Converting Hardware to a Container Solution and its Security Implication

Gustav Strömberg
bbh13gst@student.lu.se

Axis Communications

Supervisor: Christian Gehrmann (LTH), Martin Bäckström (Axis), Martin Ljunggren (Axis)

Examiner: Thomas Johansson

September 10, 2021

© 2021
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Hardware today can be inaccessible to users due to cost or the customer's desire for flexibility. By using virtualization one can reduce customer costs while increasing flexibility. To do this, companies might need to redesign or migrate their hardware to suit a virtualized environment. However, migration from custom to virtual hardware introduces security risks. This thesis, therefore, explores the possibility to transform a hardware solution into a container solution while retaining sufficient security.

The execution was divided into two steps, to gain knowledge on how one can protect the container and implementing the container. Two tools were considered to increase security: SCONE and Lic-Sec. The former one utilizes Intel SGX on the container to mitigate attacks from the host machine, while the latter is a tool that generates a profile for AppArmor that can shield it from other containers. The container was developed with Podman as its container engine since it enforces user namespace and allows the container to use systemd which was a requirement for the container to function.

The development of the container was a success, however, due to the structure of the container, neither tool could be used to enhance its security. Nevertheless, the thesis shows that systems can run on a container, although modifications to the hardware running the container or other tools are needed to obtain sufficient security for public use. Future research is needed to deduce if it is possible to replace a single container with a cluster which could increase security.

Acknowledgements

I extend my gratitude to Christian Gerhmann for all the help and guidance he has given me. This thesis would not be possible without him.

I would also like to thank Martin Ljunggren and Martin Bäckström, along with Stefan Andersson, at Axis for their input and for helping me debug and understand the system controller unit.

Thanks to Hui Zhu for modifying Lic-Sec which enabled it to be used with Podman, it would not be possible without her.

I would like to thank Robert Schambach and Christof Fetzer at SCONE for providing me with early access to their tool, and their willingness to discuss their product with me.

Finally, I am grateful to my fiancée Fredrika for all her support.

Popular Science Summary

Today, millions of people use virtualization without realizing it. It is often used to host websites or used in infrastructure cloud solutions. Imagine it like a balloon blown up halfway; when there are more users, i.e. more air, then the service requires a larger portion of virtualization, i.e. the balloon expands, and vice versa.

All virtualization uses physical hardware, however, it could be beneficial to virtually represent the hardware. This could reduce customer costs for a particular device, decrease the production cycle, and reduce the climate footprint. So is it possible to convert hardware into a virtualized entity and can it be done securely? This was researched at Axis Body Worn Solution (which develops body cameras for lawn enforcement) with the goal to convert their *system control unit* (SCU), a hardware that manages a web page and body worn cameras, into a container for public use.

A container is an OS virtualization and is an isolated process on a computer. It runs with no knowledge of the outside. Think of it like an apartment complex where a container is one apartment. They can be furnished independently of each other and the owners do not know how the other apartments look like, however, the complex owner, i.e. the computer, can peek in all apartments.

The SCU could be containerized successfully, however, it suffered in terms of security. The lax security enables anyone to manipulate the recordings from the camera on the SCU container. This can damage the container's legitimacy which furthermore could result in the footage being discarded as evidence in court.

Two security enhancements were used with the aim to provide additional security for the container, SCONE, and Lic-Sec. SCONE isolates programs and encrypt desired files. This would help to protect the SCU from the host computer, i.e. the user. Lic-Sec generates a profile that defines mandatory access control. It can prevent other containers from attacking it by tightening the rules on who can access what inside the container.

The security enhancements could not be applied to the container. This had to do with the nature of the container, in the case of SCONE, as well as technical design choices.

Table of Contents

1 Introduction **1**
1.1 Axis Body Worn Solutions . . . 1
1.2 Virtualization . . . 2
1.3 Project Goals . . . 3
1.4 Thesis Scope . . . 4
1.5 Problem Description . . . 4
1.6 Methodology . . . 5
1.7 Outline . . . 5

2 Technology Background **7**
2.1 Containerization . . . 7
2.2 Docker . . . 8
2.3 Podman . . . 9
2.4 Namespaces . . . 11
2.5 Cgroups . . . 12
2.6 Mandatory Access Control . . . 13
2.7 SELinux . . . 14
2.8 AppArmor . . . 15
2.9 vTPM . . . 16
2.10 Intel SGX . . . 18
2.11 Virtual Machine . . . 20

3 Containerizing the Hardware **25**
3.1 Overview . . . 25
3.2 Running SCU Programs . . . 27
3.3 Server Setup . . . 28
3.4 Finalizing the Image . . . 29

4 Increasing Container Security **31**
4.1 Sconify the Native Image . . . 31
4.2 Generating an AppArmor Profile With Lic-Sec . . . 33

5 Security Analysis **35**
5.1 General Security Risks . . . 35

5.2 Vulnerability Scan . . . 35
5.3 Container Enumeration . . . 36
5.4 With Security Enhancement . . . 37

6 Benchmarks **39**

7 Discussion **41**
7.1 The SCU Container . . . 41
7.2 SCU as a Virtual Machine . . . 42
7.3 Conclusion . . . 42
7.4 Future Work . . . 43

References **45**

List of Figures

1.1 Body Worn Solution setup 1
1.2 BWS setup with a container 2
1.3 VM and container architecture 3

2.1 Container image layer structure 8
2.2 Docker storage types 9
2.3 Architecture for Docker and Podman 10
2.4 Namespaces affect on container escape 12
2.5 cgroup hierarchy illustration 13
2.6 Core decision making SELinux architecture 14
2.7 The basic concept of MLS illustrated with clearance level 15
2.8 Proposed vTPM implementation architecture 17
2.9 SCONE architecture 19
2.10 Sconification process (file protection) 20
2.11 Two types of hypervisor technologies 21

3.1 Container content overview 25
3.2 Container architectural overview 26

4.1 SCU cluster design 33
4.2 Intermediate SCU cluster design 33

List of Tables

2.1 Types of namespaces 11
2.2 Number of VM vulnerabilities per product 23

3.1 Summery of container adaptations 27

5.1 Native image vulnerability scan 36

6.1 Footage upload time 39
6.2 The duration of adding and removing a camera to the system 40

Chapter 1
Introduction

Axis was founded in 1984 in Lund where they first developed printer servers that enabled printing from multiple computers to a single printer. About 10 years later they released the world's first network camera, *AXIS Neteye 200*. Although the image at the time was not suited for surveillance due to only having one frame every 17 seconds [1], it showed the aspiration and a glimpse of the future. Today, Axis is world-leading in camera surveillance, with network cameras, access control, and radars, among other products. In the summer of 2020, Axis *Body Worn Solution* (BWS) was released, and this thesis was done in cooperation with the BWS development team.

1.1 Axis Body Worn Solutions

Axis Body Worn Solution is a system aimed towards law enforcement and private security. Its purpose is to aid in evidence gathering, to deter bad behavior, and to serve as a tool in teaching how to respond in certain situations. It fulfills its purpose by recording video and audio from the perspective of the wearer and, in the case of Axis' BWS, can be triggered by touch, on fall, or when a firearm is unholstered.

Body worn solutions consist of three devices: the body worn camera (BWC), a docking station in which the camera is placed, and a system control unit (SCU). The component and the setup is shown in figure 1.1. When docked, the camera becomes locked and begins to upload its video and audio files, together with other associated data, to the SCU. The SCU then directs the footage to an evidence management system (EMS) of the user's choice. The SCU is compatible with multiple third-party evidence management systems such as Milestone[1] and Genetec Clearance[2], but also Axis' own EMS.

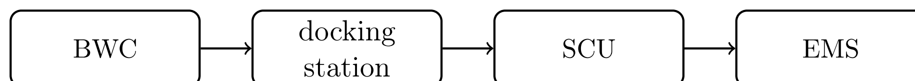


Figure 1.1: Body worn solution system setup.

[1] https://www.milestonesys.com/
[2] https://www.genetec.com/solutions/all-products/clearance

The cost of BWS may be an obstacle for a potential customer that requires one or two cameras; the reason for this is due to the cost of the SCU. Therefore, Axis wants to investigate the usage of a container-based solution to replace the SCU hardware. The goal is for the container to be deployed locally to increase flexibility and reduce customer costs. The container will have the functionality of the SCU and therefore needs to uphold appropriate security measures to ensure that the recordings are handled with a sufficient level of security. This in turn ensures that the recordings are legitimate, otherwise it might not be eligible as evidence. An overview of this system setup can be seen in figure 1.2.

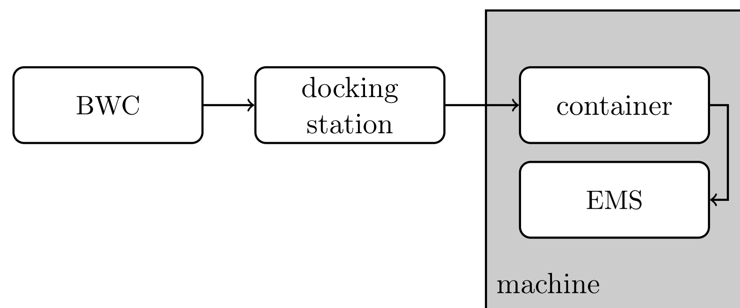


Figure 1.2: System setup with the use of a container running on the same machine as the EMS.

1.2 Virtualization

Virtualization is the act of creating virtual representations of hardware and software. Before the 1970s, virtualization was used in the form of a virtual machine (VM), as hardware was expensive. This allowed for multiple users to develop and test programs without the risk of crashing the computer and destroy other people's work [2].

Today, virtual machines allow users to emulate different operating systems (OS) and environments on a computer rather than having multiple computers for the same purpose. It is widely used in cloud computing as infrastructure as a service, platform as a service, and software as a service. These services allow the companies using them to easily expand with rising demand, or reduce with decreasing demand. This enables companies to be more cost-efficient as they pay for what they need at the moment, contrary to investing in servers to meet the highest demands.

Containers, on the other hand, are isolated processes on a host OS, in other words, it does not have an OS as a VM does. The difference in their architecture can be view in figure 1.3. Virtual machines use a hypervisor (virtual machine monitor such as VirtualBox) which the guest OS is placed on along with its dedicated resources, whereas the container is placed on top of the host OS and the container runtime is a software that is responsible for running the container, such as Docker, Podman, CRI-O, etc.

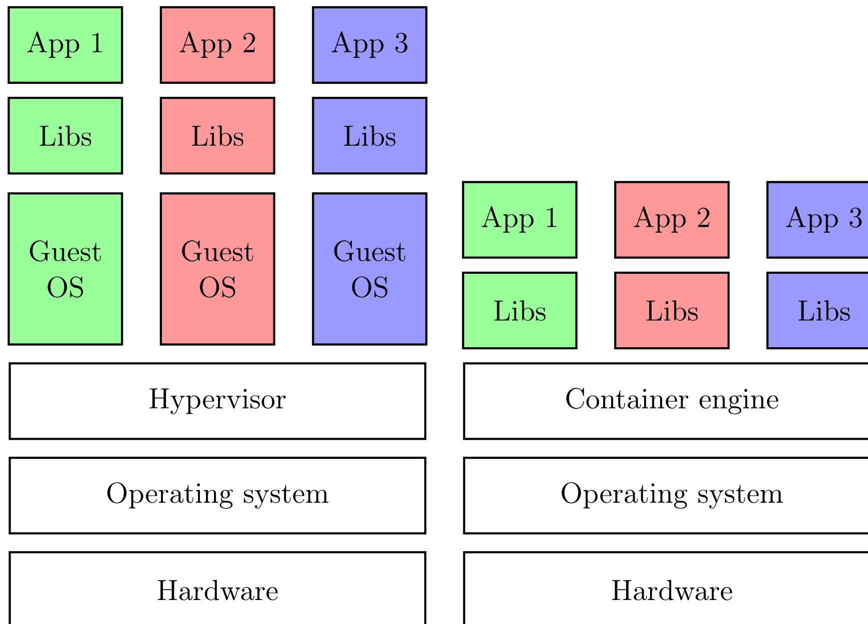


Figure 1.3: The architectural structure of the two types of virtualization. Left side is the virtual machine, right side the container.

1.3 Project Goals

The main goal is to replace the SCU hardware with a container. The container should contain the same functionality at an acceptable security level. To obtain these goals, a container architecture will be developed, followed by a security evaluation, implementation, and proof-of-concept. Alongside the implementation, a container solution will be compared, theoretically, with a VM solution to analyze the security and performance implications. Below is a summary of the thesis goals.

- Develop a functional container
- Perform a proof of concept on the implemented container solution
- Perform a security analysis on the finished container
- Make performance measurements based on the proof-of-concept implementation
- Investigate the performance and security trade-offs concerning the container vs. VM vs. SCU
- Is it possible to replace hardware with virtualization and maintain a sufficient level of security?

1.4 Thesis Scope

This thesis aims to investigate the possibility to convert a hardware solution, i.e. the SCU, into a container solution in such a way that its core functions are usable, with minimal performance and security compromises. Using a container introduces security risks since the underlying hardware and software are different from the SCU and the Axis platform it uses (Axis OS). Thus, modifications have to be made to create a functional container. Such modifications can include mocking services, code modification, and extending container namespace, etc. Due to time constraints, potential shortcuts can be made to ensure a working container at the cost of security, stability, or performance.

This thesis does not address the current security vulnerabilities in the SCU, the body worn manager (frontend), nor the communication between camera and SCU, as it is not a consequence of containerization.

1.5 Problem Description

The issue presented in this thesis concerns the security of transferring custom-built hardware to a container environment and the authenticity of the data it produces. This container is thought to be hosted by a third party (i.e. not Axis in the case of this thesis). As developing custom hardware is expensive, particularly for startups, virtualizing the product can reduce the cost and increase flexibility. However, the security and functionality of the product are affected by virtualization which is something that needs to be considered during development. Although the focus is on the development and the product, other factors are also affected when avoiding developing new hardware. A factor that is positively impacted by virtualization is the environment. Through shorter production chains, reduced need for transportation as well as the reduced or eliminated need for minerals in production.

Publicly distributed containers introduce security risks as the content of the image becomes exposed. Without security enhancements, the content and functionality of the service become easy to analyze, which in turn increases the risk of exploitation. The program(s) and the data produced by the container need to maintain their integrity such that the application can be trusted to produce a legitimate result.

A system is often seen as large, both in terms of its size and dependencies, which contradicts the view of using containers as micro-services. Therefore one should seek a minimal container system to minimize complexity and vulnerabilities. One could also consider shutting down producing hardware and focus solely on alternating the code and structure to better suit a container environment.

The results of this thesis will shed light on whether or not it could be an option to transfer the production, i.e. a system, from hardware to virtualization. Although containerization is rarely planned at an early stage of development, it might aid teams who wish to transition to a cloud-based solution.

1.5.1 Threat Model

The container is executed locally on a computer and it is assumed that the container image will be accessible by anyone. The SCU container is furthermore assumed to be executed with non-root privileges, in its own user namespace, and with mandatory access control enabled. Two types of threats are expected, one from the host and another from other containers. The adversary in both cases can be either passive or active. A passive adversary will eavesdrop on the commutation to and from the container, while an active adversary can influence and tamper with the container's functionality.

As the container image is public, it is expected that the image is pulled and run by a malicious host, and the hardware (CPU, memory, and hard drive, etc) on said host is presumed to be untampered with. Furthermore, the container is also assumed to be executed on a system with other containers, which are potentially malicious. Said container can either be designed to be malicious or have one or several malicious applications running inside it. These containers are also presumed to be run as root.

1.6 Methodology

To gain an understanding of how containers and virtual machines work, a literature study was conducted. This also provided a state-of-the-art description of their security and performance. Another aim of the literature study was to find tools that enhance container security. The results from the literature study were used to perform a theoretical evaluation of whether a VM is more suitable than a container.

To verify the possibility of converting hardware into a virtual form, a container of the SCU has to be developed. The goal is to create a script that downloads files, compiles code, and sets up dependencies automatically so that it is possible to build the image without additional input. The hardware uses services that are not present in Debian. These services, therefore, have to be mocked to fulfill the binary's requirement. After the completion of the container, a proof of concept is conducted to verify that the container performs as its hardware counterpart. When said proof is successful, security enhancements are made based on the result of the literature study, followed by confirmation through another proof of concept.

Finally, a manual analysis is conducted to investigate possible vulnerabilities of the containerized system. This is done by accessing the container from the host to eavesdrop and manipulate files on it. The container image is also analyzed, with the help of a tool, to detect known vulnerabilities.

1.7 Outline

Chapter 2 introduces state of the art for container security and information on Docker, Podman, and VM. Chapter 3 describes the development process and obstacles, and chapter 4 describes the process of implementing security enhancements. Chapter 5 provides a security analysis of the container image, before and

after security enhancements. Finally, chapter 6 presents benchmarks results and chapter 7 discusses the results and summarizes the thesis.

Chapter 2

Technology Background

This chapter provides concept needed to understand how a container work and the security features used. The security enhancements are selected to protect the container according to the threat model in section 1.5.1. Lastly, a discussion on performance and security for a virtual machine is made.

2.1 Containerization

Containers are virtual runtime environments that provide OS-level virtualization. When executed, the kernel isolates the process and defines which and how much system resources it can use, e.g. CPU, memory, network. One machine can run multiple containers where each container is an isolated process and they all share the kernel [3]. Isolation is important as it increases the security of the containers by protecting the host from containers but also protects containers from other containers. In other words, a container has no knowledge of the host or other containers on the machine. This is achieved by using kernel namespaces, which provide a unique environment for the container, and control groups (cgroups), which restrict the resources available to the container [4].

The Open Container Initiative (OCI) is a project under the Linux Foundation to create an open industry standard around containers. The project itself was initialized by Docker, CoreOS (RedHat), and others. OCI contains two specifications that set the standard: *runtime specification* and *image specification*. The former defines how to run a file system bundle, which is the structure created when unpacking the image file [5]. The latter assures a uniform image format, enabling different container engines (based on OCI) to run the same image.

One can create an image with the help of a container file, otherwise known as a docker file. This image describes the container system, its libraries, and dependencies, the programs it contains, how it will run, etc. The image consists of a set of layers that are created in form of a stack, shown in figure 2.1. These layers are read-only and for the container to write, modify, or delete files, it needs a layer that is read and writable. This is satisfied by creating a so-called container layer that conforms to these properties [6]. The container layer persists as long as the container is running, i.e. removing the container deletes all data produced by it.

In addition to one container, it is possible to create multiple containers working

together as one system, i.e. a pod or cluster. It is encouraged to develop a container to serve one function solely, e.g. a nginx server, and nothing else. By using a pod, it becomes possible to combine several containers with different functionality in such a way that they relay data between each other. This is appropriate for modular development which increases flexibility.

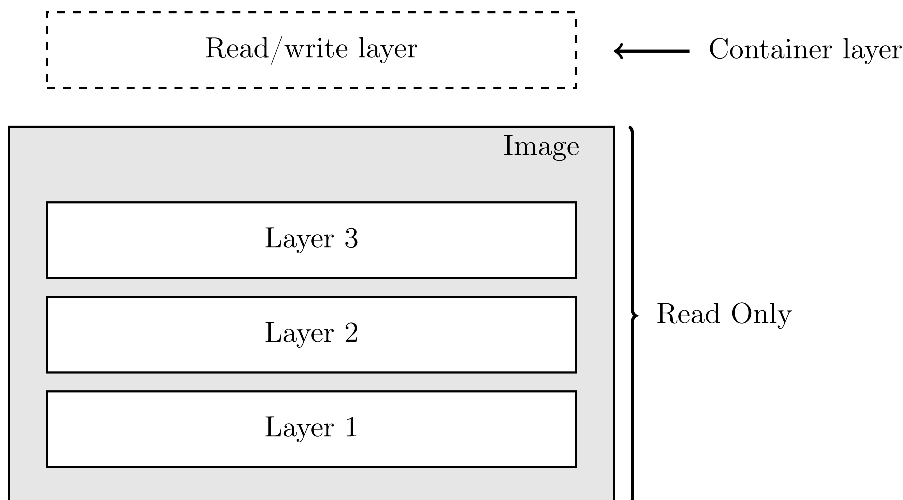


Figure 2.1: The structure of a container image where each layer inside the image is read only and the container layer is read and writable.

2.2 Docker

Docker is a container engine for running container images. It is constructed out of three parts: a client, docker daemon (*dockerd*), and a registry. The use of dockerd results in a client-server architecture as the client sends all commands to it, as illustrated in figure 2.3a. The client uses a console-line interface (CLI) which translates the input to a HTTP package that is sent dockerd [7]. The registry can be viewed as a library of different container images one can access and can be either public or private.

The purpose of dockerd is to run, manage, and monitor the containers, build images from container files, manage images from the registry, and more. The daemon is running as root and is controlled through a Unix socket or TCP [8]. Because dockerd manages all containers, if the daemon terminates, by choice or not, all active containers will also terminate by default [9]. However, dockerd does not directly create and manage containers, but instead, relies on *containerd* to do so.

Containerd is a container runtime daemon that manages the container's life cycle. It manages image transfer and storage, executing containers, namespace, and network. It has a low-level API that is wrapped in dockerd. It is furthermore

integrated with runc to create and run a container. It is also possible to detach containers from contained. In such cases, containerd-shim takes over after runc has initialized the container, and acts as the parent by managing exit status, standard I/O, and can keep the container alive even if dockerd dies [10, 11]. Figure 2.3a illustrates this in the right sub-tree of the containerd node.

Docker can store data in three ways: bind mounts, volumes, and tmpfs mount [12], which is shown in figure 2.2. The latter stores the container data inside memory and as such is never written to the host's filesystem. Tmpfs is only available on Linux and data stored in this way cannot be shared between other containers. Bind mount uses any directory on the system to bind to the container, using it as its root directory. Finally, volume is managed by docker as it creates a new directory under docker where it stores the data from the container. Bind mounts and volumes enable persistent data storage, however, volumes are the preferred way to store data [12]. These two solutions also enable containers to share data, compare to tmpfs. By default, files created in the container are stored on a writable container layer and are therefore not persistent.

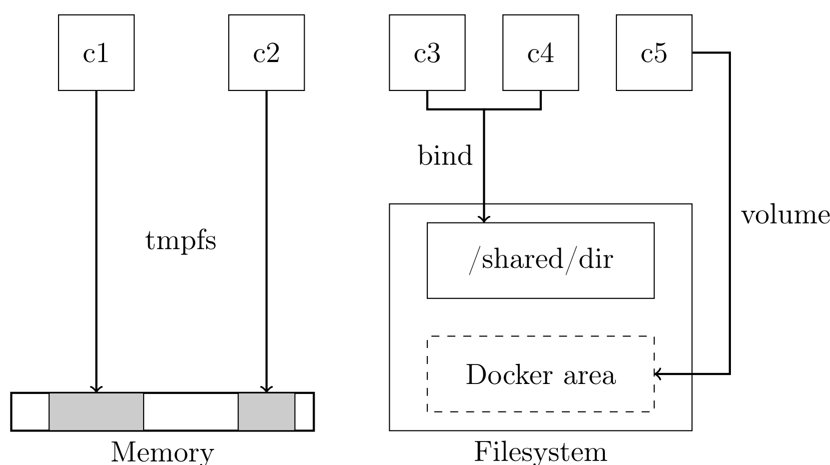


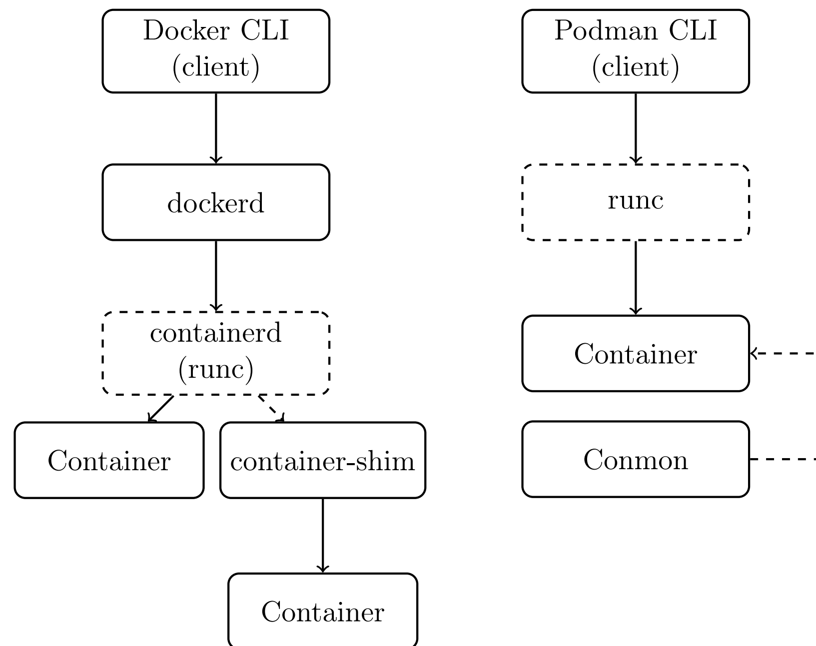
Figure 2.2: The three different type of storage provided by docker. The method using the filesystem are capable of sharing storage while tmpfs are not able to share. C1 to C5 represents 5 different containers.

2.3 Podman

Podman is an open-source container engine based on the OCI and mainly developed by RedHat. The motivation behind Podman was the need to avoid a root daemon, which Docker does. Instead of relying on the client-server architecture that comes with the daemon, Podman uses a fork-exec model which integrates more naturally into Linux [13] and avoids potential security risks associated with the client-server structure. However, since it does not have a daemon, it instead uses

conmon for monitoring and manages communication with the containers [14, 15].

Conmon is a C program that is required to watch the primary process of the container. Conmon also holds the teletypewriter open which allows the user to send standard I/O to the container. Furthermore, it allows Podman to exit and run in detached mode while the container still runs [16], i.e. the Podman process can quit while still enabling the container to run. The architecture of Podman can be viewed in figure 2.3b. When the container is launched, conmon is also launched.



(a) Docker container execution architecture. Dockerd calls on containerd which uses runc in order to run a container.

(b) Podman uses a fork-exec approach which launches conmon attached to the container. It starts containers by using runc as its container runtime environment.

Figure 2.3: Architecture for Docker and Podman

Podman can use two different container runtime: *runc*, which is used and originally developed by Docker, and *crun*. The latter container runtime is written in C, compared to runc which is developed in Golang, to have better performance and lower memory footprint. Crun is implemented according to the OCI container runtime specification and according to crun's github repository, running 100 sequentially containers is 49.4 % faster with crun compared to runc [17]. The increase in performance is welcomed by the high-performance computing community where an empirical study shows that the use of crun reduced the mean overhead from 5.10 % to 2.04 % [18] when simulating impacts of automotive crashes, explosions, and sheet metal stamping.

Podman is a rootless container engine which is possible due to the user namespace mechanism in the Linux kernel. It allows Podman to execute containers

without escalating privilege and as such, processes inside the container can be root, but outside it is running as a non-root user [18, 19]. This increases security since executing containers as non-root results in non-root host access if a container escape occurs. Additionally, it also enables containers to isolate themselves towards other containers by using different user and group id [19].

2.4 Namespaces

Namespace is a feature in the Linux kernel which aims to provide processes with isolation. There are in total seven different namespaces which can be viewed in table 2.1. Resources in one namespace are invisible from another non-parent namespace, i.e. namespace can be inherent to sub-processes, which gives the containers their isolation. Different container engines use different namespaces by default and some can be opt-in. For instance, Docker and Podman use runc, which according to [20] uses all namespaces by default except user namespace which is optional for Docker and enforced in Podman. Some resources are not bounded by namespaces, such as devices [21], meaning all containers can access any device on the system. The container engine performs the setup for the namespaces automatically upon container creation [22].

Table 2.1: The types of namespaces the Linux kernel supports [23].

Namespace	Isolates
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports
Mount	Mount points (file system)
PID	Process ID
Time	Boot and monotonic clocks
User	User and group IDs
UTS	Hostname and NIS domain name
Cgroup	Cgroup root directory

When creating a container, the user id (uid) on the host will be mapped to the uid on the container. This can be an issue if the containers are run as root and a container escape occurs. This results in the user having root access to the host machine. Therefore, it is ideal to run containers as non-root, such that the container will map to the user on the host instead of root. For example, if a root user on the host creates a container and the user on the container is root, then if escaped, the container user will have host root access. On the other hand, if the container is created as a non-root user while still being root in the container, an escape would yield user access on the host instead [20]. This is visualized in figure 2.4.

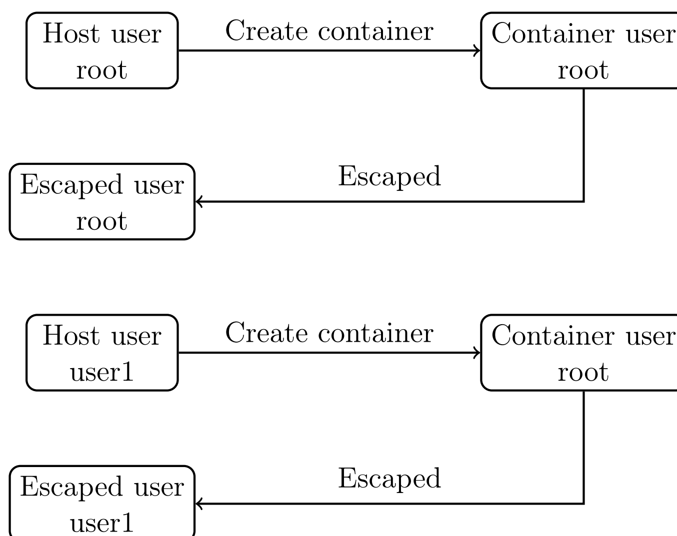


Figure 2.4: Visual of how user namespace can mitigate root access on host during a container escape.

2.5 Cgroups

Control groups (cgroups) manages and restrict the system resources among processes. There are two types of cgroup versions, namely versions 1 and 2. These two version functions differently from each other in a significant way, however, both follow a tree structure in defining restrictions. This thesis focuses on version 2.

Version 2 of cgroup is designed with a core and controllers. The core is the root of the tree and possesses all controllers available to the system. The controller limit and monitors which type of resources a cgroup can use [24]. The processes affected are located as the leaves under a cgroup as figure 2.5 shows. The child cgroup inherent its parent's restrictions at the same time as the parent dictate which controllers the child is allowed to have control over. For instance, if the child inherits the CPU controller, then it can further limit the CPU, but it cannot ease the restriction set by its parent. The creation of cgroups is mainly managed automatically by systemd, libcgroup, container engine, among others, but can also be created and configured manually [25].

Cgroups offers availability to the containers. Without a system to limit resources, a container could be used as a denial of service. For instance, a container could use all CPU or memory which will starve other processes on the system, including other containers [26].

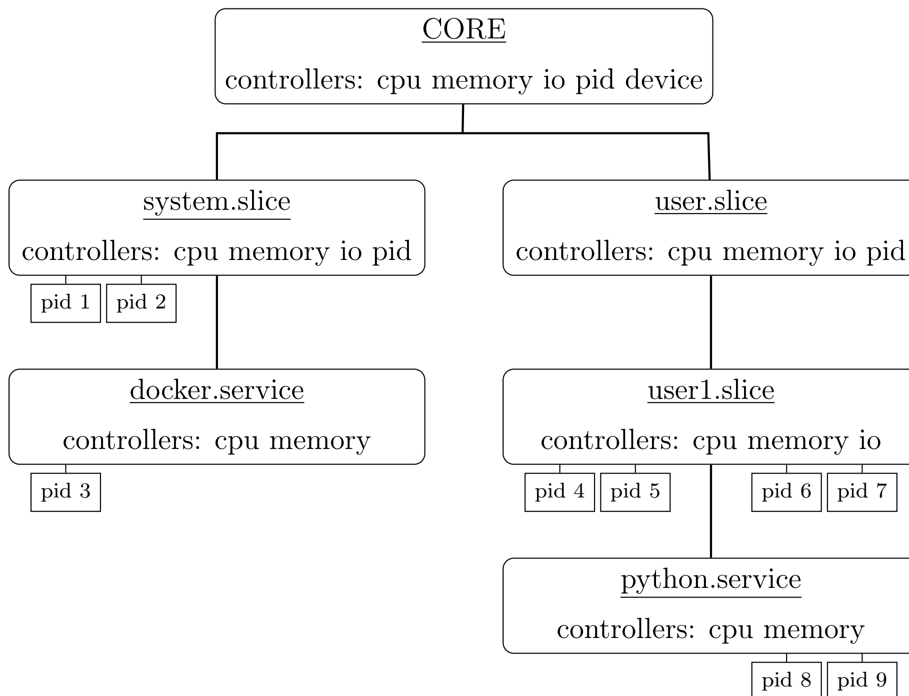


Figure 2.5: Illustration of the tree hierarchy in which cgroup is based on. The processes under a cgroup, e.g. system.slice, are leaf nodes.

2.6 Mandatory Access Control

Mandatory access control (MAC) is a method of restricting access to certain files, directories, sockets, and more, by enforcing a policy. This policy defines which processes are allowed to access a particular resource where the restriction can depend on multiple factors. In what manner they allow access is also defined, i.e. if a process can, for instance, read or write to it. Users are unable to override the policy. Mandatory access control can be illustrated with key cards to a facility. Certain people will not have access to a part of the building, rooms, or even documents, depending on the clearance level their key card holds. Tools such as SELinux and AppArmor use MAC and are discussed in the following sections.

A MAC could make it more difficult for a malicious container to perform a successful attack on a legitimate container (depending on the policy). Furthermore, it protects the host from containers as the processes inside a container will be restricted to themselves.

2.7 SELinux

Security-Enhanced Linux (SELinux) defines MAC for applications, processes, and files, by using the Linux security module (a framework integrated into the Linux kernel). When enabled, it can run in either *permissive* or *enforcing* mode, where enforcing blocks access on policy violation, and permissive only logs them. SELinux system defines subjects (processes) and objects (files, sockets, etc.) where the subject performs actions on an object, such as read, write, and execute. Each subject and object are assigned a label which defines their: user, role, type, and (optionally) level [27], and is written on the form *user:role:type:level*.

When a subject performs an action on an object, a request is sent to an object manager which queries an *access vector cache* (AVC) that stores recent policy look-up. If a cache miss occurs on the policy, then the AVC forwards the request to the security server where it queries the security policy for the requested action. The reason for the AVC is to increase the performance and thus reduce overhead [28], provided the number of cache misses is low [29]. An overview of the request process can be seen in figure 2.6.

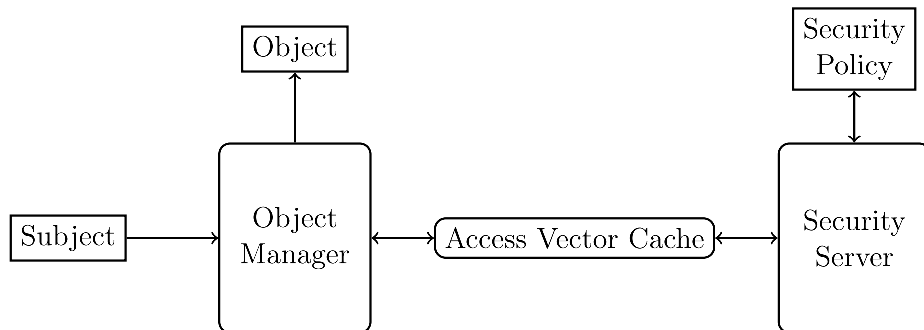


Figure 2.6: Core decision making SELinux architecture. The subject requests access to the object. The object manager first queries the AVC, secondly searching the policies, and finally returns allow or deny.

SELinux defines MAC in several ways, although the most common is Type Enforcement (TE) where the policy is based on the type attribute and is defined as

```
allow user_t bin_t : file {read write}.
```

This tells SELinux that user_t is allowed to perform the actions *read* and *write* on file belonging to the object bin_t. All requests will be denied unless there is a SELinux policy rule which allows an action.

Multi-Level Security (MLS) is another way for SELinux to define MAC. It is based on the level attribute and provides restrictions on information handling between different confidentiality levels [30]. In its simplest form, it can be thought of as an access level, where one needs a certain clearance level to be allowed to view certain documents, figure 2.7 illustrates this. Furthermore, MLS is considered

more advanced to use, while TE comes with default policy settings that covers a wide range of applications, tasks, and services [31].

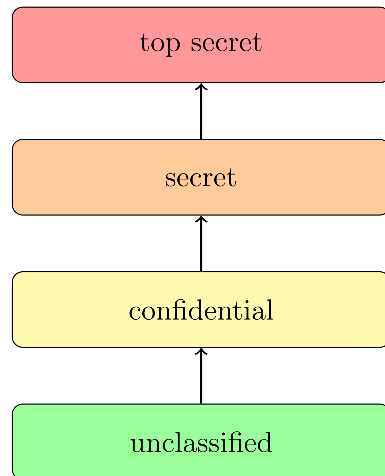


Figure 2.7: The basic concept of MLS illustrated with clearance level. One with the level confidential can view its own and all levels below, but not levels above itself.

While SELinux provides robust security frameworks, it falls short on user-friendliness. This impacts the security it could provide as end users are not able to set it up correctly [32] since it requires expertise and is, therefore, more suited towards experts.

2.8 AppArmor

Application Armor (AppArmor) is another mandatory access control scheme that utilizes Linux security module. It uses profiles for applications to restrict access based on the path. Like SELinux, it uses the MAC in two ways: enforcing, and permissive.

2.8.1 Lic-Sec

AppArmor is easier to use than SELinux, especially when considering using tools that generate the profiles. One such tool is *Lic-Sec* introduced by Zhu and Gehrmann [33] which is based on Docker-sec [34] and the older LiCShield [35]. Lic-Sec combines capability and network rules from Docker-sec with pivot root rules, file access rules, mount rules, and more, from LiCShield.

Lic-Sec collects data by tracing which *operations* are performed by Docker and the container. The data is later used to generate new profiles for AppArmor. It traces SystemTap and Auditd in parallel in such a way that the former collects data from Docker components and the latter collects data from mount, capabilities,

and network inside the container. After the trace is done, the results are analyzed and output the new profile.

It is possible to use Lic-Sec in three different modes. Each of the modes initiates the trace at different times during the container startup or running process. The first one initiates the trace of SystemTap before dockerd is launched and continues until stopped. The second one executes at runc, and the third is activated when the container is running [33].

Performance suffers slightly overall with increased overhead. The main reason for this according to Zhu and Gehrmann is the analysis of both SystemTap and Auditd. Despite the increase in overhead, the tool provides higher security in the tested containers. Lic-Sec successfully mitigated 8 kernel exploits compared to Docker-sec and had the same performance versus userspace exploits.

2.9 vTPM

As the name suggests, the virtual TPM (vTPM) is a software implementation of the Trusted Platform Module (TPM) hardware component that resides inside computers, smartphones, servers, etc. A TPM provides hardware support for an array of features such as encryption and decryption, attest hardware and software, and provide a root-of-trust as the TPM can always be trusted. The TPM is attached to a specific device and if the device is tampered with or the TPM is replaced, then the TPM will not work as it attests the entire device [36].

A vTPM could enable encryption of the container's content, something which is otherwise not possible to do securely as the keys will be accessible to the host. The keys generated by the vTPM will be available only to the container and cannot be accessed or used directly by a host. This could prevent a host from modifying or faking content as the data is be attested by the vTPM.

A requirement for the container is that the footage remains untampered with, which can be achieved by using a TPM. However, the container cannot utilize the TMP as it is bound to the host. It requires a vTPM instead that acts as a bridge between the container and the TMP hardware. This would secure the encryption keys used for video and hard disk encryption as they are stored inside the TPM instead of the system hard drive. The vTPM was first designed for VMs where each VM should have access to the vTPM. More specifically, S. Berger et al [37] presented the following requirements for a virtual TPM:

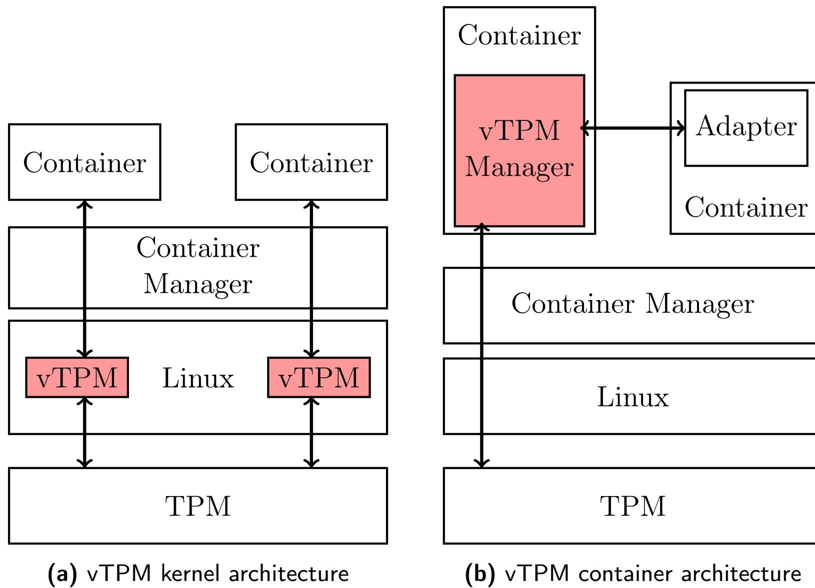
1. A vTPM function and commands towards software must be that of the TPM to the OS on the hardware.

2. The VM and its vTPM must have a strong association between them such that secrets inside the vTPM cannot be accessed outside by others and migration is possible.

3. There must be a strong association between the vTPM and the trusted computing base.

4. The vTPM is distinguishable from the TPM due to security differences.

As of the writing of this thesis, there exists no implementation for vTPM for containers. However, S. Hosseinzadeh et al [38] proposes two ways to implement a vTPM based on the requirements S. Berger et al [37] proposed (as listed above). The two types of architectures can be viewed in figure 2.8.



(a) vTPM kernel architecture

(b) vTPM container architecture

Figure 2.8: The two different architectures for virtual TPM for containers as proposed in [38].

The first architecture (figure 2.8a) is to have a kernel module that can produce an arbitrary number of vTPM. The generated virtual TPM then communicates with the physical TPM and with the associated container. This design satisfies all requirements for a vTPM under the assumption that the container is isolated and cannot access the host OS. This results in the container being able to attest its state (if the host OS can be trusted), and ideally, the container could attest the host OS as well. The virtual TPM is better protected (but not as well as a physical TPM) if placed in the kernel compared to having it in user space as per the second architecture.

The second architecture (figure 2.8b) uses a designated container with software support for the vTPM and uses it to delegate requests to other containers on the system. The container with the virtual TPM communicates directly with the hardware TPM. This architecture is easier to implement compared to the kernel version. However, it also requires a daemon to be associated with the underlying trusted computing base. The risk with this design is if another container gets access to the host OS, aka container escape, it can then attack the vTPM container.

2.10 Intel SGX

Intel Software Guard Extensions (iSGX) allows applications to ensure confidentiality and integrity via CPU extensions, even if the underlying system is malicious [39, 40]. It is designed to provide applications with confidentiality, integrity, and to protect execution. Intel SGX uses *enclaves* to achieve this and is defined as a private part of memory that encrypts and isolates parts of a program. While inside an enclave, no other process, despite the level of privilege, can access the content within it, however, the enclave can still access content outside of itself. Although iSGX provides protection on untrusted hosts, it is still exposed to attacks such as side-channel attacks which aim to obtain information through other sources than, in this case, the CPU. Examples of such exploits are controlled-channel attacks [41] and last-level cache attacks [42].

2.10.1 SCONE

Arnautov et al [39] presented SCONE, secure container environment, which utilizes Intel SGX to construct a container that can be used on untrusted platforms. The result is increased security by narrowing the trust base, while also having low overhead. This mitigates the risk of adversaries exploiting containers as the execution is now done with enclaves. The architecture of SCONE can be viewed in figure 2.9.

SCONE provides their solution via containers, from base images to tools. One of these tools, *sconify image*, takes a native image and protects a binary, data, and codebase. The binary and files need to be specified by the user. Sconify image can be used with two different modes: file protection and binary file protection. Formerly, the binary is used inside an enclave and additional data provided becomes encrypted. Latterly, all associated files are together with the binary placed inside an enclave.

Both of the modes place the provided files and binaries into an encrypted image as figure 2.10 illustrates, where sconify binary is a process that enables the binary to run inside an enclave and use SCONE services. Furthermore, it creates a security policy associated with the container image, it defines the secrets and how they are distributed, which part of the container image is encrypted, which users are allowed to read or modify the policy, and more.

The policy is uploaded to an attested configuration and attestation service (CAS) provided by SCONE. The CAS runs within an enclave and therefore protects the policies. Its core purpose is to securely maintain keys and secrets from non-authorized entities, i.e. users, adversaries, or other containers. CAS provides secrets to an enclave, according to the policy, after verifying its integrity and authenticity, enabling it to execute as intended [43].

SCONE implements the C standard library (libc) interface to be able to execute unmodified processes, i.e. unmodified in the sense that it does not require additional code or configuration to work with Intel SGX. Libc must be able to perform system calls, which cannot be done inside an enclave, and therefore requires an *external interface* to use as a bridge between the enclave and the host OS. This interface is protected by shield libraries which use transparent encryp-

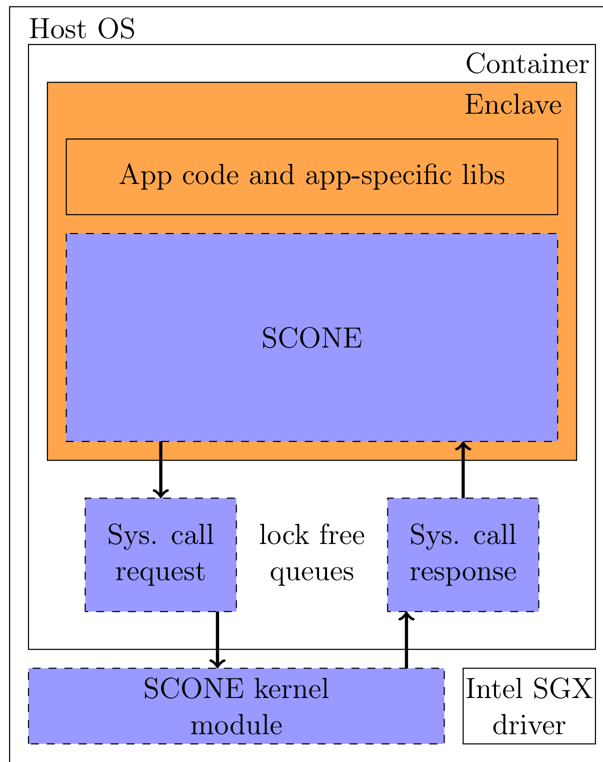


Figure 2.9: SCONE architecture. The orange zone is trusted, the blue zones are SCONE components, and the white zones are untrusted.

tion (data is encrypted at rest, but not while used or in transit) on system calls concerning I/O on files. The external interface also performs sanity checks and copies memory-based return values before passing the arguments to the enclaves to protect itself from user space attacks.

The shields prevent low-level attacks and ensure confidentiality and integrity of data passing through the OS. There are support for three types of shields: *file system shield*, *network shield*, and *console shield*.

The file system shield ensures confidentiality and integrity of files as they are authenticated and encrypted seamlessly to the service. It defines three rules which determine if the file is authenticated, encrypted (or both), or just passed on to the OS. It also contains the same support for ephemeral files, i.e. files existing only for a short period.

The network shield guarantees that the container uses TLS when communicating over a network as it enforces a client to use it. This was added since services such as Redis and Memcached do not use TLS (compared to Apache and Nginx), but assumes the traffic is protected by other means. SCONE redirects all socket operations to the network shield, which performs a TLS handshake and encrypt/decrypts the packages.

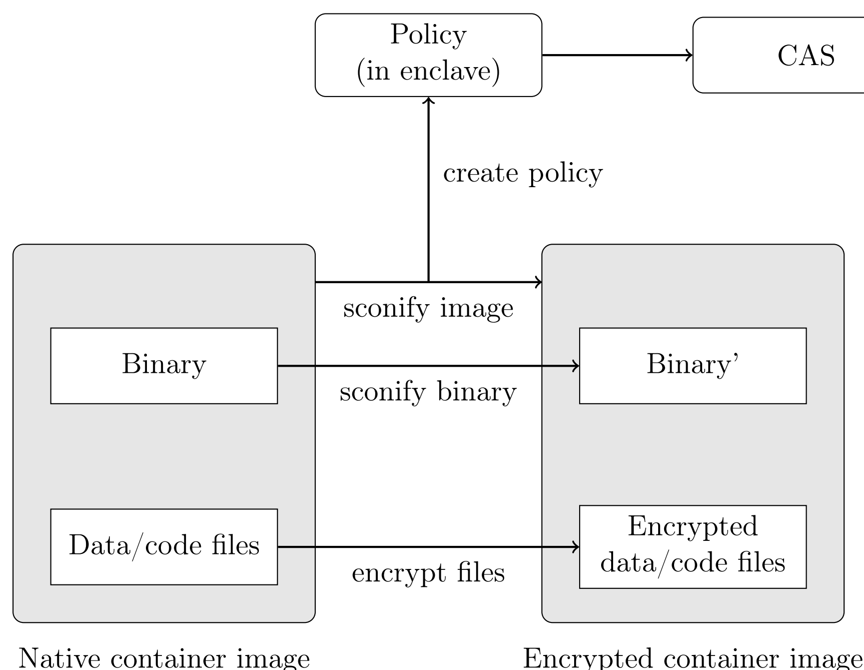


Figure 2.10: The sconification process with file protection mode. Parts of the native image are selected for protection and placed on a new, encrypted, base image. The CAS verify and holds its key to use when running the container image.

The console shield is designed to protect application data during *stdin*, *stdout*, and *stderr* console streams. Since the data is transparently encrypted, it uses a symmetric key to encrypt the stream by dividing it into blocks that are given a unique identifier. This protects against replay and reordering attacks.

As of writing this thesis, SCONE requires an image based on Alpine Linux to apply its security features, however, support for Ubuntu is under development.

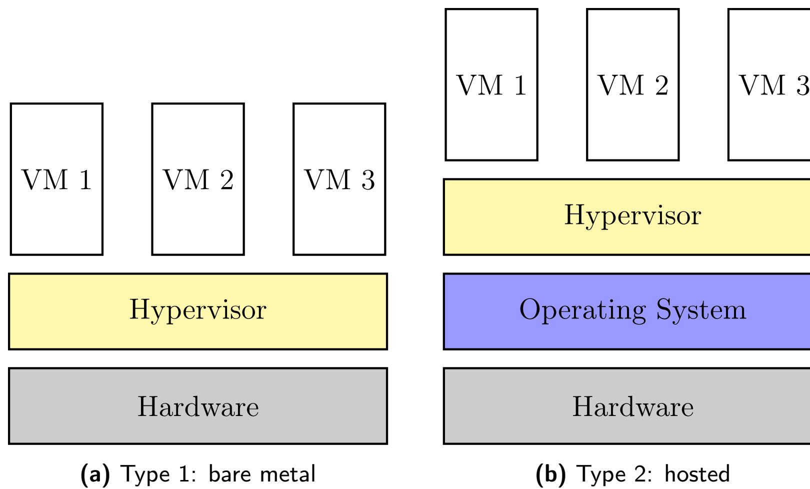
2.11 Virtual Machine

Virtual machine (VM) was the first way to achieve virtualization by using a hypervisor. The hypervisor handles scheduling, manages memory, I/O, and network, but also works as a bridge between the guest OS (i.e. the VM) and the host OS including the hardware [44]. The VMs are isolated from the hardware and are therefore the hypervisor's task to communicate requests from the VM to hardware. A hypervisor can host multiple VMs on a local machine where the hypervisor separates the VMs with firewalls.

There are two types of hypervisors, as can be viewed in figure 2.11, bare metal (type 1) and hosted (type 2). The former type integrates the hypervisor above the hardware with no underlying OS, as such, it gains direct access to the hardware.

It can also be embedded directly into the hardware. This is ideal for companies and cloud services as it is more lightweight and has increased security due to the absent operating system [45]. The second type, hosted, is installed as a software on top of an OS. The hypervisor must use the host OS to access the hardware. One advantage over bare metal is that it is easier to install, use, and maintain a hosted hypervisor [46] in addition to using the operating system's memory management, process scheduling, and other OS features.

The following two following sections will discuss VMs and compare them with containers.



(a) Type 1: bare metal **(b)** Type 2: hosted

Figure 2.11: Two types of hypervisor technologies.

2.11.1 Performance

This section discusses the performance between the two types of hypervisors, bare-metal and container, as well as between a hosted VM and a container. All of the hypervisors mentioned are compatible with Linux.

Beserra et al [47] explored if VMs are suitable for high-performance computing (HPC). The paper investigates Kernel-based virtual module[1] (KVM) and VirtualBox[2] where the former uses a bare-metal hypervisor and the latter a hosted hypervisor. HPC-challenge benchmark was used to evaluate the processor, RAM, process communication, and network communication. The paper concluded that KVM outperforms VirtualBox with near-native performance in most cases. The bottleneck presented for both hypervisors was local memory and communication between processes.

Poojara et al [48] conducted a performance comparison between Linux contain-

[1]https://www.linux-kvm.org/
[2]https://www.virtualbox.org/

ers[3] (LXC) and Xenserver[4], a bare-metal VM, where the objective was to compare CPU, RAM, hard disk, and HTTP web server, with the hardware as a reference point. The container outperformed the Xenserver in all cases while being close to the hardware's performance. Nevertheless, in most cases, the VM was comparable to LXC. Furthermore, the container was 4% slower than the hardware when benchmarking RAM read and write, with the VM being slightly slower. Finally, the largest difference is the number of requests per second on a http web server where the container was 12% slower than the hardware, and the VM falling closely behind.

Similar results from Poojara are also seen in a performance overhead comparison by Li et al [49], where they examined communication, computation, memory, and storage, through Domain Knowledge-driven Methodology (DoKnowMe) and on a feature-by-feature basis. The comparison was between Docker and VMware Workstation Pro[5], which is a hosted hypervisor. They found that the performance of the container is on average better than the VM with near hardware performance. An interesting find is the storage transaction speed which gave an overhead up to 50% for the container and hardware-like performance for the VM when solving the N-Queens problem or writing small-sized data to the disc. Furthermore, the authors argue that the performance of two types of virtualization depends on the type of feature and job it is performing, as such one is not better than the other performance-wise.

2.11.2 Security

The usage of VMs is substantial in cloud networking. The security of virtual machines is crucial as one machine may host VMs for several companies. The threat model for VMs is quite extensive with possible attacks from other virtual machines, the hypervisor, and the host. If an attacker manages to perform a VM escape, it is then able to attack another VM through the host or hypervisor [50]. An example of VM escape is by memory corruption like *Cloudburst* [51], which enabled the attacker to execute malicious code on the host that created a backdoor to it. Although a VM is considered more secure than a container, due to the isolation it provides, it is still vulnerable. Documented vulnerabilities of the virtual machines mentioned in the previous section can be seen in table 2.2. It is worth noting that KVM most likely contains vulnerabilities, even though they have zero registered from 2019 to 2020.

The hypervisor plays a vital role in VM security as it creates, manages, and handles communication with the VMs. The reason for VMs to be considered secure is the smaller code base hypervisors introduces, compare to operating systems, which results in a smaller attack surface [52]. Despite this, virtualization solutions from companies such as WMWare (Workstation), Xen (Xenserver), Oracle (VirtualBox), are affected with vulnerabilities. There have been research due to this with the hope to increase security outside the hypervisors. For instance, a tool to enforce mandatory access control, similar to AppArmor, was developed to prevent

[3]https://linuxcontainers.org/
[4]https://www.citrix.com/
[5]https://www.vmware.com/

VM escape [53]. Research has been made on intrusion detection and prevention mechanisms, like CLARUS [54]. CLARUS is a system that monitors a virtual machine. It is up to the user to analyze the data it collects to detects abnormal patterns. There is also research on preventing zero-day attacks, i.e. exploits of unknown vulnerabilities, such as *ferify* [55]. Ferify prevents zero-day on a set of user files and kernel operations which it aims to protect by catching system calls. The system calls are then analyzed and either accept or deny depending on the user or group.

Table 2.2: Number of vulnerabilities for different virtualization products from year 2019-2020 according to NIST's NVD.

Product	Number of vulnerabilities
KVM	0
VirtualBox	125
Xen	68
Workstation Pro	5

Side channel attacks

While the VMs are isolated from each other, they also share resources, such as memory, cache, and I/O. This opens up for side-channel attacks where the vulnerability lies in the information the VM shares with other virtual machines. With it, it is possible to extract cryptographic keys [56], disclose files or applications running inside another VM [57], and leak memory from the victim's memory space [58]. Note that containers are also susceptible to side-channel attacks.

A feature VM has is memory deduplication. Memory deduplication enables VMs to share virtual memory if they use the same data. Using memory deduplication comes with two advantages for cloud providers, lowers power consumption and increases the number of VMs a server can host, meaning they can reduce cost while maximizing the amount of VMs provided. However, memory deduplication is susceptible to memory disclosure attacks, as the authors of [57] show. The suggested attack makes it possible to obtain the contents within the memory by measuring the time it takes to write to the memory. Since memory deduplication needs to copy on write, this action takes longer compared to non-deduplication. One could therefore guess which applications a victim VM use by using said application. For instance, a malicious VM could run Firefox and when the victim VM runs the same application, write time to the memory will increase. In the case of Firefox, it is possible to read the content cashed by the victim's VM. In other words, one could know which pages it visited.

To mitigate this risk of memory deduplication, most cloud providers have disabled it, however, Taehun and Youngjoo [59] proposed a method that would mend the security implication of memory deduplication. By embedding a fixed random byte to the applications one wishes to secure. According to the authors, should make it infeasible to extract the data without knowing the value within a

reasonable time (years). This is due to the extra byte altering the data in such a way that it never becomes shared in memory deduplication.

Another memory leakage weakness was presented by the authors in [58]. They show how to leak the memory of KVM by utilizing spectre attacks. The spectre attack exploits the processor's speculative computation. The CPU uses its idle time (i.e. waiting for data) to pre-process the outcome and guessing where the data will appear in memory. If the guess is incorrect it discards the calculation. Although this is mainly a hardware vulnerability, VMs are still able to exploit it, which in turn should make containers able to exploit it as well. Nevertheless, there are both hardware [60] and software [61] mitigation for the spectre attack. The former extracts instructions and uses machine learning to detect spectral behavior, and the latter is a software that analyzes code and performs sanitation on it before execution.

Chapter 3

Containerizing the Hardware

This chapter describes the process, obstacles, and solutions, along with motivations, to develop the container image. In the overview, a general explanation is given to provide an understanding of the transformation process, the major components, the main obstacles they introduced, along with the results. The sections 3.2 and 3.3 provide a more detailed description with motivation to design choices regarding the main programs and the server transformation. Section 3.4 discusses the modifications done on a functional container image.

3.1 Overview

Two factors were considered to be able to transform the hardware into a container. Firstly, satisfying the requirements of the programs running on the SCU, and secondly, configure the server. Figure 3.1 provides an overview of the content in the container image. The physical SCU is built on Axis OS, a custom OS that is a

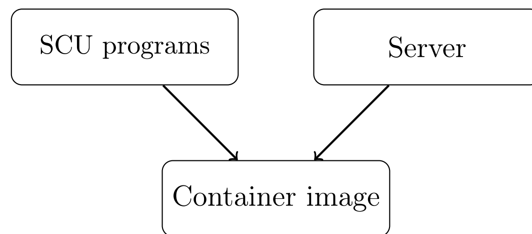


Figure 3.1: The main components used to produce the container image.

subsystem of Debian. This operating system contains services utilized by the SCU that are absent on Debian and became the major obstacle as the SCU programs rely on these services. Furthermore, the SCU heavily utilizes *systemd*, a system and service manager, to handle communication and manage program statuses within the system. The base image, i.e. the foundation, used when constructing the container image contained Debian 9 as its OS, and thus uses neither Axis OS nor systemd. Without Axis OS, the system lost several services required. These services were given by a mock script that provides a static response upon a system

call. The absent systemd could be installed via the advanced package tool (apt). It would be possible to modify the SCU code to avoid a mock service, however that would require two different implementations for the same product, which is redundant.

The container had to communicate with the BWC while also hosting a web page, i.e. the BWM, to provide an interface to the user. This was done by configuring a server to serve both the BWM and to be able to communicate with the camera, on different ports and addresses. The default SCU server configuration uses server modules and features only available on Axis OS. Furthermore, it also uses privileged ports, something a non-root container cannot use. As such, several configuration files were modified, via a script, to remove or redefine lines. Unfortunately, this was the only way to make the server run, although it would ideally be avoided to minimize maintenance.

Finally, an initialization script was constructed to start the mock service, as well as a script that aided the setup wizard process (performed manually). This initialization script is the first thing systemd started, which in turn started the mock and setup services, and created a server certificate. Figure 3.2 illustrates the final system architecture where systemd is the root. Table 3.1 summarizes all modifications necessary to acquire a functional SCU container where some entries are discussed further in section 3.2 and 3.3.

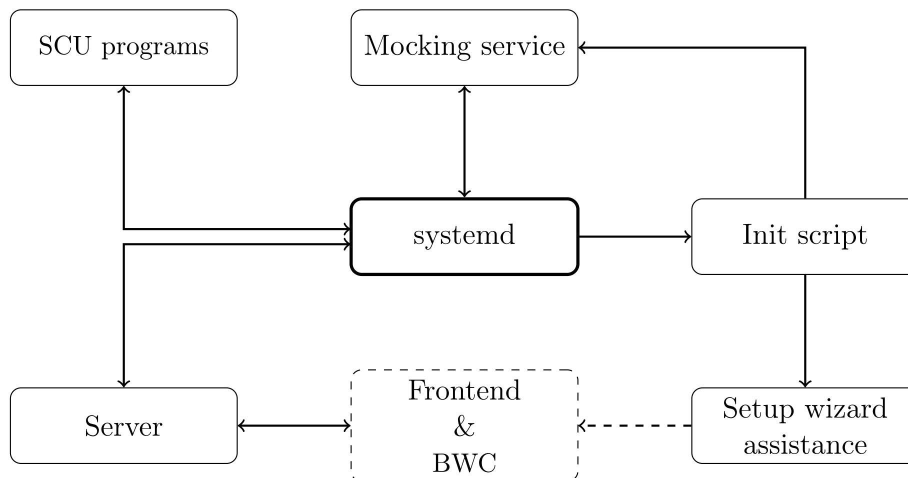


Figure 3.2: The container architecture. Systemd starts the processes and handles their communication. The initialization script is the first thing it starts which in turn launches the mocking and setup service. Frontend and BWC is outside the container and the wizard assistance is required to complete the setup in the frontend.

Table 3.1: Summery of adaptations required to achieve a functional SCU container.

Adaptation	Solves
Mock service	Absent Axis services
Setup assistance	Upload and install firmware
File modification	Satisfy server configuration
Container use host network	BWC communication
Host network modification	BWC communication

3.2 Running SCU Programs

The SCU programs are written in Go and the container image was therefore based on golang:1.15-buster. The code was copied onto the image and compiled during the image-building process. The entry point of the container was set to launch systemd, however, this binary was initially absent. The package manager apt was used to install systemd, as well as dbus (which is used to send messages between applications), on the container. The reason for this is because the Axis binaries rely on systemd and dbus to communicate with other software, and cannot work without them.

Using systemd and the golang image is not ideal due to its complexity and large size (the resulting image was over 1 GB). However, because the physical SCU uses a subsystem of Debian, it was not investigated whether or not it was possible to use something else. Furthermore, a previous container development of similar nature at BWS was developed on the golang image, which made it a natural starting point.

Executing the SCU program resulted in errors due to services not found, absent directories, non-existing users and groups, and incorrect file permission and directory ownership. The main issue, however, was the missing services. These services are integrated into the Axis OS which the SCU uses, and were therefore needed to be mocked. Through investigation of the code and the SCU, it was possible to determine the requirements for a particular service. This included expected input and output, as well as where to find the service.

The mock service was provided via a python script which was required at the start. The values used in the mock script were copied from a SCU and might not be optimal for the container. For instance, the mocked service regarding disk space is static and might not reflect the assigned disk space given by the container engine. The container is, by default, given 10 GB and as the camera can record about 55 GB, this becomes an issue. While the physical SCU tells the BWC to pause footage transfer when its storage is full, the container becomes unable to do so due to the static setting telling the programs it always has space. Nevertheless, it is possible to set the storage of the container so that it matches the BWC. Furthermore, the container (and a physical SCU) remove the footage after uploading it to an EMS. The only dynamic service is the time, which always returns the current local

time. The time is important since the SCU and BWC sync their time to verify the content, check certificates, and more.

Building the image by copying code to the container was redundant. It was time-consuming and it resulted in an unnecessarily large image size and left code behind. To avoid this, all programs were compiled locally whereafter the binaries were copied to the container.

3.3 Server Setup

A server binary had to be installed to provide camera communication and the BWM (user interface). The server installation included several unnecessary configuration files which were removed to avoid interference from the default server configuration. These configurations were then replaced by Axis' server configuration files which specified how the communication to the camera and BWM should behave.

The SCU server uses modules and settings that are not available to the container. Therefore Axis' configuration files had to undergo modification to suit the container. Ideally, this would be avoided since it creates maintenance when additions or alterations are made to these files. Nevertheless, this process was automated with a script whose job was to remove or modify lines in multiple configurations files.

The server had to redefine who was allowed to access the web page, from a restricted group to everyone. Furthermore, it also had to change the port used for http and https for the server to provide the BWM. Running Podman as a non-privileged user enforces restrictions, one such restriction is the denied access to privileged ports (0-1024), and seeing as the server uses port 80 and 443 for http and https respectively, it had to be altered.

An additional interface was required to communicate with the BWC since the programs use a specific IP address to handle communication between the camera and the SCU. For this to be possible, an additional network interface had to be created with the appropriate name and IP address to match it.

By increasing the network capabilities for the container, it became possible to manually add the required network interface inside the container. This was however an undesirable solution as it forced the user to modify the container. Therefore the same commands were integrated with the container file to automate the process. However, Podman could not construct the container image this way as the capabilities of a non-root user restrict them from modifying the network. Instead, one had to construct the image as a root user, something which was avoided with regards to the threat model.

The container was instead allowed to use the host's network stack and interface to provide the two interfaces. By inserting a network card into the computer and configure its address accordingly, it was possible to provide a connection to a BWC.

3.4 Finalizing the Image

A full proof of concept realization was implemented and verified. Afterward, efforts were made to trim the container image to reduce the size and decrease the number of dependencies. By changing the golang base image to a Debian buster base image, the image size reduces significantly, with the base image being 840 MB compared to 114 MB in size, along with unnecessary packages such as secure shell protocol (ssh) being excluded. The Debian image was originally based on the slim version but later changed to the non-slim version to benefit position independent executable (PIE) of the Axis binaries. This allows for address space layout randomization, a technique that randomizes the memory locations used, mitigating the use of return-oriented programming attacks, which is important since the host is seen as the largest threat.

Chapter 4

Increasing Container Security

As the host is suspected to be the major threat against the container, tools which could isolate and protect the container on the local machine was deemed most valuable. One of those tool is SCONE (discussed in section 2.10). By utilizing SCONE's ability to encrypt, attest, and isolate parts of the image, hope was that it could be integrated such that it was possible to run the container safely on a malicious host. In section 4.1 the implementation and expected results are discussed further.

Mandatory access control (discussed in section 2.6) can be used to protect the containers from other containers as well as the host from a container. Since manually configuring a profile for SELinux and AppArmor is both time-consuming and prone to misconfiguration, Lic-Sec (AppArmor) was a good candidate to use. The reason why AppArmor was chosen over SELinux is the difficulty of configuring SELinux in such a way that its full potential is used. Therefore, a manageable MAC was used instead. Section 4.2 discusses the implementation and results.

4.1 Sconify the Native Image

To utilize SCONE, the container had to be converted to Alpine Linux, which is a minimal operating system focused on being lightweight and secure. Due to this, Alpine does not use systemd, instead, it uses OpenRC which is another type of initialization system. To use SCONE, the container image had to be converted to use Alpine Linux. However, when working with converting the container, it quickly became apparent that it was not possible. To use Alpine, one would have to redesign the entire product.

Fortunately, SCONE[1] was developing support for Ubuntu, and indirectly Debian. They provided access to an early version of the tool for this thesis which made it possible to apply SCONE on the container image, without alterations.

The two directories, *custom-scripts* and *var*, were chosen for encryption, along with the binary to the main program to be SGX enabled. Var was encrypted to protect footage uploaded to the container, while custom scripts contain help code to make the container run properly. When running the tool *sconify image*, it yielded an error saying that the native image is not musl LibC (Alpine) or GLibC

[1] https://scontain.com/

(Ubuntu) based. Sconify image assumes that the native image is based on Alpine. To account for this, one could specify the base image used and by providing Debian Buster as its base image, the tool ran successfully and gave the encrypted image as output. Interestingly, the encrypted image was only 282 MB, in other words, smaller than the native image.

Although running the container was expected to fail, as docker cannot run the native image, it failed for another reason - missing binary for systemd. Another attempt to encrypt the image was made with the systemd binary instead of the main program. However, running the new image resulted in the same error, most likely due to systemd binaries missing files to work properly.

A final attempt was made by providing all files and binaries essential to the container. However, as the system is large, while also depending on systemd, it became difficult specifying all required files. It was not possible to use the root directory as input to sconify-image either, and all attempts made failed.

4.1.1 Why it Failed and Potential Solution

Due to Intel SGX (iSGX) limitations, which provide security, it is not possible to apply SCONE to the entire image as both iSGX and SCONE are designed for application-oriented security and not system security. Thus, SCONE can protect the functionality of the SCU inside the container, but it cannot protect the entire system it uses. The reason is that the trusted computing base of iSGX is smaller than the OS [62]. This allows only a subset of OS system calls, and therefore is only a subset of system functionality. Because SCONE utilizes iSGX, they also conform to these limitations.

As the developed container mirrors an entire system, i.e. the physical SCU hardware, it is not suitable for SCONE. Limiting the system calls of systemd is not possible without disrupting the container's functionality, i.e. the SCU programs will not be able to perform system calls or communicate with other services on the platform. Therefore, to provide a working solution with SCONE, one needs to execute system calls outside of a SCONE container.

A possible way to realize this approach with external system calls would be to modulate the SCU container, separating it into multiple containers, and use a cluster as shown in figure 4.1. A sconified container in the figure represents a container running a single SCU program. The program then communicates to other containers, which together compose a SCU, via a network. In this solution, systemd is its own container, although as an intermediate step, one could include all non-SCONE related functionality inside a single container as figure 4.2 illustrates.

This approach could, however, bring additional work, as the containers have to send packages between each other. It is possible that Podman or Kubernetes[2] possesses an easy way to implement this. Otherwise, the SCU code needs to be extended. There was no attempt to modulate the container as this would require lots of implementation efforts, something there was no room for in the scope of this thesis.

[2]https://kubernetes.io/

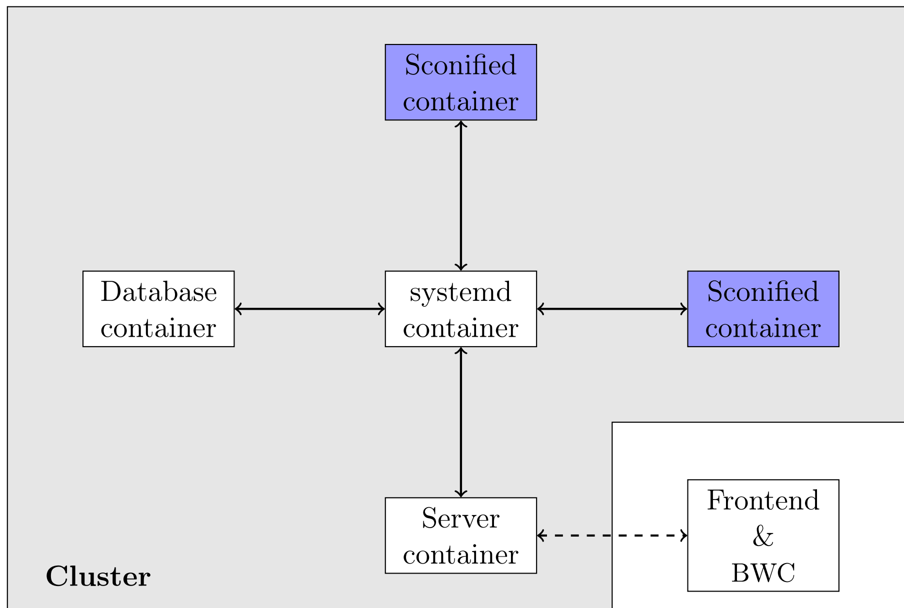


Figure 4.1: SCU cluster design consisting of multiple containers with one purpose. Some of the containers are sconified, e.g. the SCU programs. The entire cluster represents a SCU.

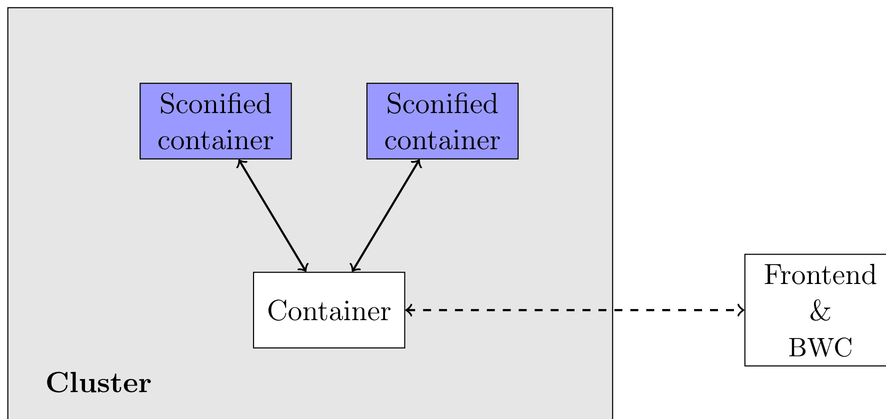


Figure 4.2: A cluster design where the SCONE components are extracted and running as its own container. The rest of the functionality, i.e. server, systemd, etc., is running inside a single container.

4.2 Generating an AppArmor Profile With Lic-Sec

At the start, it became clear that it was not possible to use Lic-Sec since the container image is not functional on docker. Docker is required as Lic-Sec runs

the container to audit it. Lic-Sec was modified to be compatible with Podman since the container could only be run with Podman. This was possible as the similarities between Docker and Podman are many.

To generate a profile, the tool had to be started after the container was instantiated. With the audit of Lic-Sec initiated, one had to use the container application as usual. As such, the wizard setup, adding camera and user, upload footage, as well as removing the camera, was performed, after which the tool seized its audit.

After the first training session, Lic-Sec generated a profile in which randomized paths were detected. In other words, the container created a non-static file system with directories containing names depending on certain variables. Currently, the Lic-Sec tool is unable to detect non-static file systems. As such, the tool had to be manually configured to treat the static root followed by the non-static part, as one.

As an example, if the container generated the file path */static/path/abc123* but on the next run it generated */static/path/xyz987*, then the tool was configured such that */static/path/* is considered the root and all sub-directories will conform to the same rules, i.e. *abc123* and *xyz987* will both have the same rule, e.g. read and writable. This was possible by applying regular expression (regex) on the directories with randomized names in the Lic-Sec configuration file.

It was also noticed that Lic-Sec was limited by the size of the logs it produces. These logs were used to later generate the profile. The max size was set too low, and therefore resulting in Lic-Sec missing information to generate a correct profile. The max size was therefore increased to collect sufficient logs.

The profile contained over 300 lines of rules that primarily consisted of defining read, write, execute, on file paths. In addition to the common access types, it also defined *prot_exec* on a few Linux libraries. Prot_exec specifies that a region of memory can store instructions that can be executed, as such defines what is allowed to run. The profile also defined which network type was used and on which domain, e.g. bluetooth stream, as well as the capabilities used, such as *chown*. Furthermore, it denied everything on the root directories *proc*, *mount*, and a handful of *sys* sub-directories, and are defaults of any AppArmor profile.

The AppArmor profile generated from Lic-Sec could be loaded and used with the container, although to do so, Podman had to be used with sudo, as Podman required root privileges to run AppArmor. It was not possible to run the container with the profile, likely due to it accessing things during start-up by systemd which was not covered by the audit. As a workaround, a default AppArmor profile was used and after the start, its content was replaced by the generated profile and then reloaded. The container could start successfully, however, the BWM was not accessible, rendering it useless.

The profile is missing definitions of the SCU programs, and likely other programs executed by systemd, which might explain why it is unable to execute the container with the generated profile. This could also be the reason for it being unable to run the BWM after reloading the profile from its default settings.

Concluding the trials, one see that Lic-Sec is better suited for containers that, on the one hand, runs on docker natively, and on the other hand is a microservice.

Chapter 5
Security Analysis

A few areas have been investigated to determine the security of the container. These are selected based on Axis' inquiry, as well as what has been deemed important in regards to the threat model. The container should be as isolated as possible, meaning it should not be possible to delete, read, or write, either from or to the container by the host or other containers. While the container has been enumerated, only a theoretical discussion is made with the effects the security enhancements would provide.

5.1 General Security Risks

It was interesting to deduce how much information one could obtain solely by inspecting the container image. It turns out that the container image possesses no confidentiality as its layers are unencrypted, enabling an adversary to easily enumerate the system. By inspecting the image, the adversary can gain knowledge of the SCU file structure, files, and permissions which could aid in finding a weakness to exploit.

Since this container requires the host's network interfaces, all packages to the container will go through its network stack. As such, these packages can be eavesdropped on by a passive adversary and an active adversary could potentially modify packages. Nevertheless, the communication between the BWM and the container is done via TLS, as such, the packages sent are encrypted. However, the container should ideally run with its own network namespace. This will allow the container to define its own firewall rules, minimizing the risk of port collision, and maintaining its own network stack.

5.2 Vulnerability Scan

To get an overview of the vulnerabilities in the image, and therefore potential exploits, a software tool called *Trivy*[1] was used to analyze the image. Trivy scans the image's packages and their version to compare it to the national institute of standards and technology's (NIST) national vulnerability database (NVD). NVD contains all known vulnerabilities and ranks them based on their severity (CVSS

[1]https://github.com/aquasecurity/trivy

score) and how one can mend the vulnerability if one exists. Although there are multiple tools for this purpose, Trivy was chosen for its ease of use.

The initial search found 697 vulnerabilities of different degrees of severity, as can be viewed in table 5.1, all without a fixed version. The two critical vulnerabilities are due to a package called *libgnutls30*, which implements the TLS and DTLS protocols. This package is mandatory for the server to run and the camera communication also requires TLS. There exist no known substitutions.

Table 5.1: Native image vulnerability scan of the container which resulted in 697 vulnerabilities. CVSS scoring is based on version 3.x.

Severity	Number of vulnerabilities
Unknown	1
Low	531
Medium	74
High	89
Critical	2

While the result from the scan is an abstract measurement of the system, it does provide some insight into the complexity and size of the image. The container can be viewed as an operating system since it is based on Debian Buster, and with it, introduces a large attack surface. It is a time-consuming task to dissect which packages are needed for the system to function, which is why it would be more ideal to find a better base image. This is however not possible to do for the SCU, as previously discussed.

5.3 Container Enumeration

By enumerating the container, i.e. information gathering, it is possible to deduce what actions one can perform to exploit and manipulate the system. In this section, the focus is on system layout, databases, keys, and certificates, as well as access control. Actions are performed without root access.

To gain an understanding of the content available on the container at the start, and after performing the setup wizard, one can inspect the container layer, after instantiating the container. The content produced by the container is unencrypted and can therefore be manipulated. Before the setup wizard, one can detect two different databases and one private and public key. The private key could be stolen, allowing anyone to impersonate its owner, and the accessible databases enable easy enumeration that can make it vulnerable to SQL injections.

After performing the setup wizard, additional keys, certificates, and configuration files were detected. While traversing the file system, one sub-directory was protected due to it being owned by another user. However, this is considered minor security as it could be bypassed with root access.

This file system enumeration could be extended to a script that monitors modifications done to it. To illustrate this, a python script was written using the library *watchdog*[2]. The script is running as a background process on the host machine and monitors all read and writes performed inside the container, i.e. to its container layer. The actions made by the container were logged by the script to a text file residing on the host machine. However, instead of a text file, the actions could be transmitted via the internet to an adversary.

The eavesdropping script was not able to monitor actions on the directory owned by another user, although it was possible to bypass it by running the script as root. If the host machine is running other containers as root, if an adversary manages to perform a container escape and run the script on the host machine, it would still be able to monitor all actions made on the SCU container.

Critical files such as private keys should be encrypted or isolated, in contrast to certificates and public keys. Some of the configuration files could be seen as critical if they dictate where footage is uploaded to. An adversary could change where footage could be uploaded by manipulating said files. Furthermore, the configuration files could also declare which user is assigned to which camera. This could lead to users becoming blocked or invalidate, or they could be framed by being assigned to a camera he or she did not wear.

As a result of enumeration, it was possible to perform an attack on the container. One could distort the video footage by interrupting the upload to the EMS followed by scrambling its bytes. When the connection was re-established, the footage was uploaded without error, however, the video could not be played.

5.4 With Security Enhancement

Next, a discussion on how the previously discussed security enhancements could potentially contribute to enhanced security of a containerized SCU service. Only the investigated methods are considered when discussing the increased security they contribute with. As it was not possible to fully execute any of the enhancements on the containerized SCU. This section, therefore, only reasons with regards to the applicability of the investigated methods.

Out of the two investigated methods, SCONE was thought to contribute the most in terms of security. Using SCONE would protect the container from host attacks as it enables binaries and files to be executed inside enclaves, or by encrypting said files instead. This could protect the SCU program's execution, as they would be located inside enclaves, and private keys and other sensitive data could be encrypted, thus providing confidentiality. This would not necessarily protect recordings from being deleted, however, it could prevent deceitful footage from being uploaded as it will not be encrypted, or encrypted with the wrong key.

An adversary needs to obtain the encryption (symmetric) key to decrypt data on the container. Obtaining said key would provide difficult as it resides inside the CAS. Another option is for the adversary to perform an exhaustive key search to obtain the encryption key. As SCONE uses a 258-bit symmetric key, the time to brute force the key is millions of years for non-quantum computers. However,

[2]https://pypi.org/project/watchdog/

suppose the key is retrieved via an exhaustive key search, then all SCU containers of a particular build will be exposed as they all will use the same key.

The SCU programs are still susceptible to attacks even if they would be executed inside enclaves. However, an attacker has to have extensive technical skills to be able to successfully perform, for instance, a side-channel attack. With that being said, the integrity of the program should stay intact, i.e. it should not be possible to modify the program for it to deviate from its purpose, but information from the program could be obtained.

Although SCONE is considered important, it does not protect the host from a running container or via an inter-container attack. Mandatory access control can provide the security necessary to protect in these cases. If an attacker manages to bypass SCONE, a MAC will most likely restrict (depending on the MAC and the profile) the access to files and directories an attacker tries to access. In other words, a solution where SCONE and MAC can be used could provide sufficient security to allow for public distribution of the container.

However, if the container is *not* distributed publicly, but only to trusted customers of Axis, then a MAC might provide sufficient security if it is set up properly. Although, if such a distribution model is used, then the customer is trusted not to manipulate the container in any way, and the MAC should prevent other containers from attacking the SCU container.

Since Podman requires root to run AppArmor, it might be more beneficial to use SELinux (even though SELinux is harder to configure properly) instead of it since it does not require root to use when running the container. However, a tool with similar nature to Lic-Sec could be used to generate the profile, although more research is needed to confirm if it is possible.

Chapter 6

Benchmarks

Performance is measured in two ways, upload time and camera assignment time. Upload time and camera assignment were measured both on the container and the SCU, and the results were then compared. To measure upload time, one video was recorded at a time. Recordings were done with different resolutions and video lengths to see if this affected the upload time for the two systems. The same video is used for each case between the container and the SCU for a fair comparison. Upload time is defined as the time between the docking of the camera to the content arriving at the EMS. The reason for this is that the SCU communicates and processes information from both the BWC and the EMS. The time is started when the BWC is docked and ends when the footage appears in the EMS. Results for upload time can be viewed in table 6.1. The computer running the container was equipped with an Intel i7-6700k 4.2 GHz CPU, 16 GB RAM, and a SanDisk 256 GB SSD.

Table 6.1: Upload time from BWC to EMS with a container and SCU.

Resolution	Video duration	Container	SCU
1080p	10 min	47.9 s	50.5 s
720p	10 min	35.1 s	37.7 s
1080p	27 min	57.1 s	45.4 s
720p	27 min	51.2 s	52.2 s
1080p	60 min	82.2 s	85.7 s
720p	60 min	58.2 s	57.4 s

The results indicate that no conclusion can be made that the container sufferers in terms of upload performance compared to the SCU. It is not surprising that the upload time increases with the size of the footage, however, theoretically the SCU should be faster in all cases. This could be explained by the extensive network communication propagated between the devices. It could be speculated that the upload time can be shorter for the container as it is located on the same device as the EMS. As such, there is no need to physically transmit the data over cables.

The performance of camera assignment was measured as the time between the addition of a camera to it becoming usable, and then through measuring the time between removal of a camera to the time of it being addable again. This was done to deduce if the container affected the time it takes to add or remove cameras to the system, in contrast to the SCU. The results can be seen in table 6.2 and shows no significant performance penalty.

Table 6.2: The duration of adding and removing a camera to the system.

Type	Container	SCU
Add	115.5 s	114.9 s
Remove	29.3 s	32.4 s

Chapter 7

Discussion

This section discusses the results obtained in the thesis regarding mitigating the Axis SCU to a container environment. It also identifies topics for future work.

7.1 The SCU Container

The success of containerizing the SCU comes with security consequences. In its current state, a public release is not recommended due to its vulnerabilities. Unencrypted image layers and system access render the recordings useless as evidence since it is easy to manipulate the content. Nevertheless, with further work, it could be made secure enough (see section 7.4) with minimal impact on performance.

The identified threats in section 1.5.1 are unfortunately not mitigated as the investigated security enhancements failed to be successfully implemented. The threat from the host is considerable without the use of the system's hardware to provide protection. However, even though the AppArmor profile failed, a simpler profile with default behavior could be applicable to protect from inter-container attacks. That being said, if it is possible to combine, for instance, SCONE and MAC, then the container have a good chance of being usable publicly in a secure fashion. Although, other combinations could be used, such as a virtual TPM and MAC.

The container could be used by Axis internally to simulate systems in their test suit. It could be beneficial in testing a larger system as it minimizes required office space and reduces manufacturing costs. A container also allows for more focus on developing the SCU software as hardware is no longer necessary. However, the container suffers due to modifications required for it to function. This could be avoided by developing a way to build the container image similarly to the SCU.

As the container lacks the Axis OS and instead contains a mock service, it is missing certain features that the SCU has. The two systems will function differently since the mock service provide static responses as well as provide double development. By developing an Axis OS base image for containers, the quality and reliability of their features will increase, and the development time for future Axis containers will decrease.

From a customer's point of view, using a container is easy as everything is packaged for it to work. The end-user only has to run it and perform the setup wizard. It is also easier to distribute updated versions of the SCU container as the

latest will always be available. In a virtual machine, the user will have to install all software and perform updates manually.

7.2 SCU as a Virtual Machine

Currently, a virtual machine solution will provide more security, compared to a container, as it is better isolated. A VM is also better suited for a system, which the SCU is. However, a different threat model is needed compared to the one in section 1.5.1. The threat from other VMs is negligible as a user seldom uses multiple VMs in parallel or runs them continuously, however, the host threat remains. To gain protection from the host, one could use Intel SGX or a vTMP. The choice of virtualization software can impact security. Although, the number of vulnerabilities (table 2.2) for a specific VM will vary with time as new vulnerabilities are discovered.

In some instances, it might be beneficial to use a VM over a container. A VM's performance can be compared to that of a container, considering the minimal work the SCU appears to do. Furthermore, by using a VM it could be possible to match the functionality and security of a SCU. However, an Axis OS image should be available for the VM in order to more closely match the SCU, much like with the container solution.

As the solution is intended for public use, a bare-metal solution is not applicable as the customer will most likely neither possess the knowledge nor the setup for it. Without bare-metal, the performance and the security of the VM will suffer. As the performance result show (6.1 and 6.2), the SCU and the container generate similar results. This suggests that the camera is the main factor affecting upload time and camera assignment performance, rather than the SCU and the container. Therefore, the performance should be of no concern regardless of VM technology.

7.3 Conclusion

This thesis shows that it is possible to implement a hardware system as a container, although it comes with limitations in terms of features, flexibility, and security. The developed container system does not meet sufficient security levels to be deployed publicly.

If such a solution is required in near future, then an investigation on whether or not it is possible to do it on a VM is recommended. The VM will provide better support for a system, compared to a container, as it provides better isolation and security out of the box. Nevertheless, security enhancements should still be implemented, despite using a virtual machine, to protect the VM from the host.

If a container solution is desired, then using a cluster should be considered to enable a more secure container. The cluster could provide the solution to use a container publicly if it enables SCONE to be used.

Finally, if Axis decides to provide the container to trusted customers, then the container could be secure enough with the use of a properly configured MAC, although more research is needed to confirm this.

7.4 Future Work

Containers are intended to work as micro-services. Some areas need to be explored further to optimize the functionality and security of the container solution. One could attempt to divide the SCU container image into modules to explore whether it can maintain functionality in a modular form and if SCONE becomes applicable. The application of SCONE would greatly improve the security of the container. In the case of this thesis, a first step could be to extract the SCU programs into separate containers (see figure 4.2) and derive a working solution from there. The next step could be to apply SCONE.

A possible substitution to SCONE could be AMD secure encrypted virtualization[1]. It can be used to provide isolation from the host and protect container data, similar to Intel SGX. It is applicable to both virtual machines and containers.

[1]https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

References

[1] "Changing the face of surveillance: The brains behind the first network camera," https://www.axis.com/newsroom/article/first-network-camera, [Online; accessed 16-02-2021].

[2] S. N. T.-c. Chiueh and S. Brook, "A survey on virtualization technologies," *Rpe Report*, vol. 142, 2005.

[3] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21 – 27, 01 May. 2018. [Online]. Available: https://content.sciendo.com/view/journals/acss/23/1/article-p21.xml

[4] T. Y. Win, F. P. Tso, Q. Mair, and H. Tianfield, "Protect: Container process isolation using system call interception," in *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks 2017 11th International Conference on Frontier of Computer Science and Technology 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*, 2017, pp. 191–196.

[5] "About the open container initiative," https://opencontainers.org/about/overview/, [Online; accessed 16-02-2021].

[6] "About storage drivers," https://docs.docker.com/storage/storagedriver/, [Online; accessed 05-05-21].

[7] "Docker architecture," https://docs.docker.com/get-started/overview/, [Online; accessed 11-02-2021].

[8] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[9] "Keep containers alive during daemon downtime," https://docs.docker.com/config/containers/live-restore/, [Online; accessed 11-02-2021].

[10] A. Holbreich, "Docker components explained," http://alexander.holbreich.org/docker-components-explained/, 2018, [Online; accessed 10-03-2021].

[11] M. Crosby, "Use of containerd-shim in docker-architecture," https://groups.google.com/g/docker-dev/c/zaZFlvIx1_k?pli=1, 2016, [Online; accessed 10-03-2021].

[12] "Manage data in docker," https://docs.docker.com/storage/, [Online; accessed 05-05-21].

[13] A. Sheka, A. Bersenev, and V. Samun, "The problem of reprodicible results on the hpc cluster," in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 0833–0837.

[14] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, "On the use of containers in high performance computing environments," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020, pp. 284–293.

[15] "conmon," https://github.com/containers/conmon, [Online; accessed 18-02-2021].

[16] B. Baude, "Podman: Managing pods and containers in a local container runtime," https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods/, 2019, [Online; accessed 18-02-2021].

[17] "Performance," https://github.com/containers/crun, [Online; accessed 17-02-2021].

[18] H. Gantikow, S. Walter, and C. Reich, "Rootless containers with podman for hpc," in *International Conference on High Performance Computing*. Springer, 2020, pp. 343–354.

[19] G. Scrivano, "User namespaces support in podman," https://www.projectatomic.io/blog/2018/05/podman-userns/, 2018, [Online; accessed 18-02-2021].

[20] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia Computer Science*, vol. 175, pp. 157–164, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705092031704X

[21] R. Chandramouli and R. Chandramouli, *Security assurance requirements for linux application container deployments*. US Department of Commerce, National Institute of Standards and Technology, 2017.

[22] S. Kenlon, "Demystifying namespaces and containers in linux," https://opensource.com/article/19/10/namespaces-and-containers-linux, 2019, [Online; accessed 05-03-2021].

[23] "namespaces(7) — linux manual page," https://man7.org/linux/man-pages/man7/namespaces.7.html, [Online; accessed 04-03-2021].

[24] "Control group v2," https://www.kernel.org/doc/Documentation/cgroup-v2.txt, [Online; accessed 12-04-2021].

[25] "cgroups," https://wiki.archlinux.org/index.php/cgroups, [Online; accessed 12-04-2021].

[26] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[27] B. S. Radhika, N. V. N. Kumar, R. K. Shyamasundar, and P. Vyas, "Consistency analysis and flow secure enforcement of selinux policies," *Computers & Security*, vol. 94, p. 101816, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820300948

[28] A. Eaman, "Tepla: A certified type enforcement access-control policy language," Ph.D. dissertation, Université d'Ottawa/University of Ottawa, 2019.

[29] J. Ko, S. Lee, and C. Lee, "Real-time mandatory access control on selinux for internet of things," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, 2019, pp. 1–6.

[30] V. Kuliamin, A. Khoroshilov, and D. Medveded, "Formal modeling of multi-level security and integrity control implemented with selinux," in *2019 Actual Problems of Systems and Software Engineering (APSSE)*, 2019, pp. 131–136.

[31] "What is selinux?" https://www.redhat.com/en/topics/linux/what-is-selinux, [Online; accessed 11-03-2021].

[32] Z. C. Schreuders, T. McGill, and C. Payne, "Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, 2011. [Online]. Available: https://doi.org/10.1145/2019599.2019604

[33] H. Zhu and C. Gehrmann, "Lic-sec: An enhanced apparmor docker security profile generator," *Journal of Information Security and Applications*, vol. 61, p. 102924, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212621001435

[34] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-sec: A fully automated container security enhancement mechanism," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1561–1564.

[35] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 559–567.

[36] T. Morris, *Trusted Platform Module.* Boston, MA: Springer US, 2011, pp. 1332–1335. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_796

[37] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "Vtpm: Virtualizing the trusted platform module," ser. USENIX-SS'06. USA: USENIX Association, 2006.

[38] S. Hosseinzadeh, S. Laurén, and V. Leppänen, "Security in container-based virtualization through vtpm," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, 2016, pp. 214–219.

[39] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers,

R. Kapitza, P. Pietzuch, and C. Fetzer, “SCONE: Secure linux containers with intel SGX,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[40] V. Costan and S. Devadas, “Intel sgx explained.” *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[41] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[42] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, “Malware guard extension: Using sgx to conceal cache attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[43] “Sconify container image (standard edition),” https://sconedocs.github.io/ee_sconify_image/, [Online; accessed 03-06-21].

[44] Qian Chen, R. Mehrotra, A. Dubeyy, S. Abdelwahed, and K. Rowland, “On state of the art in virtual machine security,” in *2012 Proceedings of IEEE Southeastcon*, 2012, pp. 1–6.

[45] “Hypervisor,” https://www.vmware.com/topics/glossary/content/hypervisor, [Online; accessed 26-03-2021].

[46] Z. Wang, C. Wu, M. Grace, and X. Jiang, “Isolating commodity hosted hypervisors with hyperlock,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 127–140. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/2168836.2168850

[47] D. Beserra, F. Oliveira, J. Araujo, F. Fernandes, A. Araújo, P. Endo, P. Maciel, and E. D. Moreno, “Performance evaluation of hypervisors for hpc applications,” in *2015 IEEE International Conference on Systems, Man, and Cybernetics*, 2015, pp. 846–851.

[48] S. R. Poojara, V. B. Ghule, M. N. Birje, and N. V. Dharwadkar, “Performance analysis of linux container and hypervisor for application deployment on clouds,” in *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*, 2018, pp. 24–29.

[49] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, “Performance overhead comparison between hypervisor and container based virtualization,” in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, pp. 955–962.

[50] S.-A. Federico, B. Rodrigo, and L. Ben, “Security issues and challenges for virtualization technologies.” *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1 – 37, 2020. [Online]. Available: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edscma&AN=edscma.3382190&site=eds-live&scope=site

[51] K. Kortchinsky, "Cloudburst: A vmware guest to host escape story," https://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf, 2009, [Online; access 22-04-21].

[52] S. Daniele and L. Emil, "Evolution of attacks, threat models, and solutions for virtualized systems." *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1 – 38, 2016.

[53] J. Wu, Z. Lei, S. Chen, and W. Shen, "An access control model for preventing virtual machine escape attack," *Future Internet*, vol. 9, no. 2, 2017. [Online]. Available: https://www.mdpi.com/1999-5903/9/2/20

[54] G. Ouffoué, A. M. Ortiz, A. R. Cavalli, W. Mallouli, J. Domingo-Ferrer, D. Sánchez, and F. Zaidi, "Intrusion detection and attack tolerance for cloud environments: The clarus approach," in *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2016, pp. 61–66.

[55] A. Peppas, G. G. Xie, and C. Prince, "ferify: A virtual machine file protection system against zero-day attacks," 2020.

[56] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2012, p. 305–316. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/2382196.2382230

[57] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/1972551.1972552

[58] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[59] T. Kim and Y. Shin, "Poster: Mitigating memory sharing-based side-channel attack by embedding random values in binary for cloud environment," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, p. 919–921. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/3320269.3405444

[60] Y. Zhang and Y. Makris, "Hardware-based detection of spectre attacks: A machine learning approach," in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.

[61] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," 2019.

[62] R. C. R. Condé, C. A. Maziero, and N. C. Will, "Using intel sgx to protect authentication credentials in an untrusted operating system," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 158–00 163.