the algorithm performed the best when the standard deviation modifier was 2, regardless of the number of previous observations. In every case where the modifier was 2 the successful detection rate was above 81%. This can also be seen in the table below:

	Algorithm Setup	SuccessfulDetection	FalsePositive	FalseNegative
1	ThresholdAlgorithm-n1-mem10	0.722	0.278	0.000
2	ThresholdAlgorithm-n2-mem10	0.811	0.086	0.103
3	ThresholdAlgorithm-n3-mem10	0.809	0.078	0.112
4	ThresholdAlgorithm-n4-mem10	0.675	0.060	0.265
5	ThresholdAlgorithm-n1-mem20	0.798	0.118	0.083
6	ThresholdAlgorithm-n2-mem20	0.815	0.082	0.103
7	ThresholdAlgorithm-n3-mem20	0.811	0.078	0.111
8	ThresholdAlgorithm-n4-mem20	0.308	0.005	0.688
9	ThresholdAlgorithm-n1-mem50	0.795	0.118	0.086
10	ThresholdAlgorithm-n2-mem50	0.814	0.080	0.106
11	ThresholdAlgorithm-n3-mem50	0.814	0.078	0.108
12	ThresholdAlgorithm-n4-mem50	0.437	0.028	0.535
13	ThresholdAlgorithm-n1-mem100	0.795	0.118	0.086
14	ThresholdAlgorithm-n2-mem100	0.811	0.082	0.108
15	ThresholdAlgorithm-n3-mem100	0.798	0.078	0.123
16	ThresholdAlgorithm-n4-mem100	0.645	0.068	0.288

Table 7.7: Success rates of the Threshold Algorithm - Memory Attack Scenario

7.1.3.2 Quartiles-based Algorithm

The quartiles-based algorithm performed the best in the case of memory attacks when it had access to the previous 100 observations with a modifier of 1:

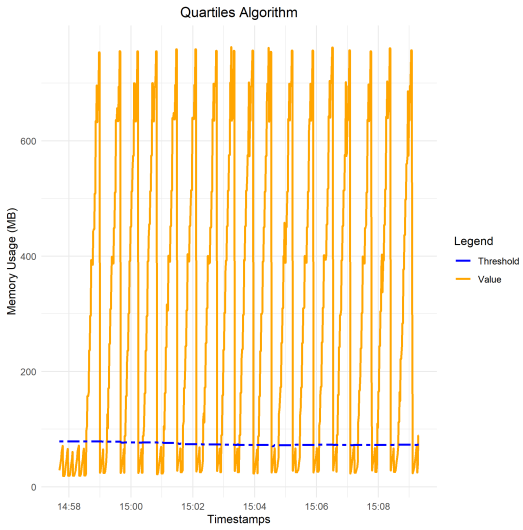

Figure 7.15: Quartiles-based Algorithm with 100 memory and a modifier of 1 - Memory Attack Scenario

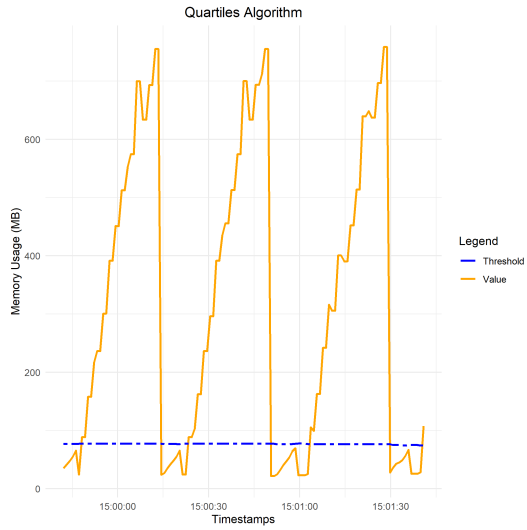

Figure 7.16: Zoom in on Quartiles-based Algorithm with 100 memory and a modifier of 1 - Memory Attack Scenario

The graph on the left shows the performance of the algorithm during a longer time frame, while the one on the right presents a closer look at what is exactly happening with the memory consumption, based on a finer grained time frame. The successful detection rate of the different algorithm setups for the quartiles-based algorithm can be seen below:

	Algorithm Setup	SuccessfulDetection	FalsePositive	FalseNegative
1	QuartilesBasedAlgorithm-iqr0.5-mem10	0.720	0.280	0.000
2	QuartilesBasedAlgorithm-iqr1-mem10	0.794	0.191	0.015
3	QuartilesBasedAlgorithm-iqr1.5-mem10	0.849	0.128	0.023
4	QuartilesBasedAlgorithm-iqr2-mem10	0.888	0.077	0.035
5	QuartilesBasedAlgorithm-iqr0.5-mem20	0.809	0.171	0.020
6	QuartilesBasedAlgorithm-iqr1-mem20	0.966	0.003	0.031
7	QuartilesBasedAlgorithm-iqr1.5-mem20	0.611	0.000	0.389
8	QuartilesBasedAlgorithm-iqr2-mem20	0.280	0.000	0.720
9	QuartilesBasedAlgorithm-iqr0.5-mem50	0.814	0.160	0.026
10	QuartilesBasedAlgorithm-iqr1-mem50	0.968	0.003	0.029
11	QuartilesBasedAlgorithm-iqr1.5-mem50	0.968	0.003	0.029
12	QuartilesBasedAlgorithm-iqr2-mem50	0.938	0.000	0.062
13	QuartilesBasedAlgorithm-iqr0.5-mem100	0.912	0.060	0.028
14	QuartilesBasedAlgorithm-iqr1-mem100	0.968	0.003	0.029
15	QuartilesBasedAlgorithm-iqr1.5-mem100	0.949	0.003	0.048
16	QuartilesBasedAlgorithm-iqr2-mem100	0.922	0.000	0.078

Table 7.8: Success rates of the Quartiles-based Algorithm - Memory attack scenario

The table shows that the setup with 100 previous observation and 1 modifier has over 96.8% successful detection rate, with 0.3% false positive and 2.9% false negative. The algorithm produces the same results when it looks at the previous 50 observations with a modifier of 1 or 1.5.

7.1.3.3 LOF

The successful detection rate of the LOF algorithm can be seen in the table below:

	Algorithm Setup	Succ*e*ssfulDetection	Fals*e*Positive	Fals*e*Negative
1	LOFAlgorithm-nn2-mem10	0.720	0.280	0.000
2	LOFAlgorithm-nn3-mem10	0.720	0.280	0.000
3	LOFAlgorithm-nn4-mem10	0.720	0.280	0.000
4	LOFAlgorithm-nn5-mem10	0.720	0.280	0.000
5	LOFAlgorithm-nn2-mem20	0.722	0.278	0.000
6	LOFAlgorithm-nn3-mem20	0.720	0.280	0.000
7	LOFAlgorithm-nn4-mem20	0.720	0.280	0.000
8	LOFAlgorithm-nn5-mem20	0.720	0.280	0.000
9	LOFAlgorithm-nn2-mem50	0.758	0.226	0.015
10	LOFAlgorithm-nn3-mem50	0.749	0.238	0.012
11	LOFAlgorithm-nn4-mem50	0.743	0.246	0.011
12	LOFAlgorithm-nn5-mem50	0.734	0.255	0.011
13	LOFAlgorithm-nn2-mem100	0.829	0.145	0.026
14	LOFAlgorithm-nn3-mem100	0.822	0.154	0.025
15	LOFAlgorithm-nn4-mem100	0.823	0.152	0.025
16	LOFAlgorithm-nn5-mem100	0.825	0.151	0.025

Table 7.9: Success rates of LOF - Memory Attack Scenario

The best setup for detecting the attacks was when the algorithm looked at 100 previous observations and the distance to the 2^{nd} nearest neighbour. In this case the detection rate was above 82.9%. In all cases, where the algorithm was allowed to look at the previous 100 observations the successful detection rate was around 82%. It is noteworthy that LOF had a minimum a 72% successful detection rate when the attacks specifically targeted memory.

7.2 Experiments with Falco and Threat Simulator

The previous experiments showed that the algorithms are capable of detecting suspicious behaviour by looking at CPU and memory usage. The experiments show that these algorithms work well within our simple experiment setup. Real-life scenarios however are more complex than what was simulated with the Mischief Simulator. Our goal with this thesis was not to create a new tool for runtime container security but to enhance an existing one. The previously described experiment results serve as proof of concept that the simpler algorithms are indeed useful when detecting attacks during runtime. The next step was to test the algorithms in a more complex setup simulating attacks with the Threat Simulator and comparing the results with the output of Falco. For this, the complete setup shown in figure 6.2 was deployed on the VM. The goal of these experiments is to simulate the three selected

attacks described in chapter 6.4.2, correlate the alerts of Falco with the results of the algorithms and raise the priority level of the alerts sent by Falco when relevant.

7.2.1 Suspicious User Behaviour

As described in the previous chapter, 6.4.2, suspicious user behaviour is a collection of nine attacks. According to the results from Threat Simulator the cloud setup is vulnerable to these kind of attacks since seven out of the nine attacks were successful. This indicates that there is a definite need for runtime security measures.

The following figures show the best performing algorithm setup and marks the Falco alerts too:

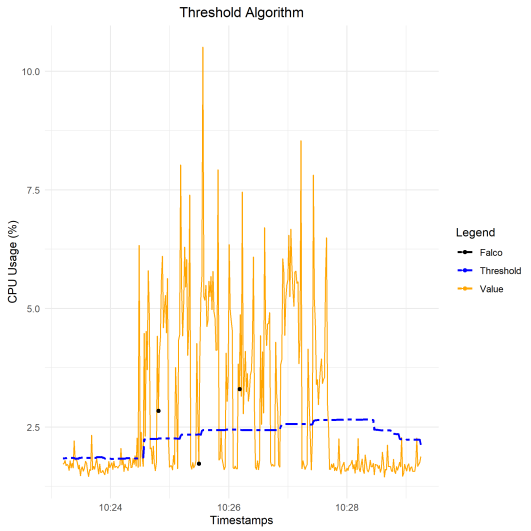

Figure 7.17: Suspicious User Behaviour - Threshold Algorithm and Falco

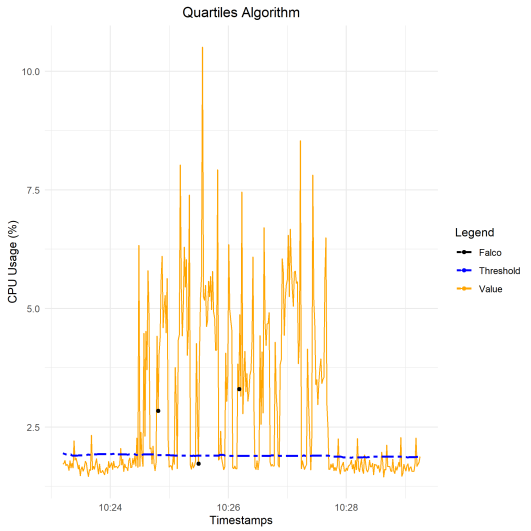

Figure 7.18: Suspicious User Behaviour - Quartiles-based Algorithm and Falco

The threshold algorithm from above looks at the previous 100 observations and has a modifier of 1 for the standard deviation. The quartiles-based algorithm also takes the previous 100 observation into account and uses a 1.5 modifier. The black dots on the graph indicate where Falco notification were sent out. For the nine attacks in this scenario, Falco sent out three alerts in total, all with the severity level of *notice*. Given the fact that two of the Falco notifications happened above the threshold, the Container Overseer can increase the level of severity of these two alerts from notice to *warning* which will be more likely to be looked at by an administrator. The same result is captured by both algorithms.

7.2.2 Malicious File Transfer

During the Malicious Media File transfer scenario Threat Simulator ran 15 attacks. The current security measures on Keysight's private cloud are not equipped to handle these kind of attacks. The results from Threat Simulator show that all 15 attacks were successful and the system got compromised by the drive-by attacks.

Results from the threshold algorithm and quartiles-based algorithm can be seen below:

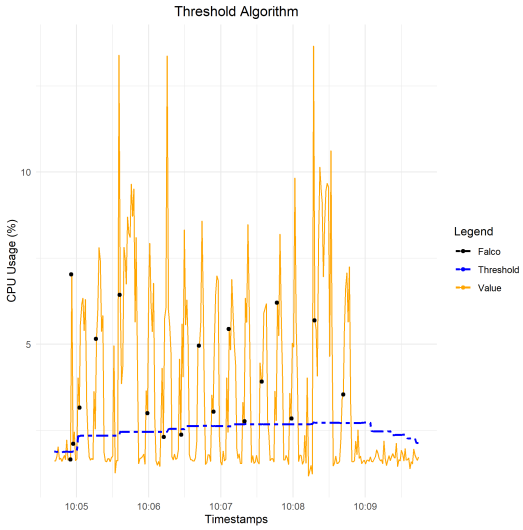

Figure 7.19: Malicious File Transfer - Threshold Algorithm and Falco

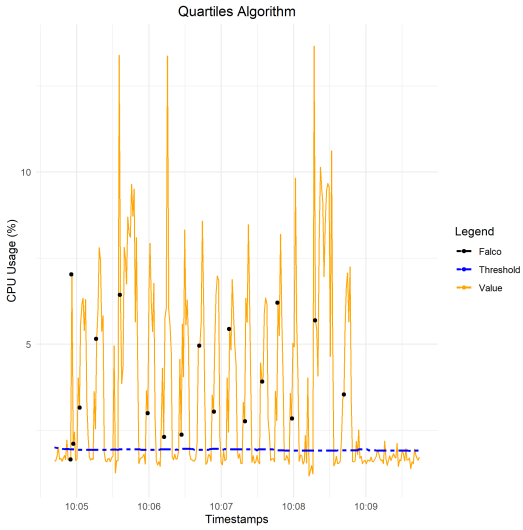

Figure 7.20: Malicious File Transfer - Quartiles-based Algorithm and Falco

From these figures it can be seen that Falco was much more receptive to these kind of attacks and sent more alerts than in the case of the suspicious user behaviour attacks. However, the severity levels of these alerts were still *notice*.

In the case of the threshold algorithm the best result was produced when looking at 100 observation with a 1 as the standard deviation modifier. Based on these results the severity levels of the Falco alerts were increased 15 out of 18 times.

The best performing setup for the quartiles-based algorithm was similarly looking at 100 observations with a 1.5 modifier. In this case 17 out of the 18 alerts were above the threshold so the severity level of these was increased to at least *warning*.

7.2.3 Sunburst

Sunburst is a supply chain attack that gained focus in 2020. As described in chapter 6.4.2, Threat Simulator simulates this attack in two steps: an installation step and a command and control step. Based on the results from Threat Simulator it is clear that the Keysight cloud has no measures in place to stop such an attack. Both the installation step and command and control step was allowed, the scenario successfully compromising the system. Our experiments show the following results regarding this attack:

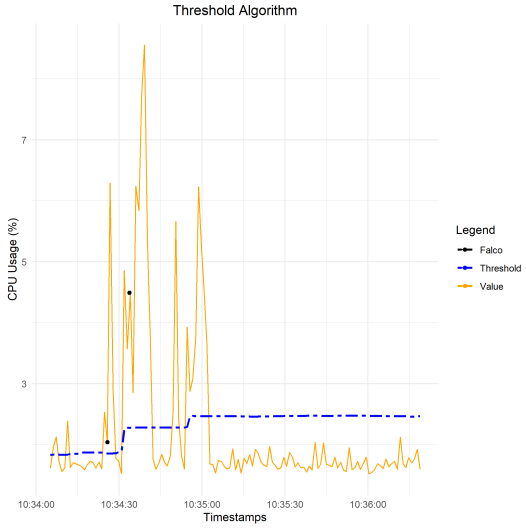

Figure 7.21: Sunburst - Threshold Algorithm and Falco

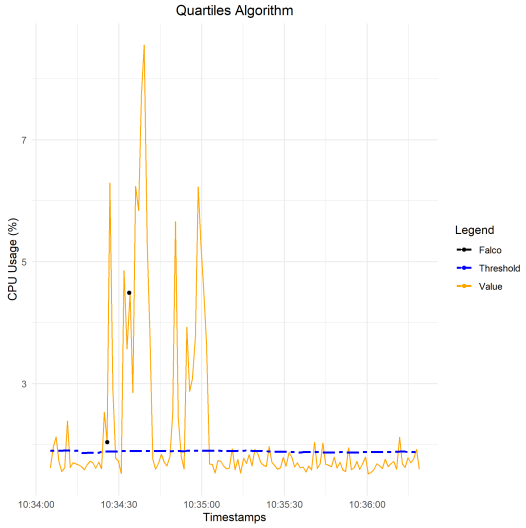

Figure 7.22: Sunburst - Quartiles-based Algorithm and Falco

The figures show that Falco correctly picked up on the two steps of the attack sending out a notice level alert for both. These alerts were above the threshold for both the threshold and the quartiles-based algorithm so the Container Overseer increased the level of severity of both of these Falco alerts. Similarly to the algorithm setups of the previous two Threat Simulator experiments, both algorithms looked at 100 observations with a standard deviation modifier of 1 in the case of the threshold algorithm and 1.5 modifier in the case of the quartile-based algorithm.

The results discussed in this section only look at CPU usage as the container metric. This is because the attacks simulated with the Threat Simulator did not have a significant impact on memory usage. The changes in memory usage were so small that the algorithms were not able to establish patterns and learn to differentiate between attack and non-attack states.

Moreover, the LOF algorithm was not tested with the Threat Simulator attacks because of license issues with the tool. This part of the experiments remain for future work.

Chapter 8

Limitations and Future Work

In this chapter limitations of the current work and possible future work is discussed. This part considers what are the shortcoming and possible issues with the current experiment setup and algorithms used as well as how could they be improved or what future experiments could provide more valuable data on the topic of this thesis.

One of the limitations of the current results is that the experiments are limited to a rather simple setting. Even an experiment that is described as complex has a clear pattern and its behaviour can be easily predicted, which means that in real systems, especially systems that perform computationally intensive tasks, the results are expected to be worse than they are in the current experiments. This leads to one of the considerations for future work, and that is to test the current experiments on an actual system, where it would be possible to more accurately investigate the detection limits of the current algorithms. This kind of investigation would allow to further understand the limitations of the currently implemented algorithms and could provide valuable information on how they could be improved or what algorithms would be more suitable for the task. In practice, this was already considered by the group and steps for adding necessary tools to an existing system on a Kubernetes cluster were investigated, therefore this part of future work could be considered to be a very important next step, since it would allow to further validate and possibly consolidate current experiment results.

Another important consideration is that the current experiments focused on only two computer resources, CPU usage and RAM usage. Although these two resources provide a lot of information and during the experiments allowed the group to explore how the differences between them affect the algorithm's successful detection rates, it is important to note that it is difficult to argue why the algorithms should be

limited to only these resources. Therefore, another addition to this project that could improve the results is monitoring of more kind of computer resources such as read and write disk bandwidth usage and network bandwidth usage. This idea could be further expanded by using different methods do combine information provided by different computing resource metrics in order to improve successful detection rates.

The idea of adding other algorithms to the experiments is also very important. The current algorithms provide very clear and easily interpreted results which in turn allow to understand which changes in container metrics result in a decrease in detection performance. While this helped a lot with the current experiment, the group thinks that this is not the best performance that can be achieved, and that the addition of more sophisticated algorithms and techniques would allow to further improve the performance.

Another limitation in the current project is the lack of experiment results from attack simulation using Threat Simulator for the LOF algorithm. This limitation appears because license for Threat Simulator expired during the project. This experiment could give valuable information about the performance of this algorithms and since all the other steps for this experiment were already prepared, it is not difficult or time consuming to perform this experiment after solving the licensing issues. Due to these reasons, solving this limitation can be considered an immediate next step in the project.

This chapter focused on analyzing current limitations of this work, looking into the shortcoming of the current experiments and algorithms as well as proposing possible future improvements that would allow to mitigate current limitation of the project and its results or even open paths to new investigation into the topic.

Chapter 9

Conclusion

This thesis focused on runtime container security in a private cloud. In chapter 1.1 we formulated a hypothesis that stated that runtime container security of cloud based containers can be improved by the use of an ensemble tool set that consists of a freely available, open-source runtime security tool, Falco and the Container Overseer, a tool developed by the writers of this thesis, that detects malicious attacks based on the CPU and memory usage of the containers.

In order to prove this hypothesis, the thesis approached the topic from two angles: a theoretical angle and practical angle. It started of with laying down the theoretical foundations of cloud computing, containers and container security. It introduced the different cloud deployment and service models, discussed the top 11 threats of cloud computing and provided an overview of cloud security strategies.

Following this we turn our attention to containers, talking about containerisation and virtualization, the differences between containers and virtual machines and the benefits and drawbacks of containers.

The next step in this theoretical exploration discussed container security, presenting the most common container threats and delved into runtime container security strategies and their importance. The final stop in the theoretical journey talked about runtime security tools and introduced Falco.

From a practical angle we conducted several experiments on Keysight's private cloud. On this cloud we deployed the Container Overseer, the Mischief Simulator, Falco, Falco Sidekick and a Threat Simulator Agent. The experiments in the first phase aimed at proving the effectiveness of the three algorithms used for anomaly detection in the Container Overseer. We ran a series of simple and complex attacks

with the Mischief Simulator looking at CPU usage as well as a series of attacks that looked at memory usage. Following this we compiled the successful detection rate of the algorithms. The findings show that:

- **Threshold Algorithm**: In the case of the simple and complex attacks the number of previous observations only slightly influenced the successful detection rate of the algorithm. In the case of the simple attacks the best results are reached when the standard deviation modifier is 4, while for the complex attack when it is 2.

  In the case of attacks specifically targeting memory the best performing setup for this algorithm was with the previous 20 observations and 2 as the standard deviation modifier.

  In the case of the attack scenarios simulated by the Threat Simulator the best setup for this algorithm turned out be the one where it looked at previous 100 observations with a standard deviation modifier of 1. This allowed the algorithm to detect issues at the same time as Falco, thus it was able to fulfill its role of supporting the increase of the priority level of the Falco alerts.

- **Quartiles-based Algorithm**: The best successful detection rate for this algorithm, in the case of the simple attacks was when the algorithm looked at the previous 100 observations with a variable of 2. For the complex attacks the best results were achieved when the algorithm looked at more than 20 previous observations with a variable of 0.5.

  In the case of the memory attacks the algorithm needs to look at over 50 previous observations with a modifier of 1 or 1.5.

  The real life attacks, simulated by the Threat Simulator showed that the algorithm performs remarkably well regardless of setup and is able to detect problems at the same time as Falco, supporting the increase of the priority level of the Falco alerts.

- **LOF**: Out of the three algorithms LOF had the worst successful detection rate in most cases. In the case of the simple attacks the best performance was achieved when it looked at 100 previous observations and considered the distance to the 2^{nd} nearest neighbour. The more complex attacks showed that this algorithm performs better if it can look at the previous 200 observations.

  In the case of the memory attacks the algorithm performed the best when it had the same setup as for the simple attacks: 100 previous observations and the distance to the 2^{nd} nearest neighbour.

Overall, the experiments show the the Quartiles-based algorithm is the most successful working together with Falco, followed by the threshold algorithm and fi-

nally LOF. Based on the experiemnts conducted in this thesis it can be concluded that the use of simple algorithms to detect anomalies in container metrics can help in improving the results from Falco, thus the hypothesis from chapter 1.1 stands.

Bibliography

[1] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos. "Security in cloud computing: Opportunities and challenges". In: *Information Sciences* 305 (2015), pp. 357–383. ISSN: 00200255. DOI: 10.1016/j.ins.2015.01.025. URL: http://dx.doi.org/10.1016/j.ins.2015.01.025.

[2] Bandar Ibrahim M Altariqi et al. "5G Core & (NFVI) Network Functions Virtualization Infrastructure Penetration Testing Simulating an Inside Cloud Attack". In: ().

[3] Domantas Astrauskas and Vivienne Spence. *Cloud Security in the 5G-VINNICloud*. Tech. rep. Aalborg: Aalborg University, 2020, p. 88.

[4] Markus M Breunig et al. *LOF: Identifying Density-Based Local Outliers*. Tech. rep. Dalles, 2000.

[5] Ginger Collison. *Introduction to NATS*. 2021. URL: https://docs.nats.io/.

[6] *Docker Engine overview | Docker Documentation*. URL: https://docs.docker.com/engine/.

[7] *Docker overview | Docker Documentation*. URL: https://docs.docker.com/get-started/overview/.

[8] Gary Duan. *What does runtime container security really mean?* 2019. URL: https://www.helpnetsecurity.com/2019/06/17/runtime-container-security/.

[9] Nikita Duggal. *Advantages and Disadvantages of Cloud Computing*. 20201. URL: https://www.simplilearn.com/advantages-and-disadvantages-of-cloud-computing-article.

[10] Frederick Fernando. *An Introduction to Kubernetes Security using Falco*. 2021. URL: https://falco.org/blog/intro-k8s-security-monitoring/.

[11] Markus Goldstein and Seiichi Uchida. "A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data". In: *PLOS ONE* 11.4 (Apr. 2016). Ed. by Dongxiao Zhu, e0152173. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0152173. URL: https://dx.plos.org/10.1371/journal.pone.0152173.

[12] Grand View Research. "Application Container Market Analysis Report By Deployment, By Platform, By Organization Size, By Service, By Application, By Region And Segment Forecasts From 2019 To 2025". In: *Million Insights* (2020).

[13] Fei Huang. *15 Tips for a Run-time Container Security Strategy*. 2016. URL: https://blog.neuvector.com/article/15-tips-run-time-container-security-strategy.

[14] IBM Cloud Education. *Containerization Explained | IBM*. URL: https://www.ibm.com/cloud/learn/containerization.

[15] IBM Cloud Education. *The Benefits of Containerization and What It Means for You | IBM*. URL: https://www.ibm.com/cloud/blog/the-benefits-of-containerization-and-what-it-means-for-you.

[16] IBM Cloud Education. *What are Containers | IBM*. URL: https://www.ibm.com/cloud/learn/containers.

[17] IBM Cloud Team. *Containers vs. Virtual Machines (VMs): What's the Difference? | IBM*. URL: https://www.ibm.com/cloud/blog/containers-vs-vms.

[18] Mohammad Saiful Islam and Andriy Miranskyy. "Anomaly detection in cloud components". In: *IEEE International Conference on Cloud Computing, CLOUD* 2020-Octob.January 2021 (2020), pp. 31–33. ISSN: 21596190. DOI: 10.1109/CLOUD49709.2020.00008.

[19] Muhammad Adeel Javaid. "Top Threats to Cloud Computing Security: The Egregious Eleven". In: *SSRN Electronic Journal* (2018). ISSN: 1556-5068. URL: https://s3.amazonaws.com/content-production.cloudsecurityalliance/5qunuas8cakrmyxjekm1qaea0hy3?response-content-disposition=inline%3Bfilename%3D%22The-Egregious-11-Cloud-Computing-Top-Threats-in-2019-April2020.pdf%22%3Bfilename%2A%3DUTF-8%27%27The-Egreg.

[20] Keysight Technologies. *Threat Simulator - Getting Started*. 2021. URL: https://threatsimulator.cloud/gs-help/Default.htm.

[21] Keysight Technologies. *Threat Simulator - User Guide*. 2021. URL: https://threatsimulator.cloud/help/security/TS_UG_PDF.pdf.

[22] Thomas Labarussias. *Extend Falco outputs with falcosidekick*. 2020. URL: https://falco.org/blog/extend-falco-outputs-with-falcosidekick/.

[23] Andrew Larkin. *Disadvantages of Cloud Computing*. 2019. URL: https://cloudacademy.com/blog/disadvantages-of-cloud-computing/.

[24] Jesus Lobo. *Detecting real-time and unsupervised anomalies in streaming data: a starting point*. 2020. URL: https://towardsdatascience.com/detecting-real-time-and-unsupervised-anomalies-in-streaming-data-a-starting-point-760a4bacbdf8.

[25] Lockheed Martin. *The Cyber Kill Chain*. 2021. URL: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[26] Ajitabh Mahalkari, Avni Tailor, and Aniket Shukla. "Cloud Computing Security, Defense In Depth Detailed Survey". In: *International Journal of Computer Science and Information Technologies* 7.3 (2016), pp. 1145–1151. URL: http://ijcsit.com/docs/Volume7/vol7issue3/ijcsit2016070326.pdf.

[27] Microsoft Azure. *What is cloud computing?* 2021. URL: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.

[28] Rich Mogull et al. "Security-Guidance-v4-FINAL". In: (2017).

[29] OpenStack. *OpenStack*. 2021. URL: https://www.openstack.org/.

[30] Rani Osnat. *A Brief History of Containers: From the 1970s Till Now*. 2020. URL: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016#:~:text=2006%3AProcessContainers,ofacollectionofprocesses..

[31] OWASP. *Cross Site Scripting (XSS)*. 2021. URL: https://owasp.org/www-community/attacks/xss/.

[32] Sam Solutions. *4 Best Cloud Deployment Models Overview*. 2020. URL: https://www.sam-solutions.com/blog/four-best-cloud-deployment-models-you-need-to-know/.

[33] Cliff Saran. "Heartbleed and Shellshock thriving in Docker community". In: *ComputerWeekly.com* (2018). URL: https://www.computerweekly.com/news/252437100/Heartbleed-and-WannaCry-thriving-in-Docker-community.

[34] Vasilios A. Siris and Fotini Papagalou. "Application of anomaly detection algorithms for detecting SYN flooding attacks". In: *Computer Communications* 29.9 (May 2006), pp. 1433–1442. ISSN: 01403664. DOI: 10.1016/j.comcom.2005.09.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0140366405003531.

[35] Mohammad Reza Khalifeh Soltanian and Iraj Sadegh Amiri. "Theoretical and experimental methods for defending against DDoS attacks". In: *Advanced topics in information security* 2016 ().

[36] Sari Sultan and Tassos Dimitriou. "Container Security: Issues, Challenges, and the Road Ahead". In: *IEEE Access* (2019).

[37] Sysdig. *Falco*. 2021. URL: https://sysdig.com/opensource/falco/.

[38] Pallavi Varanasi. *What is Container In Cloud Computing*. 2020. URL: https://www.cloudcodes.com/blog/container-in-cloud-computing.html.

[39] Xili Wan et al. "Application deployment using Microservice and Docker containers: Framework and optimization". In: *Journal of Network and Computer Applications* 119 (Oct. 2018), pp. 97–109. ISSN: 10958592. DOI: 10.1016/j.jnca.2018.07.003.

[40] *What are Containers and their benefits | Google Cloud*. URL: https://cloud.google.com/containers.

[41] Yuping Xing and Yongzhao Zhan. "Virtualization and Cloud Computing". In: 2012, pp. 305–312. DOI: 10.1007/978-3-642-27323-0{_}39. URL: http://link.springer.com/10.1007/978-3-642-27323-0_39.

[42] Zhuping Zou et al. "A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest". In: *IEEE Transactions on Cloud Computing* July 2020 (2019), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/tcc.2019.2935724.