HAAUKINS

Optimizing performance of virtualized networking

Master thesis

Robert Nedergaard Nielsen

AALBORG UNIVERSITY

STUDENT REPORT

Aalborg University
Communication technologies
Networks and Distributed Systems

Communication technologies
Aalborg University
http://www.aau.dk/

AALBORG UNIVERSITY
STUDENT REPORT

Title:
HAAUKINS: Optimizing performance of virtualized networking

Theme:
Communication Systems

Project Period:
Spring 2022

Project Group:

Participant(s):
Robert Nedergaard Nielsen

Supervisor(s):
Jens Myrup Pedersen

Page Numbers: 52

Date of Completion:
June 1, 2022

Abstract:

This thesis examines the problem of packets being lost when users are doing their initial scan on the HAAUKINS platform, developed for facilitating practical exercises in cybersecurity education. The examination found that the Docker macvlan used causes the ARP table on the host to overflow when more than 1024 containers are created.
Therefore, two solutions are proposed. One is to create a script ensuring that the correct dependencies and settings are present for running the platform. The second is to use another virtualization software for the network module og the platform. Both solutions are successfully implemented and tested.
Lastly, the thesis considers how the platform can be further developed to allow more users. This is done by creating a client server architecture and essentially creating a cluster for running the HAAUKINS labs.

Contents

1 Introduction

Lately, the cybersecurity industry has been on high alert, as cyber attacks are becoming more and more sophisticated and frequent. One factor for the increased frequency of cyber attacks is the move to the work form home operation model. Work from home makes it more difficult for the already busy cybersecurity professionals to protect company networks from attacks. The network boundary is no longer just between the internal and public network. Instead, security measures now have to protect against threats coming from unsecure private networks[11].

Other than having a larger frequency, the attacks are also becoming more sophisticated, which is especially seen when looking at phishing attacks[2]. Phishing sites are beginning to look even closer to the original pages, and the translation done in phishing emails is becoming more and more natural, making it harder for users to distinguish between phishing emails and actual emails. This does not only apply to phishing emails and websites but also phishing calls and other message types[1]. Studies also show that people working from home is more likely to make mistakes and click on phishing links[14].

The increased frequency and sophistication of cyber attacks have created a need for more cybersecurity professionals. The (ISC)²s yearly study of the cybersecurity workforce shows that the global demand for cybersecurity professionals continues to outpace the rate they can be educated[7]. In order to close this gap, Aalborg University Denmark (AAU) launched the first danish master's in cybersecurity back in 2020[17].

In parallel with the creating the curriculum for this education, the HAAUKINS platform was created to facilitate practical exercises, which are essential for such an education. The reason for creating a new platform instead of using one of the existing platforms is that none of them automates the process of setting up custom environments for teaching classes.

When looking at commercial options such as Hack the Box (HtB) and Try Hack Me (THM), HtB targets more experienced users with more complex scenarios. THM provides several paths targeted for the same audience as HAAUKINS. However, both platforms lack the possibilities for teachers to add more vulnerable machines and customize the scenarios to fit their exact curriculum[13]. Based on these statements the first version of HAAUKINS was created with the following design goals[16].

- Fully automated.
- Transparent.
- Highly accessible.
- Realistic.

As the popularity of the HAAUKINS platform increased, the target group also broadened from mainly high school students to higher educations and companies wanting to use it for security training. With the broader audience, it was found that additional requirements were needed to fit the more complex exercises. One was that the tools available should be dynamic. This is achieved by making the labs available through a VPN as well as existing the browser access. This allows the users to use their own machines and tools to attack the vulnerable hosts in the lab without violating the highly accessible goal.

The next addition to the goals is to centralize the exercises developed for HAAUKINS. This would allow teachers to use exercises developed by others as it can be time consuming to create exercises. With a growing number of users on HAAUKINS a scalability goal has also been added, spilt into teaching scalability and technical scalability. For the purpose of this thesis, the technical scalability is the most interesting of the two.

In order to meet the goal, multiple options were examined. The first one was to scale HAAUKINS to run across multiple servers, called horizontal scaling. To accommodate this HAAUKINS has been split into microservices. The second approach was to increase the servers' resources by adding more memory and disk space, which has been applied to the main server running HAAUKINS.

In order to increase the capacity even more, a sleep function was implemented, this function will put containers and Virtual Machine (VM)s into sleep when labs have not been used for a set amount of time. The reason for doing these experiments has shown that events running for a longer period of time, such as the ones running as a part of a course, have longer periods where the labs are not actively being used. The users can then wake the labs back up by simply logging in to the event again.

This scalability goal leads to the problem that this thesis will be trying to solve. A problem that has been observed when running multiple events, and large number of users are doing their initial reconnaissance, namely network scanning, some of their scanning packets are dropped. These packet drops affect the normal operations by making users lose important information from the platform. Initial investigations have shown that it is the ARP packets being dropped, and it is assumed that this is caused by all packets going through the kernel.

This problem with users losing important information is especially problematic in the cases where HAAUKINS is used in competitions like De Danske Cybermesterskaber (DDC). Here the outcome of the competition is highly affected if some users are unable to connect to or see some host during their initial scan and when trying to perform an attack later in the process.

In the past, this issue was an edge case and only seen when running huge events, like the National championship in cybersecurity for high school students. However, with the before mentioned demand for HAAUKINS the issue is commonly seen. Other than using the servers placed at AAU, multiple other universities and companies are starting to request help with deploying HAAUKINS to their infrastructure. This means that

errors like these package drops should be fixed to improve the experience for the users of HAAUKINS. Based on the description of the initial problem a problem statement can be established as:

"How can the networking of HAAUKINS be improved to prevent users from loosing crucial information from the system?"

The next section will describe the method used to approach the process of analyzing and solving the problem condensed in the problem statements above.

1.1 Methodology

This section will provide an overview of the method used for approaching and solving the problem described in Chapter 1. The first step will be to identify the probable causes of the packet drops seen, especially when many users are active. This will be done by doing an initial examination of the HAAUKINS platform itself, its use cases and its architecture. Based on this initial examination, the relevant components will be further examined, and the potential problems will be highlighted.

The second step will then be to test the probable causes found in step one and identify the actual cause of the packet drops. This is done by defining a set of hypotheses and tests that tests these. The results will be presented as Probability Mass Function (PMF)s which will allow for easy comparison between the tests, both the hypotheses tests but also the final test of the implemented solution.

The third step will be to design a solution that solves the problem of packets being dropped while still preserving all the required functionalities. This step will start with establishing the requirements for the solution based on the examination of HAAUKINS in step one, which is then used when assessing the different possible solutions to the problem.

The fourth and last step will be to test how big of an improvement the designed solution has been. This will be done by implementing the solution and running the tests designed for testing the hypotheses in step two.

With the overall problem and the method for approaching the problem explained, the next chapter will be the analysis of the system and the initial testing.

2 Analysis

This chapter will analyze the problem of HAAUKINS dropping packets presented in Chapter 1. The analysis will start with a closer look into the architecture and use of the HAAUKINS platform in Section 2.1. After this, the Docker networking components will be described in Section 2.2. Here the current networking method and other ways that Docker can handle the network traffic are examined, and potential problems identified during the examination are presented in Section 2.2.1.

Based on these examinations, a set of hypotheses is presented and tested in Section 2.3. The results of these tests will be the basis for the design of the solution made in Chapter 3.

2.1 HAAUKINS

This section will examine the HAAUKINS platform, starting with describing how the platform is used from a user perspective, both admins and users in Section 2.1.1. After this, a description of the architecture of HAAUKINS will be made in Section 2.1.2, identifying components that might lead to the packet drops seen.

2.1.1 Using HAAUKINS

To understand the different components of HAAUKINS, an overview of how it is used is needed. The users of HAAUKINS can be divided into two groups the teachers and the students. The teachers are the ones with access to create events, while the students are the users who sign up for these events. When the users sign up for the event,they will be assigned a virtual environment, referred to as a lab, which contains the exercises chosen by the teacher.

The process of setting up these events is done through a webclient component. How this interfaces with the core component of HAAUKINS will be described in Section 2.1.2. When accessing the webclient, the teachers can see, create and manage the events currently running on the server.

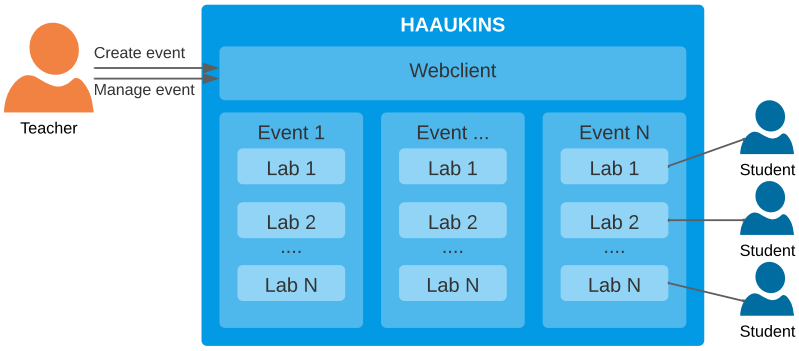


Figure 2.1: Using the HAAUKINS platform

When creating an event, the teachers will be presented with the choice between browser, VPN or a combination; the combination is still in a testing phase. The browser type will allow the students to access a VM through an RDP connection from within a browser. The VPN type will allow students to access their lab using a Wireguard VPN connection[5].

After choosing the event type, the teacher will be requested to choose a tag, capacity and availability of the event. The chosen tag will be added as a subdomain on the server. The subdomain is then used for accessing the frontend of the event. The capacity is the maximum number users that can sign up for the event, while the availability is the amount labs that are ready to be assigned to users signing up. If the browser event is chosen the teacher will be able to choose which VM should be used by the students when accessing the event. The last thing that the teacher should select is the exercises which should be included.

The students can then navigate to the website exposed on the subdomain and sign up to get a lab assigned. After signing up, the exercises chosen for the event can be seen together with a short description of the task. The students can then connect to the lab directly from the browser or by downloading a VPN configuration depending on the event type.

After solving an exercise the student will be presented with a flag, which is a random string which matches the pattern $HKN\backslash\{[[\backslash w]\{2,3\} - [\backslash w]\{2,3\} - [\backslash w]\{4,6\}]\{12\}\backslash\}$. Inserting this flag on the website will reward the user with points according to the difficulty of the exercise. Some of these exercises with many intermediate steps might also reward the user with a flag to lead the user on the right path.

With a description of how HAAUKINS is used by the two different user types, the next section will describe the architecture of HAAUKINS.

2.1.2 HAAUKINS architecture

This section will provide an architectural overview of HAAUKINS in order to find the components that might cause the packet loss. HAAUKINS consists of four components,

or microservices, as seen in Figure 2.2. It has a core component that handles the creation of labs using Docker and VirtualBox (VBox) and setting up a frontend for events.

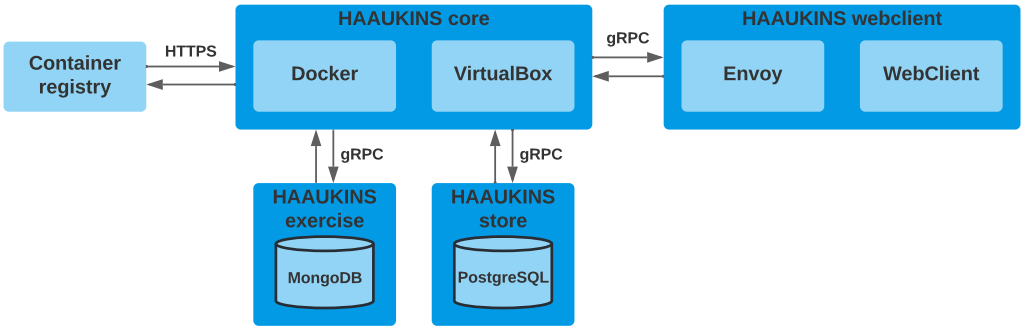


Figure 2.2: Overview of HAAUKINS components

Then it has two storage components, one for storing exercises and one for storing events and teams. The exercise store uses a MongoDB database to storing the exercise details, and the HAAUKINS store uses a PostgresSQL database to storing information about the events and their users. The last component of HAAUKINS is the webclient mentioned in Section 2.1.1, used by teachers to start up the events. The communication between these and the HAAUKINS core is done through gRPC.

Figure 2.3 shows the Docker networks used for the different types of labs HAAUKINS can create. The browser event where users can use their browser to access a VM, like Kali Linux or Parrot OS, through an RDP connection uses a Docker macvlan network. In this case, a macvlan is created, and the Docker containers are connected to it; the frontend VM is then bridged to the macvlan interface created on the host.

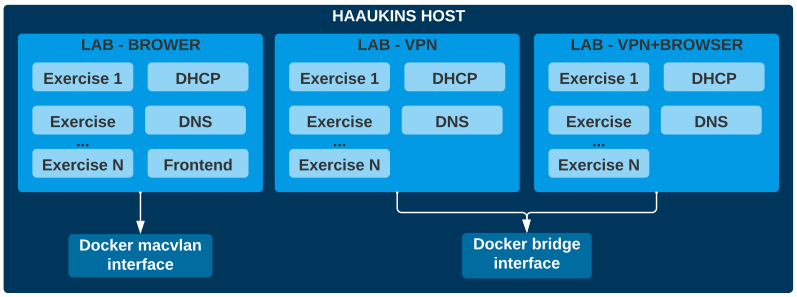


Figure 2.3: Lab network on HAAUKINS

If VPN is enabled in the event, the lab is connected using a Docker bridge network. As with the browser event, a Docker network is created, the exercise containers are connected to this network, and a VM is bridged to the network if it is a combined event.

With the basic architecture for HAAUKINS explained, the next section will examine the Docker networking.

2.2 Docker networking

This section will examine the methods for providing networking to the Docker containers used in the labs. By default, Docker has six network drivers, namely bridge, host, overlay, IPvlan, macvlan and none. As seen in Figure 2.3, the current implementation is using macvlan for networking in events running from the browser and bridge networking for events running with VPN enabled.

As the name suggests, the none network driver disables all networking for the container and is used when the container should have no internet, or it is wished to use a custom networking driver for the container. One custom networking that could be used is Open vSwitch (OVS); in this case, the containers are started with the none driver and attached to the OVS switch afterwards[9].

Bridge networking is the default driver, and if nothing is given, the container will be connected to the default Docker bridge network created when Docker starts. In networking, a bridge is a device that forwards traffic on the link layer; this can be either a physical device or software running in the host's kernel. The Docker bridge network uses a software bridge that allows containers on the same bridge to communicate while blocking communication across different bridges.

The overlay network driver allows Docker to create a distributed network across multiple Docker hosts without having to create routing rules on the host machines. This networking driver requires that the Docker daemon is a part of a Docker Swarm, which is a container orchestration tool.

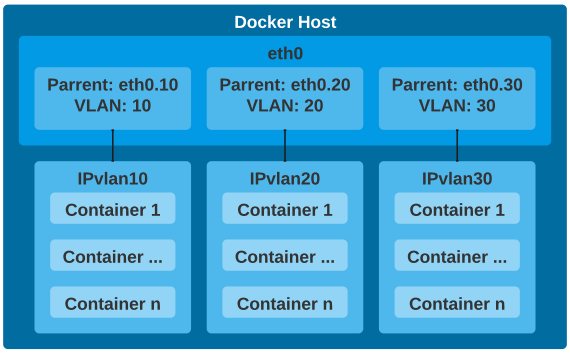


Figure 2.4: Illustration of IPvlan

The IPvlan gives complete control of the IP addresses assigned to the containers and allows for Virtual LAN (VLAN) tagging of the traffic. By default, the routing is handled on its own but can also be manipulated if the use case requires control of the underlying network. The Linux implementation is relatively lightweight as it uses an ethernet interface to create the network isolation instead of using a bridge as with the bridge driver. The IPvlan requires the Linux kernel to be above v4.2 as Docker says that earlier implementations can be buggy. Figure 2.4 shows an example setup using the

802.1q standard with three IPvlans, each of which is connected to a subinterface of the host main interface eth0. Each of these subinterfaces interfaces is dynamically created when creating the IPvlan.

The macvlan network driver assigns a MAC address to the container's network interface, making it appear as a physical network device. Using this networking mode requires a parent interface to be given and that the interface supports having multiple MAC addresses assigned. If no interface is given to the command when creating the Docker network, which is the case in the current implementation in HAAUKINS. Docker will create a Linux dummy interface mainly used for tests and development environments[8].

The last network driver is the host driver; when this driver is used, there is no isolation between the container network and the host network, which means that ports exposed by the container will be exposed directly by the host.

With the networking methods described, the next section will examine some of the problems found while examining the types of networking.

2.2.1 Problems identified

This section will describe the problems identified when examining the different ways of handling networking for Docker containers. The problems are mainly identified in the context of HAAUKINS, and some are the best practices mentioned by Docker.

Starting with bridge networking it was found that it can cause unicast flooding generating a large amount of traffic. Unicast flooding is a case where the networking bridge does not know the packet's destination and therefore sends it out on all connected interfaces. Unicast flooding can be caused by multiple things, such as the ARP table being full, a mismatch between the ARP timeout and the timeout in the ARP table or simply that the ARP table does not contain the destination MAC.

In order to populate these tables, the ARP protocol is used. When looking at the the initial examination, it was found that some of the ARP packets are disappearing. Further investigations show that when Docker sends ARP traffic, it floods the ARP packet to all ports connected to the bridge[19]. This forces the kernel to process each packet as many times as there are hosts in the network. Take an example with a HAAUKINS event with 50 labs with 50 exercises running one container each, and all the users are scanning simultaneously. The number of ARP packets to be processed is calculated in Equation (2.1). Whether or not this problem also appears in the other drivers is unknown, but it will be tested as a part of reproducing the initial error in Section 2.3.

$$P = L(sn * c) = 50(255 * 50) = 637500 packets \tag{2.1}$$

P Number of packets in total
L Number of labs
sn Number of hosts in lab subnet
c Number of containers in lab

Looking at the macvlan drive, Docker recommends using the bridge networking instead unless necessary for the containerized application to have a direct connection to the physical network[3]. Combined with the use case of HAAUKINS, where many containers are opened and closed known to cause troubles due to the large number of unique MAC addresses as a new MAC is generated for each new container.

In order to be able to expose these MAC addresses to the underlying network, the generated MAC addresses are added to the kernel's ARP table. With the large amounts of containers used for running HAAUKINS events, the ARP table on the host will grow very large. Therefore Docker recommends using IPvlan instead of macvlan, as it puts less pressure on the ARP tables on the host and the switches the host is connected to, adheres to the one port one MAC security rule and prevents MAC exhaustion on the Docker host NIC.

With the problems found during the initial examination of the Docker networking, the next section will summarize all findings from the examination.

2.2.2 Subsidiary conclusion

This section will summarize the findings from Section 2.2. The sections started with examining the different ways, Docker can provide containers with networking. It was found that it is possible to use a custom networking driver such as OVS to provide networking to the containers, which will be considered when designing the system in Chapter 3.

Section 2.2 also showed that the IPvlan and macvlan drivers for networking are quite similar. However, as mentioned in Section 2.2.1, Docker recommends using either bridge networking instead of macvlan if possible, as using macvlan puts significant pressure on the host's ARP table, especially in large container networks, as each of the containers will be present in this table.

Section 2.2.1 also considered a problem with using the bridge networking as this would force the kernel to process broadcast messages, such as ARP traffic, many times. Therefore this might not be the best solution when handling many containers, which is often the case with HAAUKINS.

Section 2.3 will try to reproduce the error seen, monitoring the system for the problems identified in Section 2.2.1.

2.3 Reproducing the problem

This section will describe the methods and results from tests done to reproduce the packet drops seen when HAAUKINS is running with the higher demand described in Chapter 1, starting with establishing a set of hypotheses as to why the problem is occurring based on the findings in Section 2.2. Then a set of tests will be created in order to confirm or deny each of the created hypotheses.

Based on what has been seen in the initial investigations, where ARP traffic disappears when many labs are running, and the users are scanning simultaneously. The theory is that the large number of packets from the scan fills up the networking queue leading to packet drops as described in Section 2.2.1. Based on this, the first hypothesis can be established as:

"The problem with the kernel having to process the same packet multiple times, leading to packet drops, also applies to other networking methods."

The examination of the macvlan driver in Section 2.2 and Section 2.2.1 found that using macvlan for large container networks is not recommended. This is because it puts enormous pressure on the ARP table on the host, and the switches the host is connected to. In the HAAUKINS case, where the network is isolated on the host machine, only the host's ARP table is under pressure. Cases where the ARP table is full or outdated due to the quickly changing network, are known to cause unicast flooding leading to packets being dropped. From this, a second hypothesis can be created:

"The large number of containers using macvlan causes problems for the ARP table leading to packet drops."

A sub hypothesis is created based on observations during training events for the 2022 version of DDC. The observation is that the packet drops are happening more consistently when the servers are stopping and starting the events simultaneously with people scanning the network. Therefore the third hypothesis can be established as:

"Starting and stopping a large number of containers when scanning affects the result of the scans."

With three hypotheses as to why the problem is occurring established, the following sections will describe the methods to how these will be tested.

2.3.1 General methodology

This section will describe the methodology used when reproducing the error and testing the hypotheses. First, a set of settings used for all tests will be presented; these settings include the number of labs used in the test and the hosts used for generating the traffic, as additional settings are needed to simulate the scenario of the initial scanning.

To separate the tests properly, part of spinning up the test events is to ensure that all labs are started and ready, and that an event is fully closed before starting a new one. To test the higher demands effect on the packet drops, the tests will be run multiple times

with different amounts of labs. The amount of labs that each test will be run with is set to 25, 50, 75 and 100. It is chosen based on common event sizes and fits the number of active labs across long-running events.

These labs will contain 17 exercises, a DHCP container, a DNS container and the frontend VM, making the total hosts in the network 20 before adding the scanning container. The list of exercises used is shown in Table 2.1.

Names	Names
Alternate History Ciphers	Linux Walkthrough
The Flag Gallery	Heartbleed
Cookie Session	Exposed Logging Login Blogging
JWT	Cross-Site Scripting
Micro CMS	Cross-site Request Forgery
Writing on the Wall	It's Samba time!
SQL Change Credentials	Git logs
Network Scanning	FTP Server Login
Rocking SSH	

Table 2.1: Exercises used in the test labs

In order to simulate multiple users scanning simultaneously, a Docker container is created which will be added to each lab network. This container will identify the network it is placed in and run a default nmap scan. As it will take some time for all containers to be created and started, the container will scan the network continuously, to ensure that the scan is run in all networks simultaneously and gain multiple measurements in all networks. However, as ARP is usually cached, this will not generate the correct traffic, so the container's ARP tables are flushed between the scans. The Dockerfiles and shell script used for creating this container can be found in Appendix A.

Each time a scan completes, the final result is logged to a text file in the container in the format shown in Code-block 2.1. After running the test for 2.5 hours, the files will be extracted using the Python script found in Code-block A.5 in Appendix A.2. This script will create a CSV file, where each column is the results from one of the containers.

```
Nmap done: 256 IP addresses (21 hosts up) scanned in 362.43 seconds
Nmap done: 256 IP addresses (21 hosts up) scanned in 354.45 seconds
Nmap done: 256 IP addresses (14 hosts up) scanned in 337.44 seconds
Nmap done: 256 IP addresses (21 hosts up) scanned in 357.46 seconds
Nmap done: 256 IP addresses (21 hosts up) scanned in 351.39 seconds
```

Code-block 2.1: Output from scans

These columns are then combined into one long list of results as it is assumed that the distribution of the hosts found by the scans is identical across all scanning containers. This is confirmed by looking at the output from processing the 25 labs version of the test described in Section 2.3.2. This list of scan results is then counted in bins based on the number of hosts found. This output bin is then normalized, resulting in a PMF describing the probability of the scan missing the different amounts of hosts. The processing script can be found in Appendix A.2. The process is then repeated for each number of labs, and the resulting PMFs can be compared.

$$E[X] = \sum_i x_i \cdot p(x_i) \tag{2.2}$$

$E[X]$	Expected value of X
x_i	Possible value of X
$p(x_i)$	Probability of X taking value x_i

These PMFs are used to calculate the expectation of the distribution. Which is used as an extra parameter for comparing the results. The expectation is calculated using the formula shown in Equation (2.2). The calculated value will be the number of hosts expected to be missing when doing a scan under the conditions in the test. With the general methodology described, the next section will describe the testing of the first hypothesis.

2.3.2 Hypothesis 1

This section will present the method for testing the first hypothesis: *"The problem with the kernel having to process the same packet multiple times, leading to packet drops, also applies to other networking methods."*

To test this hypothesis two tests are conducted; first, a baseline is created by running the test described in Section 2.3.1. The steps for the first test can be written as follows:

- Start event with set amount of labs.
- Verify all labs is started.
- Add scanning Docker to the labs.
- Run the tests for 2.5 hours.
- Extract scan results from containers.
- Repeat for different amount of labs.

The second test for this hypothesis is conducted like the previous test. Other than this, a separate event is started with five labs running the FTP Server Login exercise. Another container is then created, which tries to brute-force the FTP server; the Dockerfile and script can be found in Appendix A. This is done to test if a higher load on the host CPU and kernel affects the outputs of scans. The steps can be written as follows:

- Start event with set amount of labs.
- Verify all labs is started.
- Start event with on going brute-force attack.
- Add scanning Docker to the labs.
- Run the tests for 2.5 hours.
- Extract scan results from containers.
- Repeat for different amount of labs.

With the two tests for testing hypothesis 1 described, the next section will describe the test for hypothesis 2.

2.3.3 Hypothesis 2

To test the second hypothesis *"The large number of containers using macvlan causes problems for the ARP table leading to packet drops."* This is tested by starting a separate event with a many labs with a large number of exercises in each. This is done to create an isolated test where the ARP table is already quite large and static. The steps can be written as follows:

- Start an event with a large amount of exercises and labs.
- Start event with set amount of labs.
- Verify all labs is started.
- Add scanning Docker to the labs.
- Run the tests for 2.5 hours.
- Extract scan results from containers.
- Repeat for different amount of labs.

2.3.4 Hypothesis 3

This section will describe the method for testing the third and last hypothesis. The hypothesis is that stopping and starting a large number of containers when people are doing their initial scan will affect the output. As starting and stopping containers will force the host's ARP table to update to accommodate the newly added containers, and remove the MACs from the closed containers. In these tests, three additional macvlan networks will be created, and 20 containers will alternately be added/removed from each network, to simulate labs starting and stopping. This is done using the script in Code-block A.7, which is found in Appendix A.2

The steps can be written as follows:

- Start event with set amount of labs.
- Verify all labs is started.
- Create extra networks.
- Start script for adding/removing containers.

- Add scanning Docker to the labs.
- Run the tests for 2.5 hours.
- Extract scan results from containers.
- Repeat for different amount of labs.

With four tests described for testing the three hypotheses defined in Section 2.3, Section 2.3.5 will describe the results from the testing process.

2.3.5 Results

This section will describe the results from the tests.

2.3.5.1 Hypothesis 1

This section will examine the results from the two tests created for hypothesis 1, namely the test where the network is just scanned and the test where the network is scanned and someone is trying to brute-force a service, see Section 2.3.2.

Figure 2.5 shows the output PMFs from the first test described. It is clearly seen that when moving from 25 to 50 labs the distribution changes, and more hosts are missing in the scans. When moving from 50 to 75 labs, the probability of missing hosts is slightly worse, but the difference is less significant than the jump from 25 to 50. The same goes for the jump between 75 and 100 labs; it is getting worse, and as seen on Figure 2.5 only very few of the scans contain all hosts. Moreover, when examining the output file containing the lines seen on Code-block 2.1, it is seen that most scans take toward 1000 seconds to complete, which is a lot higher than what is seen in the other tests, which indicates some congestion in the network.

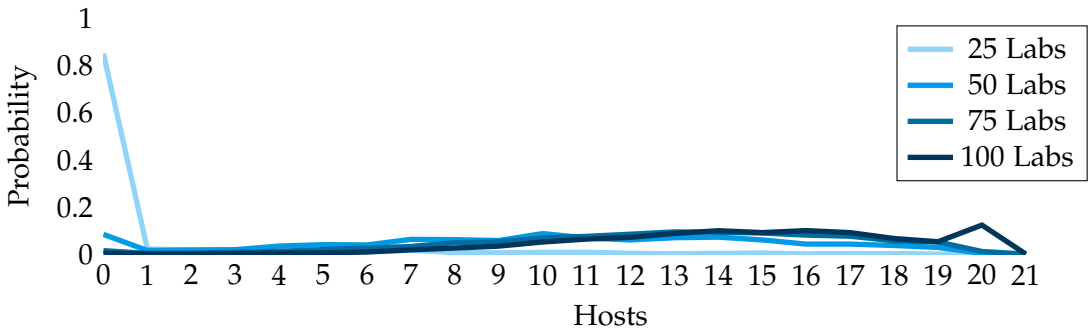


Figure 2.5: PMF of hosts dropped in the scans during test 1

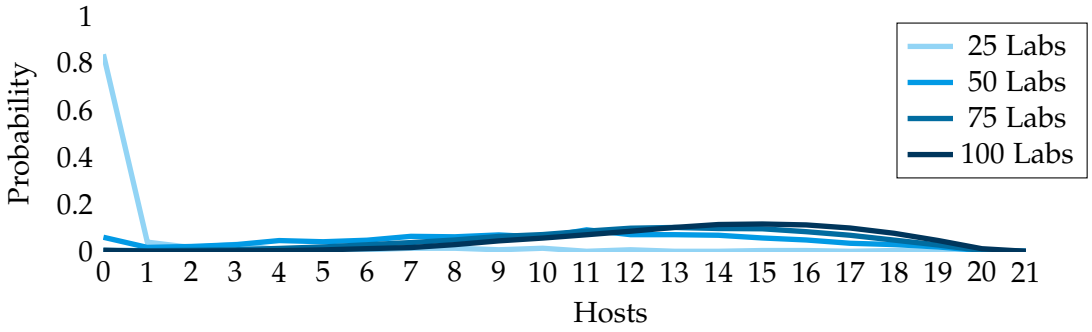


Figure 2.6: PMF of hosts dropped in the scans during test 2

Figure 2.6 shows the output PMFs from the second test described in Section 2.3.2. Comparing the PMFs for the different amounts of labs shows that the output is almost identical, meaning that the brute-force attack does not significantly impact the scanning output.

Table 2.2 shows the expectation for tests 1 and 2. These values indicate that the brute-force attack in test 2 does not affect the number of hosts missing and is actually expected to miss fewer hosts. This could indicate that the brute-force attack is not aggressive enough to affect the tests or that the system is already maxed out. However, the results from both tests show that when running more than 25 labs, the amount of missing host is rises quickly, and at 50 labs, almost 50% of the hosts is missing.

	25 Labs	**50 Labs**	**75 Labs**	**100 Labs**
Test 1	0.909586	9.942285	12.597068	14.499751
Test 2	0.870968	9.798601	12.384073	13.772186

Table 2.2: Expected missing hosts

When looking at the interface statics on the host by running the command `netstat -i`, the statics indicate that no packets were dropped during the test. This means that there could be a problem generating the traffic in the kernel or the kernel dropping the packet before it is counted. Looking at the system logs, it quickly becomes apparent that there is something wrong as the log is flooded with the statement seen in Code-block 2.2, which means that the ARP table on the host is too big[4]. These logs almost confirm hypothesis 2: *"The large number of containers using macvlan causes problems for the ARP table leading to packet drops."*

```
Apr 21 12:49:46 sec01 kernel: [531504.872464] neighbour: arp_cache: neighbor ↩
    table overflow!
```

Code-block 2.2: ARP overflow

Looking at the max size of the ARP table on the host, it is seen that it is set to 1024; this is found using the command `cat /proc/sys/net/ipv4/neigh/default/gc_thresh3`. The fact that the `gc_thresh3` value is set to 1024 means that as soon as the ARP table reaches 1024, it will clean records immediately, and the packet is dropped completely[4]. This could also explain the significant change in packet drops when moving from 25 to 50 labs, as the number of containers in the lab goes from 550 to 1050, plus some used for managing the platform, which is enough for overflowing the ARP table.

With the first two tests indicating that the issue is caused by the ARP table overflowing instead of the kernel being too slow to process the packets, the next section will examine this.

2.3.5.2 Hypothesis 2

Based on the findings in Section 2.3.5.1, it is found that it is not necessary to run the test described in Section 2.3.3, and the intended error is already provoked in the tests described in Section 2.3.2. Instead, the `gc_thresh3` threshold value is changed to four times the amount to allow the ARP table to accommodate 4096 containers or VMs. The `gc_thresh2` threshold value is set to half to minimize the soft clean up going on before reaching `gc_thresh3`. After setting this value to the higher value, the test described in Section 2.3.2 is repeated.

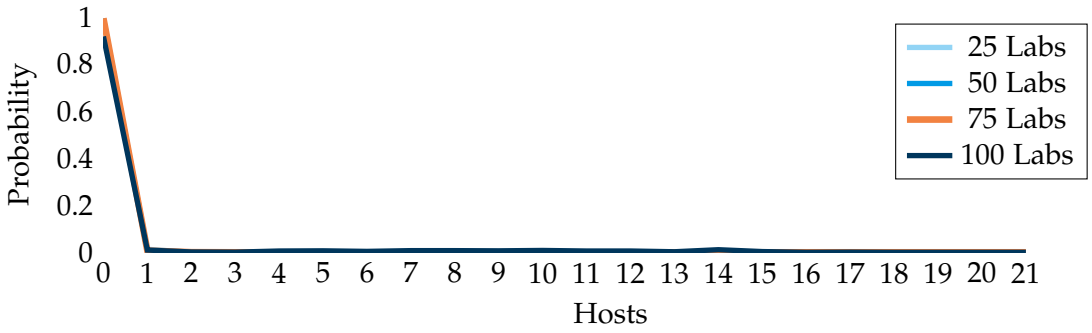


Figure 2.7: PMF of hosts dropped in the scans during test 3

Figure 2.7 shows the results from the test described above; it shows that after increasing the size of the ARP table from 1024 to 4096, the distribution is changed drastically. Here it is seen that for 25, 50 and 75 labs, the probability of missing zero hosts is almost one, and the probability of missing two or three hosts is 0.00695 and 0.00058. Calculating the expectation as done with the first two tests in Table 2.2, it is found that when running 75 labs with 20 containers, a VM and the scanning container described, the nmap scan is expected to be missing 0.008 hosts, as seen in Table 2.3.

	25 Labs	50 Labs	75 Labs	100 Labs
Expectation	0.010075	0.009855	0.008122	0.709652

Table 2.3: Expected missing hosts

However, when running 100 labs, the scans is start to miss some hosts during the scan, which is most likely because the amount of containers exceeds the `gc_thresh2` threshold, and the kernel starts to clean up the ARP table. In order to prevent this, one could set the `gc_thresh{1,2,3}` values even higher. This will be considered in Chapter 3 as when running labs with more exercises, the threshold will be reached faster.

2.3.5.3 Hypothesis 3

This section will describe the results from the last test described in Section 2.3.4, which tests if adding and removing containers from networks affect the scan results of other networks. The tests will be run with the new sizes for the ARP table described at the end of Section 2.3.5.1 while monitoring the system logs to see if the ARP table overflows.

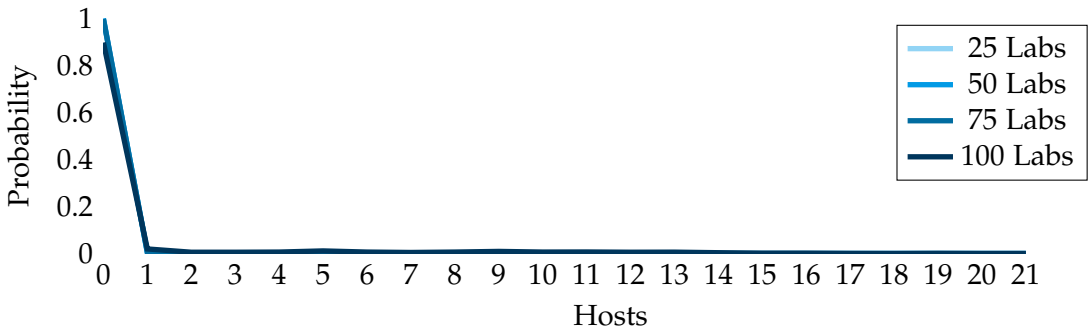


Figure 2.8: PMF of hosts dropped in the scans during test 4

When comparing Figure 2.7 with Figure 2.8, it is seen that they are close to identical, meaning that the system is not affected by the additional containers being stated and removed.

2.3.6 Subsidiary conclusion

Section 2.3 started by defining three hypotheses as to why the packets are being dropped when many people are scanning at the same time, which are stated as follows:

- The problem with the kernel having to process the same packet multiple times leading to packet drops also applies to other networking methods.
- The large amount of containers using macvlan causes problems for the ARP table leading to packet drops.

- Starting and stopping a large amount of containers when scanning affects the result of the scans.

Based on these, a set of tests were created in Section 2.3.2, Section 2.3.3 and Section 2.3.4. The two tests described in Section 2.3.2 are designed to both get a baseline of how the system responds to different amounts of labs and when running processor-heavy tasks. As described in Section 2.3.5.1, it was found that putting more pressure on the kernel did not affect the number of hosts that were missing in the scans.

However, further investigations showed that the ARP table started overflowing when running more than 25 labs, which means that the tests confirmed hypothesis 2. To further test this, a modified version of test 3 where described and the results were presented in Section 2.3.5.2.

During the fourth and last test, it was found that adding and stopping containers while running the scans did not have any impacts unless the additional containers made the total amount of containers exceed the ARP table size. Based on these findings, it is possible to accept hypothesis 2 and deny hypotheses 1 and 2.

2.4 Partial conclusion

This section will summarize the key findings from Chapter 2, which started with a short description of how HAAUKINS is used and its components in Section 2.1. The examination of the uses highlighted a set of requirements for the networking module in HAAUKINS, namely that there should be a complete separation between labs and the possibility of connecting through a VPN.

Section 2.2 then examined the current networking module using Docker macvlans for browser events and bridge for VPN events. Here it was found that macvlans might cause problems with the host's ARP table. It also found that using the bridge networking might cause problems with large amounts of containers as it causes the kernel to process broadcast packets multiple times based on the number of containers.

Based on the takeaways in Section 2.2, three hypotheses were made in Section 2.3, and a set of tests were designed to test the hypotheses. The testing confirmed that hypothesis 2 was true, and a possible solution was tested as well. Table 2.4 shows the expected amount of missing hosts in the scans calculated from the PMFs. It shows that the third test minimized the missing hosts significantly, and the solution tested will be examined further in Chapter 3.

	25 Labs	50 Labs	75 Labs	100 Labs
Test 1	0.909586	9.942285	12.597068	14.499751
Test 2	0.870968	9.798601	12.384073	13.772186
Test 3	0.012857	0.009855	0.008122	0.709652
Test 4	0.010076	0.003135	0.006094	0.716366

Table 2.4: Expected missing hosts

The results also showed that hypothesis 3 could be denied as there is no significant effect by spinning up and stopping containers as done in test 4. The next chapter will describe how the problem identified can be solved by either exchanging the networking module or by raising the `gc_thresh{1,2,3}` values.

3 Design

This chapter will present the design decisions made to improve the networking module of HAAUKINS. It will establish the requirements for a networking module used in HAAUKINS in Section 3.1. The requirements will be based on the description of HAAUKINS in both Chapter 1 and Chapter 2. These requirements are then used when considering the different options for optimizing the network performance in Section 3.2. Section 3.2 will also consider the work needed to implement the different networking drivers, as this is an essential factor to consider when working with an existing codebase. Based on the findings in Section 3.2, a final design is presented in Section 3.3.

3.1 Requirements

This section will highlight the requirements to the networking module to make sure that alternative networking modules support these and preserve the functionalities. The requirements will be established based on the description of HAAUKINS in the previous chapters and can be listed as follows:

- Fully automated.
  - Adding and removing Docker containers.
  - Adding and removing VMs
  - Creating and deleting networks.
- Complete isolation.
- Accessible.
  - Connection through VPN.
  - Connection through RDP.
- Scalable.
- Realistic.

One crucial factor when choosing the networking module is that it should be possible to fully automate the process of creating and deleting the networks, and the adding and removing hosts from the networks created. The network to which the host is added should be automatically configured when added, meaning that they should be assigned an IP address, subnet and gateway. The networks should also be completely isolated to prevent the users of HAAUKINS from attacking each other. The lab networks should

be separated from the host network to prevent people from using HAAUKINS to attack real targets.

It is also essential to maintain the accessibility aspect of the platform, which means that people should be able to access their lab by either RDP or VPN. The goal of the work done in this thesis is to improve the overall scalability of the platform, namely that it should be possible to run a large number of labs without users losing important information or deteriorating the usability, e.g. latency for the ones using RDP to connect.

With the requirements for the module established, the next section will examine the different possible solutions.

3.2 Choosing a networking module

This section will examine the different possible networking modules that could be used in HAAUKINS. A part of this examination will also be to consider the work needed to implement the new module. The possible networking modules will be the ones mentioned throughout Chapter 2. The options found in Chapter 2 were to stay with the current Docker macvlan networking and increase the size of the ARP table, switch to Docker IPvlan or move away from using Docker networking and use another software switch such as OVS. These options will be examined closer in the following sections.

3.2.1 Macvlan with larger ARP table

This section will describe the considerations regarding keeping the current macvlan networking. As described in Section 2.3.6, it is possible to keep the current Docker macvlan networking by increasing the size of the ARP table on the host. The ARP table on Linux hosts is controlled by three values `gc_thresh{1}`, `gc_thresh{2}` and `gc_thresh{3}` as briefly described in Section 2.3.5.2.

`gc_thresh{1}` is the minimum number of entries that are kept in the ARP table, and the table will not be cleaned unless the value is exceeded[15]. The default value is set to 128 and is not necessarily desirable to raise as this will mean that the kernel never will remove stale connections. A high `gc_thresh{1}` in a network where the ARP records change often, e.g. HAAUKINS where containers are often started and closed, might lead to lower IP reachability as the kernel will not update the table if the table is smaller than the value. With this in mind, the `gc_thresh{1}` is not a relevant parameter to change considering the problem to solve.

`gc_thresh{2}` is the soft maximum of entries in the ARP table, meaning that when this limit can only be exceeded for 5 seconds before the garbage collector starts emptying the table. By default, this value is set to 512 but is recommended to be set to the expected amount of entries plus a 20%-50% buffer[15]. In the HAAUKINS setup, where an event can easily have 60 participants and 45 exercises, this threshold should be raised to 4096, as calculated in Equation (3.1). However, with the high demand mentioned in Chapter 1,

it might be necessary to set this value even higher than this value. Increasing the value does not significantly impact one ARP entry is approximately 0.5KB, which means that 10000 superfluous entries will use 5MB of system memory.

$$T = L * c * b = 60 * 45 * 1.5 = 4050 \tag{3.1}$$

T Table size
L Number of labs
c Number of containers in lab
b Buffer factor

`gc_thresh{3}` is the hard maximum of entries in the table. If this is exceeded, the garbage collector will be running at all times. If the limit is reached, additional entries will be discarded, and the packet will be dropped by the kernel[4]. The default value is set to 1024 but should be raised to a value 20%-50% higher than the `gc_thresh{2}` threshold in case of a burst of additional hosts[15]. Alternatively, the ratio between the defaults could be kept, setting the `gc_thresh{3}` to double the amount of the `gc_thresh{2}` value.

These values can be set by either appending the lines seen in Code-block 3.1 in the `/etc/sysctl.conf` file and running the command `sysctl -p` to load the settings or by running the commands seen on Code-block 3.2. The first option makes the setting persistent through reboots.

```
net.ipv4.neigh.default.gc_thresh1=128
net.ipv4.neigh.default.gc_thresh2=4096
net.ipv4.neigh.default.gc_thresh3=8192
```

Code-block 3.1: Settings for ARP table

```
sysctl -w net.ipv4.neigh.default.gc_thresh2=4096
sysctl -w net.ipv4.neigh.default.gc_thresh3=8192
```

Code-block 3.2: Settings for ARP table

As seen on Code-block 3.1 and Code-block 3.2, minimal effort is required to implement this solution, and these values could be changed in a script for installing the HAAUKINS dependencies initially. With the changes necessary for keeping the current macvlan networking explained, the next section will consider exchanging macvlan for IPvlan networking.

3.2.2 IPvlan

This section will consider the option of using IPvlans as the networking provider. From the initial examination of the different Docker networking modules in Section 2.2, it

was found that IPvlan and macvlan are quite similar. The major difference between the two is that with IPvlan, all the containers in the same network will get the same MAC address but different IP addresses, and in macvlans all containers will get a unique MAC addresses.

The fact that all Docker hosts in the network will have the same MAC address will interfere with the requirement that the system should be realistic, as usually, each host has a unique MAC address. Initial testing also shows that when using IPvlan it will not be possible for the users to do network sniffing, which is most likely because of how Docker routes the traffic between the hosts in the network. These two factors mean that IPvlan is most likely not the solution to the problem, as the solution should preserve functionalities and comply with the existing requirements.

Because of the similarities between the current macvlan and IPvlan the work necessary to switch is very low and will only be to switch a couple of lines in the function creating the networks. With the option of switching to IPvlan examined, the next section will examine the possible switch to a non-Docker networking module.

3.2.3 Open vSwitch

This section will examine the possibility of exchanging the networking module with a non-Docker module. This is done by using the none as the the networking driver when starting the container and then adding the network to the container afterwards. Some of the tools that could be used for connecting the VMs and containers could be OVS, vmware NSX, vmware vSphere or VDE switch.

	Open-source	Golang SDK	Experience	Active development
OVS	Yes	Yes[18]	Yes	Yes
NSX	No	Yes[21]	No	Yes
vSphere	No	Yes[22]	No	Yes
VDE	Yes	No	No	No

Table 3.1: Comparison of different virtual switches

Table 3.1 compares the four packets mentioned. The first parameter on which they are compared is if the programs are open-source. This is considered to ensure that it is free to start an instance of HAAUKINS, which means that the vmware NSX and vSphere are not suitable solutions for HAAUKINS. The next parameter is if a Golang SDK exists for the solution here, it is found that the only one without is VDE switch. This is needed to automatically be able to create and remove networks from the HAAUKINS core component.

When looking at the development of the packages, it is seen that the VDE switch

is the only of the four without any active development, and the recent pushes made to their Github repository have only been fixes to their DevOps pipelines[20].

The last important fact considered is prior experience using the software. Here, the only one used before is OVS, as OVS has been used for creating the networking for the DefAtt platform, a highly automatized platform for red team and blue team games, particularly focusing on network-based attacks and their defences[9][10]. Therefore it is chosen to examine how OVS could be implemented to provide the networking to the VMs and containers in the HAAUKINS labs.

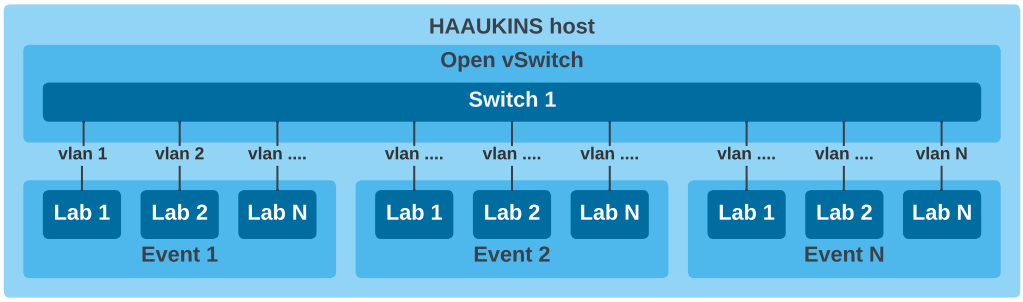


Figure 3.1: Network assignment using OVS

A setup where OVS is used to provide networking could be designed in multiple ways. The first way is illustrated in Figure 3.1, a single OVS switch is created, and all labs across events are connected to this. Each lab is then assigned a VLAN, meaning that it is possible to have up to 4094 different networks on the switch[6, p. 39], which is much higher than the number of labs that the server is capable of running. However, OVS also has a maximum size of its ARP table, which is set to 8192 for each unique switch created on the host[23].

This table limit of 8192 could cause problems in scenarios when large events in suspended while other large events are still running. Therefore the second way of implementing OVS, illustrated in Figure 3.2, is to create a unique OVS switch for each event and a VLAN for each lab.

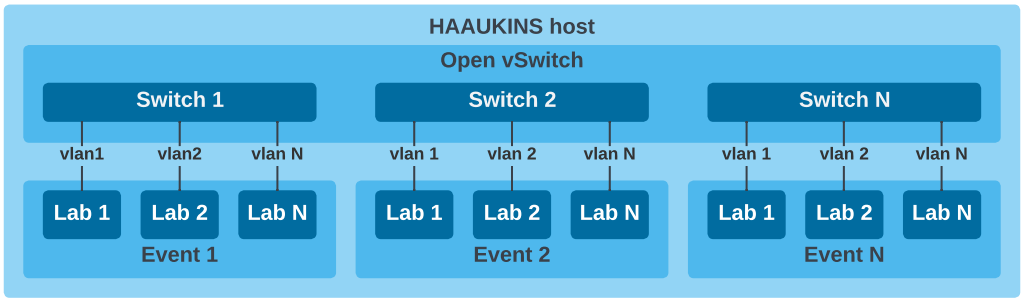


Figure 3.2: Network assignment using OVS

The third way OVS could implemented is to create a separate switch for each lab,

as illustrated in Figure 3.3. This setup will provide the largest amount of separation between the different lab networks and ensure that the ARP table in the switch will never overflow as with the setup described for the other setups.

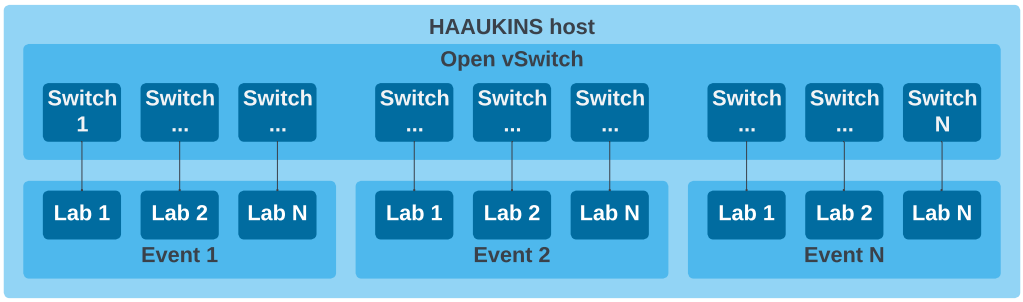


Figure 3.3: Network assignment using OVS

A quick testbed implementation where the switch and hosts are created and connected manually showed that all three solutions comply with requirements established in Section 3.1, and the functionalities are preserved. However, one downside of switching to OVS would be that it requires a major refactoring of the HAAUKINS codebase, both in creating and managing networks and managing exercises, as these are currently tied together. The task of separating these modules and implementing OVS as the networking module could improve the modularity and maintainability of the HAAUKINS platform.

3.2.4 Subsidiary conclusion

This section will present the findings from examining the different possible solutions to the problem. Section 3.2.1 examined the possibility of keeping the current macvlan networking by increasing the size of the ARP table on the host.

Here it was found that this could quickly be done by setting the `gc_thresh{2}` and `gc_thresh{3}` values. It was also found that setting the `gc_thresh{2}` value to 4096 should provide a sufficiently sized ARP table based on the calculation made in Equation (3.1). The calculation assumes that the server will be running 65 labs with 45 containers each across all events, but there are no downsides to setting the values higher. The `gc_thresh{2}` and `gc_thresh{3}` values only need to be set once when the server is initially set up for running HAAUKINS.

Section 3.2.2 examined the possibility of switching macvlan for IPvlan, and it was found that it could be done by changing a few lines within HAAUKINS due to the similarities. However, the testing showed that some of the core functionalities were not possible, like listening to the network traffic, and that the realistic requirement was compromised as all hosts would have the same MAC address.

Lastly, the option of switching Docker networking for another networking module

was explored, it was chosen to focus on OVS due to prior experience and it being the open-source alternative. Section 3.2.3 proposes three possible that provide the same separation between networks as the current macvlan, while also preserving the current functionalities. However, it was found that the work associated with the switch is much higher than implementing the solution proposed in Section 3.2.1. However, this higher workload could improve the maintainability of the platform by separating the components, making it easier for developers to fix bugs and add new features in the long term.

3.3 Final design

This section will present a final design, which is split into a short term and a long term solution. The two solutions are based on the considerations made throughout Section 3.2. The short term solution will be to implement the increased ARP table and will be described in Section 3.3.1. The long term solution will be to exchange the macvlan for OVS, which includes a refactor of the current networking implementation. This design will be described in Section 3.3.2.

3.3.1 Short term solution

This section will describe how the solution described in Section 3.2.1 can be implemented as a quick and possible short term solution. The solution presented is mainly changing system settings on the server running HAAUKINS. This solution will be designed as a script that prepares a server for running HAAUKINS.

The script will have a couple of elements, as illustrated in Figure 3.4, but assumes that some basic configurations have been made to the server, such as disabling logging in as root. The first step will be to add configurations described in Section 3.2.1, namely to set the `gc_thresh{2}` and `gc_thresh{3}` values to 4096 and 8192, respectively. After setting these system settings, the dependencies of HAAUKINS are installed hereunder Docker and vbox.

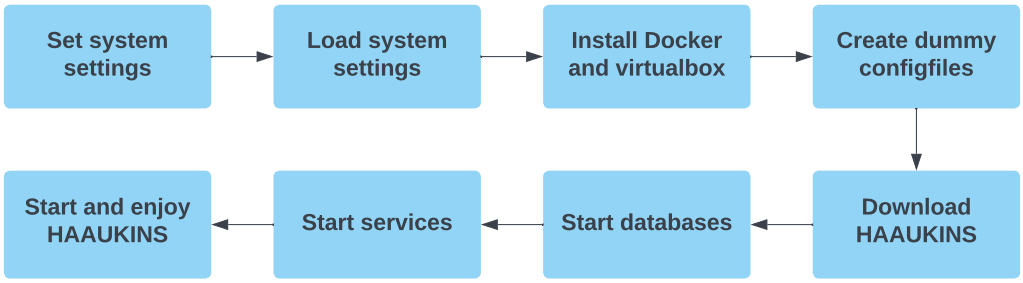


Figure 3.4: Flow of script implementing the short term solution

After installing the required dependencies, a directory with dummy configurations

is created, and the HAAUKINS modules are downloaded. After downloading the HAAUKINS modules, the databases can be started either as containers or running directly on the host. With the databases running, the exercise and store services can be started before finally starting HAAUKINS.

With a short term solution described, the next section will describe the design changes necessary for implementing a long term solution.

3.3.2 Long term solution

This section will describe the design of a networking module for HAAUKINS that will improve the networking performance long term. This long term improvement consists of two parts; the first part is to refactor the code handling the creation and managing og networks, and the second part is to use this improved networking module to implement OVS.

The long-term solution design will be using the third setup described in Section 3.2.3, where each lab has its own OVS bridge, see Figure 3.3. This is done by creating an interface describing the functions that a networking module should implement to be used as a networking provider for a lab. The interface design is inspired by the current implementation of the Docker networking, and a UML diagram of the interface is illustrated on Figure 3.5.

Figure 3.5 shows the structs containing an instance of this network interface, namely the environment and exercise structs, and the two structs implementing this interface, namely an OVS struct and the old Docker network struct.

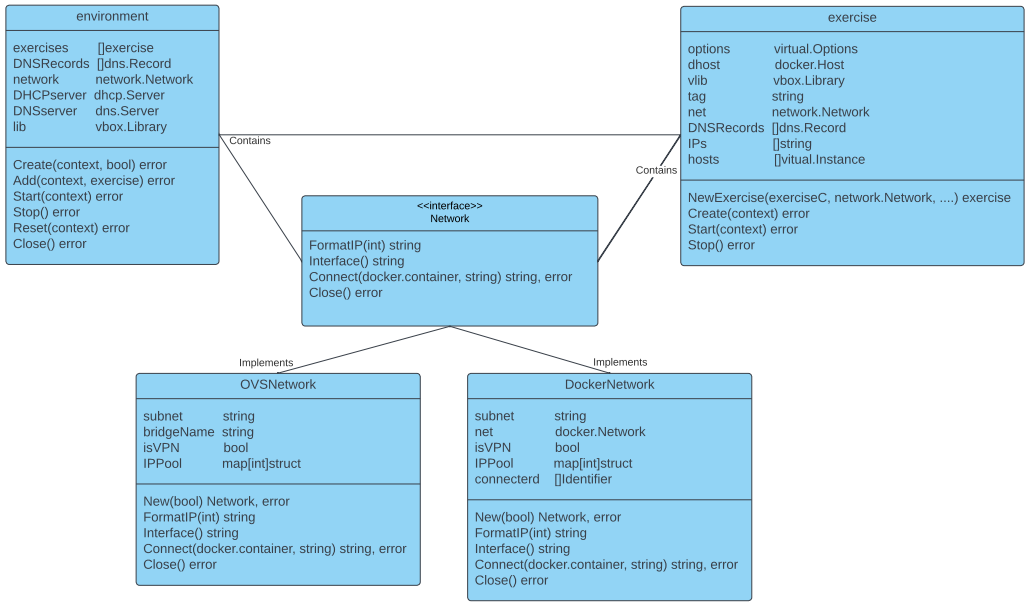


Figure 3.5: UML of network interface

This design will be reconsidered in Chapter 6, where further consideration of how HAAUKINS can be improved is presented. Hereunder how it could be possible to create a client-server architecture where a central server distributes the load across multiple servers, allowing HAAUKINS to be scaled horizontally as well as vertically. With a design of how the current networking could be changed to implement other networking modules other than Docker, the next section will summarize all the findings from this chapter.

3.4 Partial conclusion

This section will summarize the findings and design choices from Chapter 3. The design started with a presentation of the overall requirements for HAAUKINS and a networking module used in HAAUKINS. It was found that it is important that the network used within HAAUKINS should be as realistic as possible while still ensuring that the network is separated from the internet and other labs.

Section 3.2 then described three possible solutions while also considering the workload associated with implementing the solution. Here it was found that it is possible to keep the current macvlan by increasing the size of the ARP table, which is then used for creating a short term solution to the problem described in Section 3.3.1. It was also considered if IPvlan could have been used as it requires fewer recourses than the macvlan. However, initial testing showed that it interfered with the realistic requirement and some of the platform's functionalities.

Lastly, the possibility of exchanging macvlan with OVS was examined, and it was found that this could be a possible solution for a long term change to HAAUKINS. The examination of how OVS could be implemented in Section 3.2.3 presented three different solutions to how it could be used to provide networking. The last was used to create the long term solution described in Section 3.3.2.

With two designs, one for a short term solution and one for a long term solution presented, the next chapter will describe how these two are implemented.

4 Implementation

This chapter will present how the two solutions designed in Section 3.3 has been implemented. The structure will follow Section 3.3, starting with a description of how a short term solution has been implemented in Section 4.1, followed by the long term solution in Section 4.2.

4.1 Short term solution

This section will describe how the script for adding the short term fix has been implemented. The script itself will consist of multiple parts, as illustrated in Figure 3.4. The script will be created so that it can be used on a server where HAAUKINS is already installed or to completely prepare a newly installed server to run HAAUKINS with a set of default settings. To limit the scope of the script, it will be written to work on Ubuntu or Debian servers.

Code-block 4.1 shows the part of the script that will be run when HAAUKINS is installed. It starts by verifying that the user has administrator privileges (sudo) is it is required to install anything and set and load the settings for the system.

```
if [ $UID != 0 ]; then
    echo "ERROR: This scripts requires sudo"
    exit 1
fi

function set_system_settings {
    echo "Setting size of ARP table"
    echo "net.ipv4.neigh.default.gc_thresh1=128
net.ipv4.neigh.default.gc_thresh2=4096
net.ipv4.neigh.default.gc_thresh3=8192" >> /etc/sysctl.conf
    sysctl -p > /dev/null
    echo "Size of ARP table set and loaded"
}

while true; do
    read -p "Is HAAUKINS already installed? " yn
    case $yn in
        [Yy]* ) set_system_settings; exit 0;;
        [Nn]* ) fullInstall; exit 0;;
        * ) echo "Please answer yes or no.";;
    esac
done
```

Code-block 4.1: Script for installing HAAUKINS part 1

Then the function for setting the requirement is defined by writing the setting to the sysctl.conf file as described in Section 3.2.1. The last block in Code-block 4.1 will prompt the user if HAAUKINS has already been installed. If not, the function seen in Code-block 4.2 will be called. Otherwise it will run the `set_system_settings()` function.

```
function InstallDependencies {
    sudo apt update
    sudo apt upgrade -y
    sudo apt install apt-transport-https ca-certificates curl software-↩
        properties-common git virtualbox unzip -y

    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --↩
        dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
    echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings↩
        /docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu↩
         $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker↩
        .list > /dev/null

    sudo apt update
    sudo apt install docker-ce -y
}

function fullInstall {
    mkdir haaukins
    cd haaukins
    set_system_settings
    InstallDependencies
    InstallHAAUKINS
    createDefaultConfigs

    echo "Change the ower of the haaukins folder by running th command  -  ↩
        sudo chown -R <USER> haaukins - and add your user to the docker group↩
         sudo usermod -aG docker <user>"
    echo "Start databases by running docker-compose up -d in the configs foler↩
        "
    echo "Please update the files in /configs to run HAAUKINS over HTTPS and ↩
        pull images from regristry"
    echo "Then start haaukins by starting the binaries in the following order↩
         - store, exercise and then hknd - with the parameter --config=↩
        configs/<respectiveconfig>"
}
```

Code-block 4.2: Script for installing HAAUKINS part 2

The first block seen in Code-block 4.2 is the function for installing the dependencies of HAAUKINS, such as Docker and VBox. It starts by ensuring that the system is fully updated before installing the Docker dependencies and VBox, then Dockers signing key is added to the system and Docker is installed.

The second block runs all the helper functions created, this function will start with the settings for the system as seen in Code-block 4.1, and then the dependencies are installed. After installing all dependencies, the script will download the latest release of the HAAUKINS components from their Github repositories. Then default configuration files are added to the configs folder allowing the user to run HAAUKINS without

certificates.

With the script for installing HAAUKINS on a freshly installed Ubuntu or Debian server described, the next section will describe how OVS is implemented as the new networking module.

4.2 Long term solution

This section will describe how the design for the solution described in Section 3.3.2 has been implemented in HAAUKINS. The OVS integration will be done using a Golang wrapper for the OVS commands. The wrapper is maintained by the security group at AAU and was originally developed by Digital ocean but was forked and extended to support the needs of projects within the group[18].

The first step of implementing the solution is to create the general interface, which the networking providers should implement and exchange it in the places where the old network is used. Code-block 4.3 shows the environment struct with the old network being replaced with the new one.

```
type environment struct {
    exercises   []*exercise
    dnsrecords  []*DNSRecord
    network     network.Network
    dnsServer   *dns.Server
    dhcpServer  *dhcp.Server
    lib         vbox.Library
}
```

Code-block 4.3: Environment struct

```
type Network interface {
    FormatIP(num int) string
    Interface() string
    Connect(c docker.Container, ip ...int)↩
        (int, error)
    io.Closer
}
```

Code-block 4.4: Network interface

Then the OVS networking can be implemented. This is done by creating a new struct which implements the functions required by the new networking interface seen in Code-block 4.4. Other than the functions seen in Code-block 4.4, a `NewOVSNetwork()` function is created, returning the struct implementing the interface. The `NewOVSNetwork()` function starts by creating a unique string used at the networking name and the interface to which the frontend VM is bridged. Then the OVS switch is created and assigned a subnet. The `NewOVSNetwork()` function takes a boolean as input indicating if the network is for a VPN lab, which is currently unused. The implementation currently only considers the browser events due to the limited time frame.

```
func NewOVSNetwork(isVPN bool) (*OVSnetwork, error) {
    netName, err := nanoid.Generate(nanoid.DefaultAlphabet, 10)
    if err != nil {
        return nil, err
    }

    if err := ovsClient.VSwitch.AddBridge(netName); err != nil {
        log.Error().Err(err).Msg("failed to create a bridge for the new ↩
            network")
        return nil, err
    }

    if err := shellExec("sudo", "ifconfig", netName, "up"); err != nil {
        log.Error().Err(err).Msg("failed to bring interface for lab up")
        return nil, err
    }

    sub, err := Pool.Get()
    if err != nil {
        return nil, fmt.Errorf("ip pool get new network err %v", err)
    }

    subnet := fmt.Sprintf("%s.0/24", sub)
    ipPool := make(map[uint]struct{})
    for i := 30; i < 255; i++ {
        ipPool[uint(i)] = struct{}{}
    }

    return &OVSnetwork{bridgeName: netName, subnet: subnet, isVPN: isVPN, ↩
        ipPool: ipPool}, nil
}
```

Code-block 4.5: Function for creating new OVSnetwork

After the switch has been created, the hosts can be connected. VMs is connected by bridging them to the interface returned by calling the `Interface()` function and Docker containers by using the `Connect()` function. The `Connect()` function takes the container which should be connected and the final digit of the IP address it should be connected at if a specific address is needed for the exercise to work. A random IP is chosen from the pool if no address is given.

After finding the IP address to which the container should be connected, it is verified that the container is running, and if not, it is started. Then the containers are added using the Docker part of the OVS client.

```
func (n *OVSnetwork) Connect(c docker.Container, ip ...int) (int, error) {
    var lastDigit int

    if len(ip) > 0 {
        lastDigit = ip[0]
    } else {
        lastDigit = n.getRandomIP()
    }
    if c.Info().State != virtual.Running {
        if err := c.Start(context.TODO()); err != nil {
            return 0, err
        }
    }

    ipAddr := n.FormatIP(lastDigit)
    gwAddr := n.FormatIP(1)
    options := ovs.DockerOptions{IPAddress: ipAddr + "/24", Gateway: gwAddr}

    if err := ovsClient.Docker.AddPort(n.bridgeName, "eth0", c.Info().Id, ↩
        options); err != nil {
        if len(ip) == 0 {
            n.releaseIP(ipAddr)
        }
        return 0, err
    }

    return lastDigit, nil
}
```

Code-block 4.6: Function for connecting Docker to OVSnetwork

As the old Docker networking already implements the new interface with minor changes, namely that it implements the functions seen in Code-block 4.4, these will not be shown. The last function changed is the function for creating the networks for new instances of the environment seen in Code-block 4.3. As seen in Code-block 4.7, the function takes a context and a boolean indicating if it is a VPN lab. The context is not used but could be implemented to abandon the function if it takes too long or to contain metadata needed for the function. The isVPN boolean is used to choose if the network should be created as the old Docker network using Docker bridge networking or if it should be created as the new OVS implementation.

```
func (ee *environment) Create(ctx context.Context, isVPN bool) error {
    if isVPN {
        net, err := docker.NewNetwork(isVPN)
        if err != nil {
            return fmt.Errorf("docker new network err %v", err)
        }
        ee.network = net
    } else {
        net, err := network.NewOVSNetwork(isVPN)
        if err != nil {
            return fmt.Errorf(" new network err %v", err)
        }
        ee.network = net
    }

    ee.dnsAddr = ee.network.FormatIP(dns.PreferedIP)

    return nil
}
```

Code-block 4.7: Function for creating the network the environment

With the main changes made to implement a working version of OVS networking that can be used for testing the performance explained, the next section will summarize the conclusions of the two implementation sections.

4.3 Partial conclusion

This section will consider the result of the implementations of the solutions. Starting with the implementation of the short term solution presented in Section 4.1, here the script for either installing HAAUKINS completely or just setting the required system settings based on feedback from the user. The script follows the design other than the additional check that allows it to be used on servers with a running version of HAAUKINS.

After presenting the script implementing the short term solution, Section 4.2 presented the implementation of the long term solution, namely to exchange the current Docker macvlan for OVS. The design presented in Section 3.3.2 was successfully implemented by creating the new network interface, adapting the old Docker network to implement the new interface and developing the functionalities for OVS.

The implemented solution is initially tested by running HAAUKINS on the computer used to develop the solution by spinning up an event on this computer. Then one of the labs is then accessed, and the network is scanned to see if the networking works as intended, which it does. The scalability and performance of the implemented solution will be tested in the following chapter.

5 Testing

This chapter will describe how the implemented solutions will be tested, both in terms of performance and functionalities. It will start with a description of how the short term solution is tested, here the focus will be mainly on the functionalities. After testing the short term solution, the method for testing the long term solution will be examined in Section 5.2.

5.1 Short term solution

This section will describe how the script developed as the short term solution has been tested. The testing of the script focuses on the functionalities of the script as there are no requirements to how quickly this should run. The performance of the solution has already been tested during the initial testing in Section 2.3. To ensure that it is running 100% as intended, the full script, where the system settings are changed and HAAUKINS is installed, is run on a cleanly installed Ubuntu server.

Testing the functionality of adding the system setting to a system with HAAUKINS already installed will be done on the laptop used for developing the script. This is done by running the script and using the cat command described in Section 2.3.5.1 to verify that the setting has been applied. The result of the test can be seen in Code-block 5.1. Here the current setting is outputted, the script is executed, and the new settings are outputted, which show that the script works as intended.

```
lanestolen@ThinkPad-E580> cat /proc/sys/net/ipv4/neigh/default/gc_thresh*
128
512
1024

lanestolen@ThinkPad-E580> sudo ./prep-server.sh
Is HAAUKINS already installed? yes
Setting size of ARP table
Size of ARP table set and loaded

lanestolen@ThinkPad-E580> cat /proc/sys/net/ipv4/neigh/default/gc_thresh*
128
4096
8192
```

Code-block 5.1: Testing function for setting system settings

To test the script on a newly installed server the full script will be tested will be

an instance of the latest Ubuntu server, at the time of writing, Ubuntu server 22.04 LTS. To test the script multiple times quickly, the instance will be run in a VM using VBox. Running the script in the VM shows that the script successfully installs all the dependencies and that it is possible to start HAAUKINS after modifying the default configuration files created. With the script for installing HAAUKINS and setting the system settings for running HAAUKINS tested, the next section will describe how the long term solution is tested.

5.2 Long term solution

This section will describe how the testing of the newly implemented OVS module will be done to evaluate the performance compared to the current module. The module will be tested following the general method described in Section 2.3.1. The reason for doing this is to be able to compare the results obtained from this testing with the baseline gained from the first test described in Section 2.3.2.

The test will be run with the exact event sizes, namely 25, 50, 75 labs and the 17 exercises listed in Table 2.1. After the event is completely spun up and ready, a slightly modified version of the scanning container created during the initial testing is added to the network. The modification done to the container is that it will wait a couple of seconds after being started before it tries to get its IP address. This is done as the way Docker containers are added to OVS switches requires the containers to be running. Other than that, the only thing that should be modified is that the script created for automating the process of adding the scanning containers should also be adapted to add the containers to the OVS switches instead.

The modification needed for the scanning container can be done by adding a `sleep 30` to the script seen in Code-block A.1 after the shebang. This will make the script wait for 30 seconds before getting the IP address and starting scanning the network. Due to the limitations of the Python OVS SDK, it is not possible to adapt the current script to add the containers to an OVS switch. Therefore a Golang version of the script is created utilizing the extended functionalities of the OVS Golang wrapper maintained by AAU.

The Golang code used to add the scanning container to all networks can be seen in Code-block 5.2. First, the clients for interacting with Docker and OVS are created, then all the switches are listed, and the containers are added to these by looping through the bridges.

```
func main() {
    ovsClient = ovs.New(ovs.Sudo())

    cli, err := client.NewClientWithOpts(client.FromEnv)
    if err != nil {
        log.Fatal().Err(err).Msg("failed to connect to docker")
    }
    dockerCli = cli

    ss, err := ovsClient.VSwitch.ListBridges()
    if err != nil {
        log.Fatal().Err(err).Msg("failed to list bridges")
    }

    for _, sw := range ss {
        log.Debug().Str("net", sw).Msg("adding container to network")
        ctx := context.TODO()
        id, err := createContainer(ctx)
        if err != nil {
            log.Error().Err(err).Msg("failed to create container")
        }
        if err := dockerCli.ContainerStart(ctx, id, types.↩
            ContainerStartOptions{}); err != nil {
            log.Error().Err(err).Msg("failed to start container")
        }
        if err := ovsClient.Docker.AddPort(sw, "eth0", id, ovs.DockerOptions{↩
            DHCP: true}); err != nil {
            log.Error().Err(err).Msg("failed connect container to network")
        }
    }

}

func createContainer(ctx context.Context) (string, error) {
    conf := container.Config{
        Image:           "lanestolen/scan-cont",
        NetworkDisabled: true,
    }

    resp, err := dockerCli.ContainerCreate(ctx, &conf, nil, nil, nil, "")
    if err != nil {
        return "", err
    }
    return resp.ID, nil
}
```

Code-block 5.2: Function to add Docker containers to networks

Due to time limitations, it was not possible to run the tests before handing in this report as these tests should be coordinated with the server maintainers, who are busy writing their thesis and users of the HAAUKINS instance running on the testing server. However, the results from running the tests will be presented during the exam.

5.3 Partial conclusion

This chapter presented the method and results from testing the solutions implemented in Chapter 4. The first section presented how the short term solution was tested by running the developed script on a freshly installed version of Ubuntu server 22.04 LTS. The test showed that the script successfully prepares the server for running an instance of HAAUKINS. Section 5.1 also showed that it is also possible to use the script for setting the system settings on a server with HAAUKINS already installed.

Section 5.2 described how the testing of the long term solution should be done using the same method as used for the initial testing in Section 2.3. The only difference is that the script used for adding the scanning container is converted to Golang as the OVS SDK for Python lacks the needed functions. The actual testing of the implemented solution has been moved to after the handing in this report as the testing should be coordinated with the maintainers of the HAAUKINS running on the server where the rest of the test is conducted as this would require this to be offline during the testing period. The reason for testing everything on the same server is to keep as many parameters the same as possible.

With the method for testing presented and the result of the testing of the short term solution presented, the next section will describe the future work that could be done to improve HAAUKINS further.

6 Future work

This chapter will consider further work that could be done to allow the HAAUKINS platform to scale further and thereby support more users. For the purpose of this chapter, the perspective will be expanded to focus on the platform as a whole and not just the networking part of the system. Chapter 1 presented earlier efforts to scale HAAUKINS; this was done by scaling it vertically and horizontally. The vertical scaling was done by adding additional memory and storage to the servers currently used for running the platform. The horizontal scaling was done by separating the platform into multiple microservices as described in Section 2.1.2, allowing the platform to be run on multiple servers simultaneously[13].

There are limitations to how much the platform can be scaled vertically, both in terms of cost and hardware limitations. This chapter will focus on how the platform can be scaled further horizontally. This is done by considering how the attempted microservice architecture can be adopted further to allow the HAAUKINS platform to automatically split the workload of running large events across multiple servers. This could improve the user experience for the teachers by not having to manually split the students across multiple events on multiple servers during events larger than what one server can handle. But also for the students as they could get a lab assigned faster and the lab being more responsive as the server is under a lower load.

Multiple ways of splitting the workload were considered hereunder the possibility of creating a Kubernetes cluster consisting of all the serves placed at AAU for running the Docker containers used in the labs. However, this idea was quickly discarded as this will drastically increase the complexity of handling the separation between labs, but also that this could create a large amount of control traffic between the servers, putting further pressure on the already pressured networking on the servers.

Therefore the possibility of creating a setup where the labs are distributed across the servers is proposed. This is done by splitting the HAAUKINS core component described in Section 2.1.2 into a server and client service. The server service will then be running on a central server managing the HAAUKINS cluster created by multiple client services connecting to it. Based on the specs of the server running the server service, this could also, be running all the databases as well as an instance of the client service as illustrated in Figure 6.1.

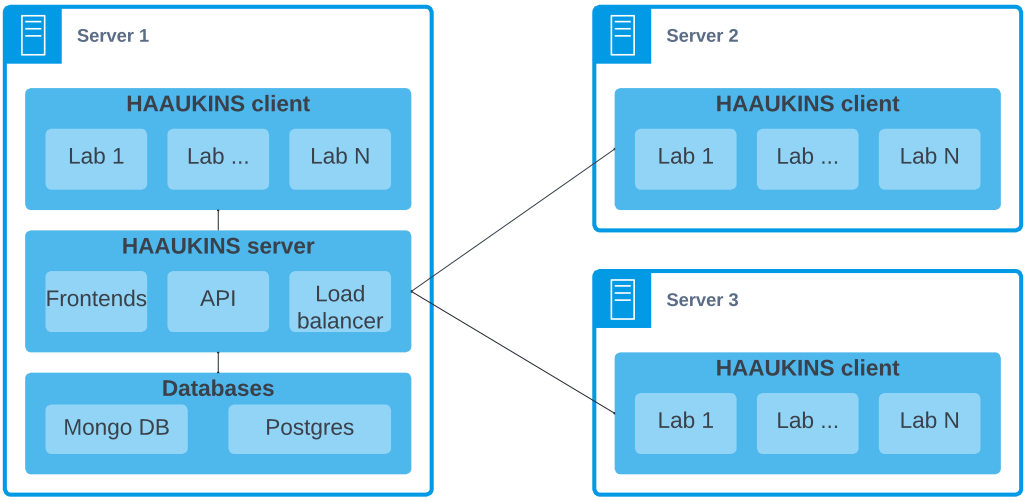
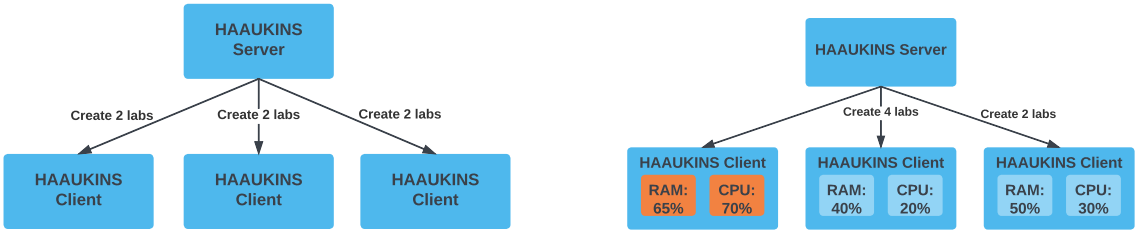


Figure 6.1: High level architecture of separated HAAUKINS

The server service will be presenting the frontends to the users and handle the distribution of labs across the clients connected to it. The connections are handled by an API exposed by the services used by both the webclient and client service. When users connect to their labs through the frontend, they are seamlessly redirected to the server running the lab.

The client services will then create labs based on configurations received from the server, including a list of exercises and lab type. This means that the client should integrate with the services used for enabling VPN and RDP access to the labs. But also Docker, VBox and potentially OVS in order to create the labs.



(a) Standard round robin

(b) Weighted round robin

Figure 6.2: Round robin load balancing

The load balancing between clients could be done using different algorithms. One option is to use round robin, which will divide the load equally between the clients, as illustrated in Figure 6.2a. However, as the servers that will participate in this HAAUKINS cluster will have different amounts of resources, a weighted round robin based on the specs the servers could provide better overall performance. This could be done statically by checking the system capabilities when the client initially registers with the

server service or dynamically by monitoring the system utilization of the servers and then assigning labs accordingly, as illustrated in Figure 6.2b.

The last type of load balancing that could be considered is to allocate the lab based on the latency between the user and the servers. This could improve labs' responsiveness, especially the browser events using RDP, as the user experience deteriorates quickly when the latency rises[12]. As all servers are placed in the same place, this is not relevant at the time of writing. However, if HAAUKINS gains international traction, deploying servers running the proposed client in datacenters in other places than on AAU could be relevant to lower the latency between the user and the server.

With considerations to how HAAUKINS can be modified to support even more users and improve accessibility further presented, the next chapter will summarize key findings from this project.

7 Conclusion

This thesis highlighted the increasing need for cybersecurity professionals and how the HAAUKINS platform was created to facilitate practical exercises during their education. However, as the platform's popularity has grown, it has been found that when running large events on the AAU servers, the students lose crucial information during their initial reconnaissance. Initial examinations found that the ARP traffic disappeared, but the cause was not found. This initial problem was then summarized in the problem statement:

"How can the networking of HAAUKINS be improved to prevent users from loosing crucial information from the system?"

Section 1.1 described the method used for approaching the problem, defining a set of information goals and how this information is obtained.

Chapter 2 continued with an examination of the different components and use cases of the HAAUKINS platform, trying to identify the possible causes of the problem. The examination of the components in Section 2.1.2 identified that it could be the Docker networking used that causes the problems. This led to an examination of the different networking modes that Docker provides in Section 2.2, which found that when Docker macvlan is used, each container has a unique MAC address which is added to the host's ARP table. Moreover, using Docker bridge networking could cause unicast flooding as broadcast packets are transmitted to all containers. Both of which could cause problems for the kernel processing the network traffic; therefore, the following set of hypotheses was created:

- The problem with the kernel having to process the same packet multiple times leading to packet drops also applies to other networking methods.
- The large amount of containers using macvlan causes problems for the ARP table leading to packet drops.
- Starting and stopping a large amount of containers when scanning affects the result of the scans.

These hypotheses were then tested in Section 2.3, and it was found that hypothesis 2 could be confirmed as the system logs showed that the host's ARP table overflowed already when running 50 labs with 17 exercises in each. This resulted in a slight modification of the third test, testing a possible solution for this overflow. This solution worked, and the probability of a scan missing any hosts decreased drastically. Testing

hypothesis 3 quickly denied the hypothesis but confirmed hypothesis 2 further as the system started dropping packets as soon as the ARP limit was reached. The results from the initial testing show the importance of considering the system settings when running large virtualized networks on a single server.

Based on the findings, two possible solutions were presented in Chapter 3, both of which optimize the networking performance of the labs The first solution was to implement a script that could set the system settings on a system with HAAUKINS already installed or set the settings, create the necessary configuration files and continue to install HAAUKINS and its dependencies. The second solution was to completely exchange the current Docker networking and switch to a provider such as OVS. Both solutions were successfully implemented and are described in Chapter 4.

As described in Chapter 5, the results from implementing the first solution, namely to increase the ARP table size, were already shown in the initial testing of the hypotheses, showing that it work as intended, increasing the performance of the network by minimizing the hosts missing in the scans. Chapter 5 also described that the long term solution has yet to be tested, as the testing was delayed due to coordination with the other maintainers of the AAU servers. However, the results will be presented as a part of the project presentation.

Lastly, Chapter 6 presented considerations how HAAUKINS can be further developed to support more users, and improve the user experience of both the teachers running the events and the users. Chapter 6 presented new client-server architecture where one central server will handle the web interface and redirection of the users to clients running the labs. Here it was also considered how this could be load balanced between multiple servers.

Acronyms

AAU	Aalborg University Denmark	2, 3, 32, 37, 40, 42–44
API	Application Programming Interface	41
ARP	Address Resolution Protocol	3, 9–12, 14, 16–19, 22, 23, 25–27, 29, 43, 44
CSV	comma-separated values	12
DDC	De Danske Cybermesterskaber	3, 11
FTP	File Transfer Protocol	13
gRPC	Google Remote Procedure Call	7
HtB	Hack the Box	2
OVS	Open vSwitch	8, 10, 22, 24–29, 32–35, 37, 39, 41, 44
PMF	Probability Mass Function	4, 13, 15, 16, 19
RDP	Remote Desktop Protocol	6, 7, 21, 22, 41, 42
SDK	Software Development Toolkit	24, 37, 39
THM	Try Hack Me	2
UML	Unified Modeling Language	28
VBox	VirtualBox	7, 31, 37, 41
VLAN	Virtual LAN	8, 25
VM	Virtual Machine	3, 6, 7, 12, 17, 21, 24, 25, 32, 33, 37
VPN	Virtual Private Network	3, 6–8, 19, 21, 22, 32, 34, 41

Bibliography

[1] Zainab Alkhalil et al. "Phishing Attacks: A Recent Comprehensive Study and a New Anatomy". In: *Frontiers in Computer Science* 3 (2021). ISSN: 2624-9898. DOI: 10.3389/fcomp.2021.563060. URL: https://www.frontiersin.org/article/10.3389/fcomp.2021.563060.

[2] Tom Burt. *Microsoft report shows increasing sophistication of cyber threats*. 2020. URL: https://blogs.microsoft.com/on-the-issues/2020/09/29/microsoft-digital-defense-report-cyber-threats/.

[3] Docker community. *Networking overview*. 2021. URL: https://docs.docker.com/network/.

[4] Kubernetes community. *arp_cache: neighbor table overflow! #4533*. 2018. URL: https://github.com/kubernetes/kops/issues/4533.

[5] Jason A Donenfeld. "Wireguard: next generation kernel network tunnel." In: *NDSS*. 2017, pp. 1–12.

[6] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)* (2014), pp. 1–1832. DOI: 10.1109/IEEESTD.2014.6991462.

[7] (ISC)². *A Resilient Cybersecurity Profession Charts the Path Forward*. 2021. URL: https://www.isc2.org//-/media/ISC2/Research/2021/ISC2-Cybersecurity-Workforce-Study-2021.ashx.

[8] Hangbin Liu. *Introduction to Linux interfaces for virtual networking*. 2018. URL: https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking.

[9] Rasmi-Vlad Mahmoud, Robert Nielsen, and Ahmet Turkmen. *DefAtt- Virtual Cyber Labs for Attack and Defence*. URL: https://github.com/aau-network-security/defatt.

[10] Rasmi-Vlad Mahmoud et al. "DefAtt - Architecture of Virtual Cyber Labs for Research and Education". In: *2021 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. 2021, pp. 1–7. DOI: 10.1109/CyberSA52016.2021.9478236.

[11] Sudakshina Mandal, Danish Ali Khan, and Sarika Jain. "Cloud-Based Zero Trust Access Control Policy: An Approach to Support Work-From-Home Driven by COVID-19 Pandemic". In: *New Generation Computing* 39.3 (2021), pp. 599–622. ISSN: 1882-7055. DOI: https://doi.org/10.1007/s00354-021-00130-6.

[12] Craig Medland. *Troubleshooting User Experience over RDP*. 2022. URL: https://bit.ly/3sZhjFI.

[13] Gian Marco Mennecozzi et al. "Bridging the Gap: Adapting a Security Education Platform to a New Audience". In: *2021 IEEE Global Engineering Education Conference (EDUCON)*. 2021, pp. 153–159. DOI: 10.1109/EDUCON46332.2021.9453985.

[14] Cedric Nabe. *Impact of COVID-19 on Cybersecurity*. 2021. URL: https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html.

[15] Xav Paice. *default gc_thresh settings for Linux are too small*. 2018. URL: https://bugs.launchpad.net/charm-nova-compute/+bug/1780348.

[16] Thomas Kobber Panum et al. "Haaukins: A Highly Accessible and Automated Virtualization Platform for Security Education". eng. In: *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*. Vol. 2161-377X. IEEE, 2019, pp. 236–238. ISBN: 9781728134857.

[17] Jens Myrup Pedersen. *DANMARKS FØRSTE UDDANNELSE I CYBERSIKKERHED*. 2020. URL: https://www.nyheder.aau.dk/2020/nyhed/danmarks-foerste-uddannelse-i-cybersikkerhed.cid447828.

[18] AAU network security and Digital Ocean community. *go-openvswitch*. 2021. URL: https://github.com/aau-network-security/openvswitch.

[19] Seetharami Seelam. *Scaling the Docker Bridge Network*. 2015. URL: https://www.ibm.com/support/pages/scaling-docker-bridge-network.

[20] virtualsquare. *VDEv2: Virtual Distributed Ethernet*. 2022. URL: https://github.com/virtualsquare/vde-2.

[21] vmware and community. *go-vmware-nsxt*. 2022. URL: https://github.com/vmware/go-vmware-nsxt.

[22] vmware and community. *govmomi*. 2022. URL: https://github.com/vmware/govmomi.

[23] Open vSwitch. *ovs-vswitchd - Open vSwitch daemon*. 2022. URL: http://manpages.ubuntu.com/manpages/impish/man8/ovs-vswitchd.8.html.

Appendices

A Reproducing the error

A.1 Dockerfiles and bash scripts for creating the containers

```
#! /bin/bash

# Find and generate IP in CIDR
IP=$(hostname -I)
ScanString="$(echo -e "${IP}/24" | tr -d '[:space:]')"

while true
do
# Scan the network and output hosts found
    nmap ${ScanString} | grep "Nmap done:" >> out.txt
# Flush ARP table
    sudo ip neigh flush all dev eth0
    sleep 10
done
```

Code-block A.1: Scanning script

```
FROM lanestolen/base-image
LABEL maintainer="Lanestolen"

COPY bash.sh .
RUN chmod +x bash.sh

RUN set -xe \
    && apk add --no-cache --purge -uU nmap sudo\
    && rm -rf /var/cache/apk/* /tmp/*

CMD [ "./bash.sh" ]
```

Code-block A.2: Scanning Dockerfile

```
#! /bin/bash

IP=$(ip -f inet addr show eth0 | grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'↩
    |sort -u)
DNS="$(echo -e "${IP}.3" | tr -d '[:space:]')"

TARGET=$(dig ftplogin.com @${DNS} +short)

while true
do
    hydra -l admin -P rockyou.txt ftp://${TARGET}
    sleep 10
done
```

Code-block A.3: Bruteforcing script

```
FROM ubuntu
LABEL maintainer="Lanestolen"

RUN apt update && apt upgrade -y \
    && apt install hydra dnsutils iproute2 -y

COPY bash.sh rockyou.txt ./
RUN chmod +x bash.sh

CMD [ "./bash.sh" ]
```

Code-block A.4: Bruteforcing Dockerfile

A.2 Python scripts for automating the tests

```
import docker
import re
import pandas as pd

client = docker.from_env()
reString = "\((\d) host"

data = []
headers =[]
conts = client.containers.list(filters={"ancestor":"lanestolen/scan-cont"})

for cont in conts:
    out = cont.exec_run("cat out.txt")
    hosts = re.findall(reString,out.output.decode("utf-8") )
    data.append(list(map(int, hosts)))
    headers.append(cont.short_id)

dataframe = pd.DataFrame(data)
dataframe.T.to_csv("test.csv",index=False,header=headers)
```

Code-block A.5: Script for collecting scan results

```
import pandas as pd
import numpy as np

df = pd.read_csv("50.csv")
arrays = df.T.values
data = []
normalized = []
expectation=0

for a in arrays:
    a = a[~pd.isnull(a)].astype(int)
    data.extend(a)

count = np.bincount(data)
for i in count:
    normalized.append(i/len(data))

out = np.flip(normalized)

dataframe = pd.DataFrame(out)
dataframe.to_csv("50out.csv",index=False, header=["pmf"])

for id, p in enumerate(out):
    expectation+= p*id

print(expectation)
```

Code-block A.6: Script proccessing output

```
import docker
import threading
import nanoid
import logzero
from logzero import logger
import time

FORMAT = '%(color)s[%(levelname)1.1s %(asctime)s]%(end_color)s %(message)s'
logzero.json()
formatter= logzero.LogFormatter(fmt=FORMAT)
logzero.formatter(formatter=formatter)

client = docker.from_env()
NumberNetworks = 3
nets =[]

i=0
while i < NumberNetworks:
    i+=1
    netname = nanoid.generate()
    net = client.networks.create(netname,driver="macvlan")
    nets.append(net.name)


def addContainers(net):
    logger.info("Adding containers in net: %s", net)
    c = 0
    while c < 20:
        c+=1
        client.containers.run("lanestolen/base-image",detach=True,network=net)
    logger.info("Added containers in net: %s", net)

def removeContainers(name):
    logger.info("Removing containers in net: %s", net.name)
    network = client.networks.get(name)
    for cont in network.containers:
        logger.debug("Killing container %s",cont.id)
        cont.remove(force=True)
    logger.info("Removed containers in net: %s", net.name)

def cycleContainers(net):
    logger.info("Starting thread")
    while True:
        addContainers(net)
        time.sleep(120)
        removeContainers(net)
        time.sleep(10)

for n in nets:
    thread = threading.Thread(target=cycleContainers,args=(n,))
    thread.start()
    time.sleep(20)
```

Code-block A.7: Script starting and stopping containers