Application security for embedded systems

Mikael Bäckman
dic12mba@student.lu.se
Fredrik Hagfjäll
dic12fha@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Martin Hell, Fredrik Larsson

Examiner: Thomas Johansson

April 19, 2017

© 2017
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

With the rise of Internet of Things (IoT) accessories such as network attached cameras, light bulbs and thermostats are all constantly connected to the Internet and security concerns must be taken seriously. If a bug exists in an application it could be hacked by a malicious adversary that then could harm the underlying system, leak information or attack other devices or networks.

Applications should not be allowed to damage the underlying system and there exists many isolation techniques for the general purpose computer, but these solutions are not designed for the embedded world and needs to be evaluated. This thesis compares isolation techniques in Linux for a specific embedded system and benchmarks performance and security. The thesis concludes that there are many non working isolation techniques for embedded systems and that further work is needed to enable them. However the best current solutions in this dissertation is Bubblewrap, Firejail and TOMOYO.

Foreword

We would like to thank Axis Communications AB for allowing us to pursue and complete this thesis. Thanks are in order to the teams of Linux Firmware Platform: Event and Linux Firmware Platform: Network and Security for putting up with all of our very time consuming and specific questions. Special thanks are set out to Niklas Hjern, Mattias Hansson and our supervisor Fredrik Larsson.

Secondly we would like to thank LTH and our advisor Martin Hell for being there to help us plan and structure our work.

Popular Science Summary

In all operating systems applications should be executed in a fashion that has no possibility of harming the underlying operating system or other applications. This kind of protection already exists in many flavours for ordinary systems, but for the use in embedded systems the already running protections have to be tested and evaluated regarding disc space usage, CPU usage and other limited resources. The solution proposed in this dissertation for achieving operating system protection is isolation of applications in a Linux environment.

The isolation part in this thesis is aimed at embedded systems and after elaborating the basics regarding how applications can be isolated in a Linux environment and what general threats exist for an isolation technique the actual techniques are presented. There exists various tools in the Linux kernel that helps to do this isolation, for example seccomp, capabilities and namespaces. However these cannot provide enough isolation by themselves and should only be seen as parts of the operating system that will be used by more elaborate tools to achieve isolation.

Linux Security Modules is one of the more evolved candidates that can achieve this type of isolation and with it being built into the kernel the possible benefits are large. Other solutions presented include containers and other tools that can be seen as sandboxes. In total 11 different implementations were chosen to be described further and applied on the embedded system. However not all of them turned out to be suited for our embedded system and were not able to be run due to various reasons.

One crucial part for the embedded system used in this thesis was the extra size required by the implementation and two had to be discarded straight away since they required too much disc space to function. Other problems included unsupported architectures, lack of user space tools and support for transfer an entire Linux file system to the embedded system. The thesis presents implementations that are running on the system and recommendations on which one to use based on security evaluation and performance, but it also provides some valuable insight as to where the focus and continued work should be regarding implementations that did not run.

One conclusions of the thesis is that even though some isolation techniques that is being used in embedded systems today it might not work on the embedded system within the scope of this thesis. The reason for this is that all embedded

systems are so fundamentally different with architectures, available memory and use cases. The final recommendations of implementations that should be used are TOMOYO and Bubblewrap. TOMOYO is recommended since it almost does not add extra overhead and the user space tools is easy to use and activate. Bubblewrap is recommended since it isolates the applications very easily and runs unprivileged which means that if an isolation breakout is achieved the resulting damage would be limited on the system.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Scope . . . 2
- 1.3 Layout . . . 2

2 Preliminaries **5**
- 2.1 Access control . . . 5
- 2.2 Virtualization . . . 6
- 2.3 Sandboxing . . . 7
- 2.4 Containers . . . 7
- 2.5 Linux Kernel built in tools . . . 8
- 2.6 Possible vulnerabilities and weaknesses in isolation techniques . . . 13
- 2.7 Licenses . . . 15
- 2.8 Benchmarking tools . . . 16
- 2.9 Related work . . . 17

3 Description of the system **19**
- 3.1 Hardware . . . 19
- 3.2 Software . . . 19

4 Methodology **21**
- 4.1 Use case . . . 23

5 Isolation techniques **25**
- 5.1 Linux Security Modules . . . 25
- 5.2 Sandboxes . . . 28

6 Results **37**
- 6.1 Non running isolation techniques . . . 37
- 6.2 Techniques with theoretical possibility of execution . . . 38
- 6.3 Running isolation techniques . . . 39
- 6.4 Security evaluation . . . 40
- 6.5 Licenses . . . 47

7 Discussion **49**
7.1 Results 49
7.2 Fault sources, possible solutions 50

8 Conclusions **53**
8.1 Recommendation 53
8.2 Future work 53

References **55**

A Kernel flags **61**
A.1 Firejail, Bubblewrap 61
A.2 TOMOYO 61
A.3 AppArmor 62
A.4 SMACK 62
A.5 SELinux 63
A.6 LXC 64
A.7 nspawn 64

List of Figures

2.1 Flow chart of access control 5
2.2 Traditional architecture versus server virtualization 7
2.3 Architechtural overview of the OCI-developed tool stack. 9
2.4 LSM Hook architecture 10
2.5 Concept of linux namespaces 12

6.1 Numeric sort performance 40
6.2 String sort performance 41
6.3 Bitfield "bit twiddling" performance 41
6.4 Emulated floating-point performance 42
6.5 Fourier coefficients performance 42
6.6 Assignment algorithm performance 43
6.7 Huffman compression performance 43
6.8 IDEA encryption performance 44
6.9 LU Decomposition performance 44
6.10 I/O performance 45
6.11 Image size comparison 45

List of Tables

5.1 LSM comparison 26
5.2 Sandbox comparison 29
5.3 Continued sandbox comparison 29

Chapter 1
Introduction

With the rise of the Internet of Things (IoT), accessories such as network attached cameras, light bulbs and thermostats have become increasingly popular. Many of these IoT devices have the possibility of running applications and programs that are developed and maintained by both the vendors and third party developers. This could raise concerns about the security and robustness of the underlying operating system and other installed application in the system. These applications can often be dependent on other applications and system features which helps to increase the ever growing concern of what will happen to the system if the application is malicious. The applications do not even have to be intentionally malicious but instead have bugs or vulnerabilities that an adversary could use as an advantage to damage the underlying system.

1.1 Motivation

The number of IoT devices are increasing for every year, and will according to estimates from IHS, grow by 57.8 million devices until the year 2025 [1]. With this massive projected growth of the IoT-markets in general, and the possible gains for a company by continuing to develop these devices there should be a way of protecting the underlying operating system from harm when running applications. Software bugs will continue to exist and in order to limit the damage some sort of isolation of the applications should be enabled on the device. This dissertation is going to make a study regarding the most suitable solution for isolating applications on embedded systems that have limited resources. The study will test a set of already existing isolation techniques of different natures and finally reach a conclusion of what techniques that fit embedded systems better.

1.1.1 Security threats

As with all complex systems and attack preventions there is not a single threat to take into consideration and not one single solution. If an application is executed with an unique user id and group id on a Linux system the application will have limited access to parts of the system, i.e. permissions for different files. This solution is not always suitable and does not limit which kernel calls can be made, and does not generally restrict access on what executables that can be executed

on the system. If a binary is malicious or becomes compromised it can damage the system and other applications, especially if the permissions have not been restricted properly. In general, read permission of files and other system resources are not restricted. By isolating a whole user or group and the application together with some of the techniques described in this dissertation the necessary amount of security for the system and other applications can be achieved.

1.2 Scope

Isolation can be made on various operating systems but since Linux is a widely used operating system in embedded systems the dissertation will focus on a specific Linux distribution. The actual Linux distribution and hardware is developed and maintained by a company and presented in Chapter 3 and 4.

Since embedded systems often are resource constrained and the use case, that will be presented in Chapter 4, is to only isolate applications, this thesis will not focus on full virtualization (explained in Section 2.2) but instead focus on just isolating the applications on the host OS. This isolation technique will have a smaller performance impact on the system as opposed to full virtualization. When looking at the various isolation techniques, there exists a very wide variety of different techniques that has the possibility of being supported by the system running on to the cameras. This dissertation is time limited and the actual phase for implementing and enabling the techniques is limited as well, which means that not all of the techniques that are going to be covered will have the possibility to be specifically rewritten to best suite the test environment. Most of the techniques are going to be run as generically as possible, which also means that changing the source code of the isolation technique could solve possible forthcoming problems but is outside the scope for this dissertation.

1.3 Layout

The organization of the report is:

- *Chapter 1 Introduction:* Introduction to the specified problem with the main focus of providing an insight into why there is a need for running applications in a secure fashion.

- *Chapter 2 Preliminaries:* Explains the theory needed to understand how the isolation can be achieved. It covers a variety of techniques such as virtualization, sandboxing and system call filtering that will work as the base for the comparison.

- *Chapter 3 Description of the system:* Gives a system description that is used as a base for the use case in the methodology.

- *Chapter 4 Methodology:* Explains the methodology that is being used to reach the final conclusion and recommendations.

- *Chapter 5 Isolation techniques:* Compares and elaborates the techniques that has been chosen and provides an insight in what the comparison is going to focus on.
- *Chapter 6 Results:* Covers the results of the implementation and has an security evaluation of the working isolation techniques.
- *Chapter 7 Discussion:* Contains a discussion about the results, both in the project aims and the general level. Some possible error sources are discussed as well.
- *Chapter 8 Conclusions:* Concludes the dissertation and opinions as to where continued work should be focused.

Chapter 2

Preliminaries

Isolation can be done in a variety of ways and the various concepts as well as relevant background information on these concepts are covered in this chapter. The chapter will also cover needed information on Linux kernel features that various isolation techniques are using together with licensing information.

2.1 Access control

Access control in operating systems is needed since the operating system should provide confidentiality and integrity, i.e., a user should be able to deny other users read access to his files and in the same way protect these files from modification and deletion by other users. The actual active user/process is called the subject and the passive part being the file or resource is called the object. The subject is going to request an access right (read, write or such) that in turn is going to be checked by the reference monitor that is going to grant or deny access on this object based on the access rights as shown in Figure 2.1. The access rights can basically be in two directions, either to tell what a subject is allowed to do or what may be done to an object.

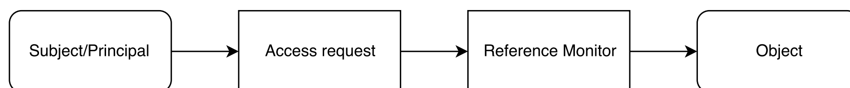

Figure 2.1: Flow chart of access control

On an elementary level there has to be only two access operations on an object. To observe and look at the contents of an object or to alter the contents of said object. For most cases however this is too general to be applied on large scale and some other access control model has to be implemented. If a subject can control an access control mechanism so that it allows or denies object access then it is called discretionary access control (DAC). If the system decides the access rights it is called mandatory access control (MAC). Even if they are very different they can be used together, and if both MAC and DAC is used both mechanisms must grant a subject access to an object.

One of the most famous models of enforcing access control is the state machine described in the Bell-LaPadula Model [2]. This access control scheme is used for

enforcing the access control in military and governmental usages. The scheme describes access control rules that use labels on the objects within the system and specific clearances for the subjects.

Another, security model is the Access Control Matrix (ACM) that helps characterize each subject's access right on each object in the system. In practice this is not a feasible option since the matrix size will be very large, there will be much redundancy and the management will require extensive work. The ACM can be separated so that each column is by itself, and is then called an Access Control List (ACL). This list will then contain which subject has what access rights for the specified object. It however gets difficult to get an overview of the individual subjects permissions if this is used. The other way around is also possible, i.e., separate each row so that the subjects contains a list called capabilities that states the access rights on objects. This of course brings the opposite problem that it is difficult to see who has access to the objects.

There are two principles that should be used in the aspects of access control. The principle of least privileged, meaning that a subject only should have access to the necessary objects and the other is separation of duties where functionality critical to security must be done by more than one user [3].

2.2 Virtualization

In computer science the term virtualization means that something is creating a virtual version of something. This virtual version could be for example storage devices, operating systems, hardware platforms, network resources and so on. The virtualization goal is to abstract away the underlying hardware and software from the the applications, data or operating system running on top of the virtualization software. One key technique in computer science is server virtualization where a layer called hypervisor is used to emulate the underlying hardware, which is shown in Figure 2.2.

The operating system is, when the hypervisor is used, the hypervisor itself and it handles the communication between the guest OSes and the hardware by simulating the hardware to the guest OS. The performance is not equal to the execution on true hardware but many guest operating systems does not need the full capacity of the underlying hardware but instead can make use of the greater flexibility, isolation and control that is gained [4].

Application virtualization is when the application layer is abstracted away from the operating system and allows applications to execute on a system, but with some virtualization activated. This could be achieved by providing a virtual file system with unique settings/registries and thus no modifications on the underlying system can be made. Heavy operating system integrated applications such as modifying the look of the GUI or user's settings will not be possible to put in a application virtualization environment since no modifications will be made to the real system [5].

There exists many other forms of virtualization, but in the context of running applications isolated in an embedded system these two are the most relevant ones. For example network virtualization is when the bandwidth is split into channels

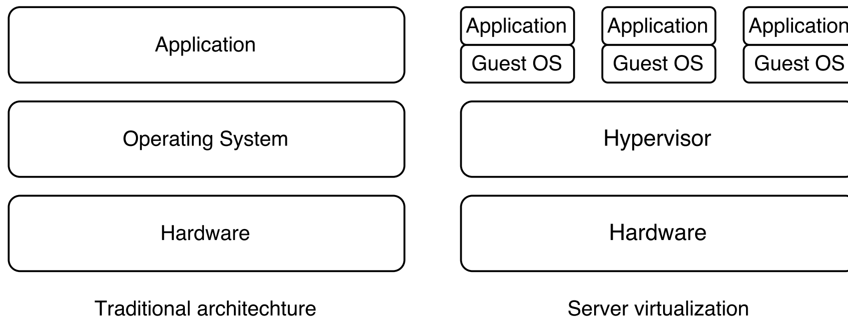

Figure 2.2: Traditional architecture versus server virtualization

that can be used separately and storage virtualization is when physical storage from many sources is pooled together to be seen as only one storage location that is managed from one place.

2.3 Sandboxing

In the context of computer security a sandbox is a mechanism to separate running programs with the aspect of keeping the operating system and other applications isolated and safe when running programs. The sandbox does so by isolating or virtualizing an environment where the program can be executed in, limiting the possible access to the underlying operating system and other applications. Monitoring in a sandbox can occur as well as trying to check whether or not the program executing is malicious or not [6].

Exactly how a sandbox works differs but there are some main focuses. One solution is to let the sandbox act as a standalone program that basically allows other executable code to be run inside of this program and thus the program itself restricts access to the global system resources. One example of this is Chromium [7]. Another solution is to restrict every executable on the system by letting the OS itself sandbox every process from the system and each others [8]. The sandbox can be in use for just a short period of time, scanning the unknown content and when it has finished the scan the content is moved to the trusted part of the system, but it can also run for the entire time of the execution and limit the use of certain functions and continuously scan the operations.

2.4 Containers

Where a sandbox is a way to check and isolate programs from doing potentially malicious operations in a confined area, a container is a virtual place for the application to exist which runs for the entire time of execution. This definition means that a container is a subset of sandbox. Basically a container consists of the entire runtime environment (the application including the dependencies, other needed binaries, libraries and configuration files) packed together. When making a container

the underlying operating system and hardware is abstracted away. Containers can sometimes be mixed with virtualization, and while this is true to some aspects the main difference is that with virtualization an entire virtual machine including hardware and operating system is abstracted away while the containerized version share kernel with the other containers and the host OS. These shared parts of the operating system are read only and a new file system is mounted within these for writing. One major advantage of containers to virtualization is that they are more lightweight than fully virtualized systems. Containers and application virtualization can be seen as the same within this dissertation (definitions differ slightly from various sources and in some cases they are not the same).

For security aspects the general consensus is that virtualization and fully virtualized machines are more secure compared to containers. The reason for this being that if a kernel vulnerability is found, it could be used to access the containers or to break the isolation provided by the container. This could also be stated for the hypervisors, but since hypervisors provide less functionality than the kernel it means that the possible attacks are fewer (or at least in theory) [9].

2.4.1 Open Container Initiative

This initiative's aim is to specify the configuration, execution environment and lifecycle of a container and by focusing all major companies to one standard the hope is that balkanization will be avoided. Some of the companies behind the initiative are Google, Docker and CoreOS (developers for rkt, see Section 5.2.6). This dissertation will only describe the most relative parts of the specification that is used by both Docker (see Section 5.2.4) and rkt. Open Container Initiative (OCI) has also created tools to help follow and implement the specification.

`runC` is one of these tools for spawning and running containers. It makes use of libcontainer as the underlying mechanism, as shown in Figure 2.3. Libcontainer is the actual containerization implementation and it uses the isolation techniques on Linux to achieve the isolation, which will be described later in this dissertation. Some of these techniques are capabilities (see Section 2.5.2), namespaces (see Section 2.5.5), file system access control and system resource limitations. Extra security can be applied by either SELinux (see Section 5.1.3) or AppArmor (see Section 5.1.1) policies [10] which will be described later on in this thesis as well.

The container's format defines how the container should be packed and bundled to include all necessary data and corresponding metadata needed to startup and run the container. Besides the container's OS file system a file named `config.json` must be included that defines environments, capabilities, namespaces, rlimits and mounts so that the container software can be setup and executed as intended. This can enable shared resources with both the host OS as other containers.

2.5 Linux Kernel built in tools

The Linux kernel has some built in tools for isolation of processes but each of them with limited functionality. These tools are stand-alone and have no dependencies outside of the kernel. Many of the isolation techniques that will be discussed later will include or be based on these tools, but most of them are not assessed

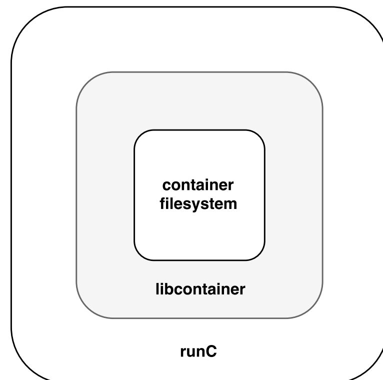

Figure 2.3: Architechtural overview of the OCI-developed tool stack.

individually since they do not provide the desired level of isolation on their own but are instead covered in these preliminaries to give the reader an underlying understanding of the more high-level techniques.

2.5.1 Linux Security Modules

Linux Security Modules (LSM) is a design principle in the kernel that supports loading of modules for security purposes. The goals with support of LSMs were to allow a truly generic security module that is conceptually simple, minimally invasive, efficient and supports existing capabilities logic (POSIX.1e). The LSM places hooks in the kernel before the access to the object is granted, and is enforced by policies created by these modules, as shown in Figure 2.4. The system calls first go through the existing logic for allocating and finding resources, and has the error checks as in the usual DAC access controls but just before the kernel would grant access to the object the LSM hook is enabled and makes an call to the LSM and checks whether access should be granted or not [11].

This design results in that the LSM access control decisions are restrictive. The module can only really deny access to objects, and all the usual errors and security checks results in that access can be denied before it reaches the LSM. MAC systems are usually implemented the other way around and for Linux it limits the flexibility of LSMs but limits the performance penalty on the kernel [12].

2.5.2 Capabilities

The original UNIX privilege mechanism did suffice for some time, but it had one significant flaw; programs that require some privilege that is more than a normal user must in fact run will full privileges. This is for example when a process has to open network sockets with port below 1025 that will require root access. To allow the process to opens ports below 1025 there can basically be two solutions, one is to give the process root access and thus full access to the system (it can open

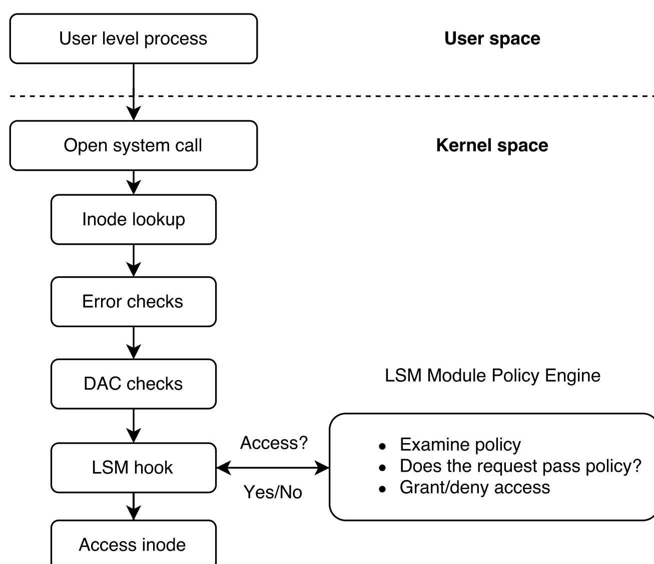

Figure 2.4: The LSM hook architecture with example of reading from a file (inode).

up to system abuse as backdoors, circumventing access control and data changes) and the other is to use what is called capabilities [13].

Capabilities are flags that inform the kernel what the specified process is allowed to do and does so system wide regardless of which user id that is currently executing the process or what object is being accessed by the process. When doing this the usage of root with full access can be avoided and instead different capabilities can be set, one can see it as capabilities trumps root privileges. This technique divides the root privilege into a set of capabilities that break the superusers privileges, for example the ability to switch between UID's that are configured by `CAP_SETUID` and the ownership of objects that is changed by `CAP_CHOWN` [13].

2.5.3 Berkeley packet filter

The classic Berkeley packet filter is a method that was used to filter network packets and did so by letting the kernel analyze the packets before preceding them for the sake of ensuring no harm was done to the running system. Today there exists an extended (eBPF) version where all system calls, file descriptors and so on can be filtered. The filtering should be done by white-listing allowed system calls with a map and then loading the eBPF program into the kernel which will run a statical analysis of the program to make sure no other calls are made than the white-listed ones [14]. By white-listing instead of black-listing future implemented syscalls will be blocked and blacklist bypass will be mitigated [15].

The classic BPF for network packet filtering is used in many applications such as Wireshark, tcpdump/libpcap, DHCP and nmap. The extended version is integrated in seccomp (see Section 2.5.4) and is used by Chrome, Firefox and

OpenSSH to name some of them [16].

BPF kernel internals have been developed and implemented to mimic the underlying architecture native instruction sets thus increasing the speed. The instructions are compiled to bytecode and executed by the kernel. The design is to be just in time compiled with one to one mapping and generate optimized code that performs almost as fast as code compiled natively. This optimization is made with the x86_64 architecture in focus. The 10 registers in BPF are mapped one to one with the 64-bit architecture [17]. An application (any binary) to be filtered with BPF will ask the kernel to execute the BPF program.

The reason BPF can be seen as secure and controlled is due to the filtration and execution of the BPF program. The BPF program is kept in check and reduced in complexity by the limitations of the bytecode. Some examples on how it is limited are [16]:

- All the jumps made by the program are forward only so that there can not be any loops in the BPF program and the program is also bound to terminate
- All the instructions are within range and valid
- A single BPF program has a limited numbers of instructions (maximum 4096)

These limitations makes sure the BPF programs within the kernel will run fast and not get stuck in loops.

2.5.4 Seccomp

Seccomp, short for Secure Computing, is a tool for sandboxing, integrated in the Linux kernel. It has been in the kernel since version 2.6.12 [18]. When seccomp is enabled the process that is using it enters a "secure computing mode" where just four basic system calls are available; `read()`, `write()`, `exit()`, and `sigreturn()`. This first version was very limited and getting code to run with only these four calls was difficult.

seccomp-bpf

The second version of seccomp was released in the kernel version 3.5 [19] and added a second mode for seccomp; `SECCOMP_MODE_FILTER`. With this command the process can specify which system calls should be available and by using BPF programs (based on eBPF) the process can restrict system calls entirely or for certain arguments. When using these seccomp-bpf programs all the positive functionality on the kernel will follow from eBPF (as stated in Section 2.5.3). BPF data is inserted into seccomp and has some different fields that tells us what system call is made. It has the system call number (which differs from architecture to architecture), what type of architecture is being used, the instruction pointer and system call arguments. It is received as a read-only buffer that the current process can use but not change [20].

2.5.5 Linux namespaces

Historically the Linux kernel only had one process tree, but with namespaces Linux got the possibility to isolate a global system resource to a namespace and make the processes within that namespace believe that they have their own instance of that global resource, i.e., nested process trees. This technique was implemented in kernel version 2.6.24 [21]. The reason for nested process trees is that if you run multiple services it is essential to the security concerns that these services are isolated from each other. If namespaces are not used an adversary might be able to use one compromised service to attack other services, but if the services are isolated the effects of these kinds of attacks are minimized. Containers for Linux often use this technique [22]. To make and manage namespaces in Linux there exists three system calls; `clone(2)`, `setns(2)` and `unshare(2)`. The `clone(2)` system call is made to create a new process and creates namespaces for that process and the child processes depending on what kind of input is given to the call. `setns(2)` is a system call for allowing processes to join already existing namespaces with a file descriptor as input. Finally the `unshare(2)` system call takes the process that is calling it and moves it to a new namespace. An illustration on how a namespace can be seen graphically is shown in Figure 2.5.

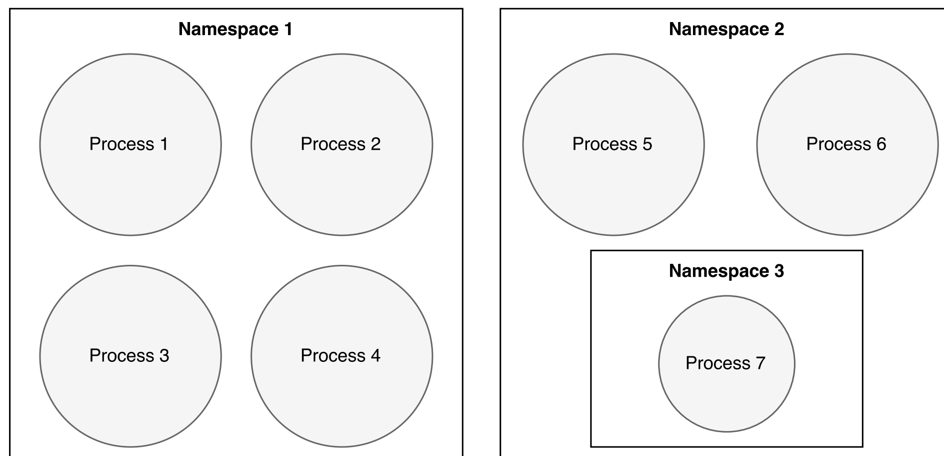

Figure 2.5: Linux namespaces example; process 1-4 can communicate with each other but not with any other processes running in the system. Process 7 will be completely isolated to any other processes but process 5-6 will see process 7.

This kind of isolation can be made on seven different aspects in the Linux kernel [23]:

- **Cgroup** sets the root directory of the namespace to an arbitrary path in the system. For example if a namespace has its cgroup set to `/home/user/applications/app1` processes in that namespace will have their `/` (root) linked to `/home/user/applications/app1`.
- **Interprocess communication** isolates System V IPC objects and POSIX

message queues. Objects created in a namespace are visible to all other processes within that namespace but not to other namespaces. All objects are destroyed when the namespace is destroyed.

- **Network** isolates and separates different networks from each other and one namespace will, to each process, present a unique network interface. The loopback interface is also different for each namespace.
- **Mount** sets different mount points for each namespace and provides `umount(8)` and `mount(8)` capabilities for every process without touching the global mount points. Note that changes to the file system will still be reflected globally and not just for the current namespace.
- **PID** namespaces isolates the process ID number space. With PID namespace isolation the child namespaces do not know that the parent's process exists, but the parent namespace has a complete view of the child processes exactly as if it were another ordinary process. These namespaces can be nested, i.e., a child process spawns another child process in a new PID namespace, and so on. PID namespaces can only be created by using the `clone()` system call at the time a new process is spawned.
- **User** isolates the user and group ID space within the namespace from the outside. This will make it possible for a process inside the namespace to have root privileges only inside the namespace but not outside the namespace.
- **UTS** makes it possible to set other host names and domain names than the system has.

2.6 Possible vulnerabilities and weaknesses in isolation techniques

Some of the previously described techniques were developed for the sake of adding security to the operating system, while some were developed for other reasons but ended up providing more security and are used for that purpose within the limitations of this thesis. Whatever the reasons for implementing these techniques in the first place was for security perspective or not there exists various ways for an malicious application to circumvent the isolation(s). Below we will discuss some vulnerabilities and weaknesses.

Privilege escalation is when an application is exploiting a bug or design flaw in the operating system to obtain a higher access grade than it is meant to have. When an application gets higher privileges than it is supposed to it can execute a wide variety of malicious operations such as reading more files or deleting information [24].

For isolation techniques there exists many attacks and new ones pop up each day. Some of the main attack vectors [25]:

- **Kernel exploitation**: Since the kernel is shared among the sandboxes and the host this magnifies the importance of having a secure kernel. If a security flaw is found or if the sandbox causes the kernel to crash the entire host goes

down. In pure VM this is not as big of a problem since the attack has to reach the hypervisor, which generally is harder.

- **Denial of service (DoS)**: This attack is also tied to the fact that only one operating system is used. If one sandbox uses all of the system resources then the others will be left out and a DoS has occurred. This can be achieved through for example privilege escalation.

- **Isolation breakout**: If an adversary gains access to an application inside a sandbox he should not be able to gain access to the isolation technique itself since if he does he gets the same access rights as the sandbox has. This means that if the sandbox requires root access and is broken an adversary also has root access to the system.

- **Poisoned images**: If the sandbox uses images the image could have been tampered with. If a malicious image is being run both applications and the data inside the containers are at risk. Even though the images may not intentionally contain malicious code there still can be versions of software inside them that is old and contain security bugs.

In a study from 2015 conducted by Forrester Consulting on the behalf of Red Hat apparently 53% of IT operations in North America see the security concerns as the biggest problem for not using containers. This means that there should be steps taken to ensure the actual security before integrating containers in a production environment [26].

For the sake of LSMs one of the major concerns regarding the security is the complexity of the actual LSM. If the generation of policies or profiles is too complex there will always be the possibility of forgetting or lacking something in the configuration of the products. This can lead to applications that can break the isolation if the correct access rights have not been set on all the various files for example.

2.6.1 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a reference model for publicly known security vulnerabilities and is maintained by a corporation called MITRE and funded by the National Cyber Security Division of the United States Department of Homeland Security. Each CVE has an id number to make it easy to share the data among databases and tools so a baseline for coverage can be established. When creating a new CVE first a potential security vulnerability has to be found and reported, then this information is assigned a CVE id number by a CVE Numbering Authority (CNA) and gets posted on the CVE List (MITRE is the primary CNA but there exists some others as well). The CVE Numbers may take some time to appear on the list if the vulnerabilities are not made public, and it could take years for them to appear [27].

2.7 Licenses

Since all software is licensed under different licenses we must assert that it can be used in our use case and test suite (the use case and suite will be covered in Chapter 3 and 4. There exist a wide variety of licenses but only those that is being used by the isolation techniques is going to be covered in this dissertation.

2.7.1 GNU GPL

GNU GPL, short for General Public License and originally written by Richard Stallman. It is based on four levels of freedom. The levels are;

- **Freedom 0:** Freedom to use the software in optional purposes.
- **Freedom 1:** Freedom to examine the program to understand its use and use this knowledge for own purposes.
- **Freedom 2:** Freedom to distribute copies freely to help others.
- **Freedom 3:** Freedom to modify the software, adapt it to your own requirements and distribute the improvements to others.

The first version dates back to 1989 and currently version 3 is the most recent version. Version 1 is obsolete but both version 2 and 3 are still being used where version 2 is the more successful one [28]. The accessibility of source code is a requirement to be able to meet the above freedoms. Another aspect is that GPL is a copyleft license, which implies that work and software developed from the source code with GNU GPL license has to be released under the same license terms.

2.7.2 GNU LGPL

GNU Lesser General Public License (LGPL) is as well as GPL published by the Free Software Foundation. This license follow the same version numbering as GPL for parity. LGPL allows other people and businesses to integrate software that is released under this license without being forced to release their own source code. However LGPL has the requirement that code released under it has to be modifiable by end users (through the source code). LGPL is mostly used when releasing software libraries [29].

2.7.3 Apache Licence, version 2.0

The Apache License allows the modification of other programs, but to do this one has to explicitly explain what your own code is. It provides freedom to change, distribute and distribute modified versions of the software without having to be concerned about royalties. The Apache License also requires the preservation of the copyright notice and disclaimer. All software that is licensed under this has to include a copy of the license in a text document. Version 2.0 was released in 2004 and the main changes from previous versions was that it became easier to use the license and that the compatibility with GPL-based software was improved [30].

2.8 Benchmarking tools

Since embedded systems more often than not are dependent on running software in an efficient way and have a very small performance buffer the possible performance overhead needs to be measured. The measurements are made by two different benchmark tools, and are described in the following sections. These two tools are well used and the tests cover most parts of the system.

2.8.1 NBench-byte

NBench-byte is a benchmark program that was developed in 1995 by BYTE magazine and is used to measure the CPU, FPU and memory speed of the system. The benchmark runs in its original mode ten different tests [31]:

- **Numeric sort**: Array sorting of long integers, and is meant to test performance of cache.
- **String sort**: Sorting of an array with various sized strings and is designed to test the non-sequential performance of caches.
- **Bitfield**: Runs bit manipulation functions and are meant to see the "bit twiddling" performance.
- **Emulated floating-point**: Calculates floating-points and points towards the overall performance of the system.
- **Fourier coefficients**: Does numerical analysis for waveform approximations and is used to test the performance of the floating point unit (especially transcendental and trigonometric performance).
- **Assignment algorithm**: Moves large chunks of integer arrays in both row-wise and column-wise fashion. This test should give high results if the cache or memory has good sequential performance (since memory is just altered in one place and not moved around as in sorting operations).
- **Huffman compression**: This is a combination of "bit twiddling", byte operations and overall integer manipulation. The test should point to the general performance and be a good measurement in general.
- **IDEA encryption**: Is a block cipher algorithm and moves data sequentially in 16-bit chunks. Is designed to provide a pointer to the raw speed of the system.
- **Neural net**: Does floating point tests on small arrays that depend on exponential functions. It does not show that much since the arrays are small (and thus the cache/memory architecture would not come in to play) and it is not so dependent on floating point unit performance due to the small arrays. Is meant to show overall performance of the system.
- **Lu Decomposition**: This is an algorithm for solving linear equations and tests floating points through movement of arrays in row-wise and column-wise fashion.

NBench has some shortcomings though. The benchmark is only testing the theoretical limits of the CPU, floating point unit and memory architecture of the system and can not measure disk or network throughput which then has to be tested elsewhere. NBench also does not support multi-threading, which means that the benchmarks use a single execution thread. This leads to that the performance of scalability when more tasks are executed simultaneously is not explored.

2.8.2 dbench

dbench is a tool to benchmark performance of the file system or a NFS server. It emulates the file system load that is made from the Netbench benchmark and does so by doing the same system calls and I/O operations as the samba server would produce if it was executed in Netbench. The benchmark can be used to stress the file system to see how many applications that can be executed concurrently before the operating system starts to lag but also to see at which workload the server becomes saturated. The benchmark is a simulation but if we see the operating system as a server for applications it reproduces what many applications running at once would do, i.e., large amount of files/directories that have to be created, written, deleted and read. Loading a description file that is derived from an actual capture of a netbench run makes the actual benchmark. The file is called client.txt and is about 25MB and contains roughly 500 000 operations.

dbench supports running on multiple threads and if executed with multiple threads all will get the same workload [32][33]. The benchmark runs for 600 seconds with a 120 second warm up phase and measures the number of operations that had the time to be made, which means that a larger number is better and the throughput is also better with a higher value.

2.9 Related work

Isolation of applications, processes and virtualization on Linux has been implemented many times before, for example in [34], [35], [36] and [37]. However the work before has mainly been focusing on doing the most effective implementation for hardware where there exists support for all modern techniques (for example hardware virtualization, x86 support and so on). There has been some work done for embedded systems regarding the isolation of various things, as can be seen in [38] and [39] but they focus mainly on achieving isolation, the limitations and how to use virtualization in embedded systems. This thesis will start analyzing a variety of proposed isolation techniques and check the compatibility for the specific use case that is going to be presented in Chapter 4 with limitations specified in Chapter 3 and make an evaluation to which technique should be pursued further.

Chapter 3

Description of the system

The embedded system that will be used for implementation is the Axis camera model M1065-L [40], which implies that all tests and conclusions will be made according to the results on that specific camera model.

3.1 Hardware

The chip on the camera is an Ambarella S2L63 SoC that includes an ARM Cortex-A9 CPU that runs at 816 MHz on a single core. It provides 512 MiB of DDR3 memory of which 226 MiB is available for the operating system.

The firmware, all the third party applications and data are stored on an internal flash memory with physical capacity of 512 MiB. In order to support flashing of the device over the network, maximum half of the memory can be used, i.e., 256 MiB. Otherwise new firmware would not be possible to store on the device during the upgrade. The available space of 256 MiB is then further divided to different internal partitions where one of them is used for third-party software and is limited to 26 MiB.

3.2 Software

The tested firmware is built on Linux baseline kernel version 4.4.19. By default no isolation techniques are activated and since it is an embedded system these techniques are removed from the kernel before shipping to reduce the size of the firmware. This includes but is not limited to namespaces, eBPF, seccomp, etc. Systemd is used for init system and installed applications will get its own UID and GID. The file system in use is UBIFS, which is a file system that supports extended file attributes that is needed for some LSM modules. UBIFS also has the possibility to compress all the files stored in the file system and decompress them on the fly when accessed. This function is activated for this camera to reduce the flash memory usage.

3.2.1 Build system

The build system is based on the OpenEmbedded framework, BitBake and the Yocto project. OpenEmbedded is a framework that is aimed for creating Linux

distributions for embedded systems (however it can be used for non embedded systems as well). The Yocto project is a collaboration project that brings templates to be used for various hardware architectures and is used by developers with many different underlying architectures in their products (for example ARM and MIPS-processors).

BitBake is the actual build tool used and was first a part of OpenEmbedded until it was separated out and is now co-maintained by the Yocto project and the OpenEmbedded project.

BitBake works by specifying how various packages are built and includes package dependencies, where the source code can be fetched from and configuration instructions (where to install, how to build and so on). All different packages has a specific recipe that tracks these dependencies and will perform cross compilation of the package and package that so it can be installed on the target device. There exists a stack of recipes called layers that can be added entirely and thus not has to be added one by one, and are maintained by the creator of that specific layer.

3.2.2 Updates

In order to update any software that is included in the firmware a new firmware must be distributed to and updated on each camera. The firmware updates must be manually installed either by using the AXIS Camera Management [41] or by downloading the firmware from Axis web page and then uploading it through a web interface or FTP.

3.2.3 Installation of third-party software

The third-party software can be both developed and installed by integrator, reseller or the end user [42]. The developing company itself is not in control of what software is being developed or installed on each individual device, thus backdoors or security related bugs can be implemented by previously mentioned partners.

The application to be installed must be cryptographically signed by Axis server prior to installation for them to be accepted and executed by the camera. A manifest file must be shipped with the application, this manifest file describes the application such as name, how to start the application, what requirements are needed such as API-calls, dbus and more. With the manifest file the package manager opkg [43] sets up systemd to include the service file created for this newly installed application.

Chapter 4

Methodology

This thesis aim is to reach a recommendation for the use of one or many isolation techniques in embedded systems and specifically the embedded system described in Chapter 3. The techniques are first going to be compared in the aspects of which are supported as well as lack of support with embedded systems in mind. The techniques will be a mix of containers, general sandboxes and LSMs that are well known and widely used to ensure good quality of the product. The techniques within each field were chosen to try and cover the most popular techniques based on search engine hits but also to cover as many different underlying techniques as possible. In general there were difficulties finding useful information regarding what techniques even exist on the market. The dissertation tries to cover every relevant technique we could find.

The elaboration of techniques and comparison is made so that the reader will get a grip on a variety of techniques, what security measurements are being supported and how different techniques solves the same problem. Also for the sake of bringing substance to why these techniques even will provide enough isolation for them to be used in a release grade operating system. This part of the thesis is, as stated, going to be made completely detached from the embedded system world and just focus on providing enough substance to be further investigated and in the long run made possible to be used on the embedded system of our use case. During this elaboration phase the licensing part is also going to be covered, and made sure that all of the techniques being used is available to be used within the limitations of license requirements.

When the found techniques have been elaborated and explained the actual implementation was going to be made. There are no certainties that all of the mentioned techniques are possible to use in our system, and thus the result is divided in to three parts. Running techniques, non-running techniques and techniques that had the theoretical possibility of running but are either too difficult or too time consuming to be added to this thesis. To prepare for the work on the actual embedded systems all of these techniques were tried out on x64 virtual machines first, for the sake of making sure that they were running on systems with practically no limitations and to get familiar with the techniques. Embedded systems imply more limitations than ordinary systems and thus there are some aspects that have to be taken in consideration when choosing the technique or implementation that is best suited for the use in this dissertation [44]. These aspects are the following:

- The system must be dependable. For example if the device is to be used in the video surveillance business the cameras and other units that are using the operating system must be reliable. Thus the isolation technique must be reliable, but also have high maintainability so that if there exist some errors in the code it can be fixed quickly and deployed. These two aspects together gives us availability which is the term we will be using to address these issues.
- The system must be efficient. An embedded system should use the available hardware and operating system functions as much as possible. For example containers should provide minimal overhead, unnecessary functions should be avoided and the code should be minimal to increase the efficiency.

With the limitations described in Chapter 3 the goal is to find one or many techniques that are recommended to be used and further researched. One solution can of course be to combine many different techniques, but that leaves the question whether the security will be improved by an acceptable level or if it just adds more complexity and overhead. The selection will be made according to the techniques by themselves and how they fit in to the requirements and not according to how many different tools they can be combined with. The Linux kernel features that were explained in Section 2.5 will not be taken in consideration in the comparison. Containers and LSM are often using these features and alone the Linux kernel features are not flexible and usable enough to use in real practice.

When all possible candidates are tested and possibly running on the camera test suites will be executed and validated with regards to CPU usage, memory usage, possible added latency for execution, input and output performance of file operations and disk usage. If the results do not point to a single specific technique that stands out as better all others factors of usability, maintainability, stability and possible security aspects will be taken in to consideration and added to the final assessment of the recommended technique. The tests and benchmarks are going to be made on the same physical camera for all the tests. Since continuous updates of the source code and upstream packages are being made same source code revision is used in the entire thesis to ensure that all the benchmarks is as valid and accurate as possible.

Another approach to the testing phase would have been to beforehand choose one technique from the chosen techniques after comparing them theoretically and go deeper to just ensure so that specific technique is running and perhaps start the work of automating the process of auto enabling an isolation technique when an application is loaded to the camera. This approach was however discarded since had no prior experience at all testing isolation techniques within their build system and since the isolation techniques in their base form are not designed for embedded systems a complete evaluation is needed.

When assessing the security features they are supporting the assumption is going to be that these isolation techniques are providing what they are promising and does so bug free (otherwise much time has to be spent on elaborating the different attack vectors of the chosen techniques). However there will be an assessment regarding how many weaknesses/vulnerabilities that has ended in a Common Vulnerabilities and Exposures (CVE) has on them and the possible ef-

fects from these that possibly is going to factor in on the recommended technique or techniques.

The thesis will also assess if the technique is under active development so that if (most likely when) any bugs are found the maintainers or other people/organizations can supply a patch and fix the problem.

4.1 Use case

The end user is Axis. Wishes has been expressed that they are interested in a solution that helps them add extra security to the application section of their cameras. The usual scenario is going to be that an end user (i.e., Axis customer) buys a camera for the purpose of surveillance. This customer may then want to add some of their own functionality or applications to the camera and that is where this thesis will try to improve the cameras. Axis cameras should be flexible enough to allow users adding their own application but should do so in a responsible way and these applications should not crash or make the system panic.

4.1.1 Use case

Application execution use case:

- **Use case 1**: An application is executed to the camera
- **Actor**: Customer who bought an Axis camera
- **Stakeholder**: Axis
- **Use case overview**: The customer wants to add some extra functionality to the camera for surveillance purposes, and does so by developing an own application that is loaded to the camera and executed
- **Trigger**: Application is executed on the camera
- **Precondition 1**: The application is transferred to the camera
- **Precondition 2**: There is space left on the camera

Basic Flow: Ordinary execution

- **Description**: This scenario describes the situation where an application is executed with no malicious attempts and is the main success scenario.
- **1**: User starts the application
- **2**: An isolated environment is started
- **3**: The application is executed
- **4**: The application makes all needed system calls
- **5**: The application runs its course and is terminated
- **6**: The isolated environment is removed

Alternative Flow: Execution with malicious attempt

- **Description**: This scenario describes the situation where an application, intentionally or unintentionally does malicious operations.
- **1**: User starts the application
- **2**: An isolated environment is started
- **3**: The application is executed
- **4**: The application tries to do malicious things
- **5**: The application is denied of doing these malicious things
- **6**: The application is terminated
- **7**: The isolated environment is removed

Chapter 5

Isolation techniques

The different isolation techniques will be introduced and explained one by one divided into LSMs and sandboxes. The explanation of each technique will focus on the parts that is relevant for this dissertation, i.e., the security properties. Since each technique is unique in how and what in the system that can be secured examples will be shown how a certain acceptable level of security can be achieved by each technique.

5.1 Linux Security Modules

Ever since LSMs were included in the Linux kernel there has been a growing number of security modules being developed. This section is going to cover the four largest security modules that are included in the kernel and that are available on the operating system flavor that the use case of this thesis is focusing on. There is one more available LSM called YAMA that has the possibility of being used within the range of the use case but was discarded as the amount of information regarding it is parsimonious. A brief comparison between the considered LSMs is shown in Table 5.1 but are further examined and explained in the coming subsections. Each row in the table is defined as below:

- **Version** - specifies what version each technique used throughout this whole dissertation.
- **License** - that applies for each techniques, see Section 2.7 for description.
- **Policy generation** - states if there exists tools to automatically generate policy based on the learning mode.
- **Label or path enforcement** - defines if the technique requires labels on the file attributes or is using the path to enforce the access control.
- **Learning mode** - means support in the technique for a specific mode where all calls will be allowed but logged so that application specific inspection is later possible.
- **Configuration** - states how the configuration of the LSM is done.
- **System call overhead** - show in percent how much overhead each technique adds compared to a system running without any LSM activated.

Name	SELinux	AppArmor	SMACK	TOMOYO
Version	2.6	2.3	4.4	2.5
License	GPL v2	GPL	GPL v2	GPL v2
Policy generation	Yes	Yes	No	Yes
Label or path enforcement	Label	Path	Label	Path
Learning mode	Yes	Yes	Yes	Yes
Configuration	C-like	ACL	Domain table	Interface
System call overhead [45]	7%	2%	5-10%	N/A

Table 5.1: Matrix comparison over the covered LSMs.

5.1.1 AppArmor

AppArmor is a running daemon that restricts programs by a set of rules stated by the profile for that binary by using path name based security. White-listing of allowed rules are stated in the profile file. AppArmor has been integrated in the Linux kernel since version 2.6.36 [46].

By using Linux Security Modules (LSM) the daemon can check that the specific application is allowed the resources it asks for. AppArmor has two states, enforcement or complain. The enforcement state will report violations to syslogd and also enforce the program to follow the rules stated in the profile. Complain will only report violations but still allow the program to run even after a violation of the rule.

AppArmor applies the profile to a process when `exec(3)` is called. The profiles are text files that contains the following access control: system capabilities, files access and modifications, mount, network, dbus, IPC and rlimit [47] [48].

AppArmor is often seen as easier to use than other LSMs while still having powerful access control and is still under development.

5.1.2 Smack

Smack is an acronym for Simplified Mandatory Access Control Kernel and has been in the Linux kernel since 2.6.25 and is still under development [49].

The MAC scheme that is used in Smack is based on labels that can be attached to processes and typically file objects. To receive clearance that a process can have access to the object the labels must match (there exists some pre-defined system labels but the majority has to be set). Smack only allows privilege to be a factor when labels of tasks shall be changed, and leads to that the security in Smack is more seen as an attribute of the process and not the program. If a label is given to a new storage object only a process that is privileged has the ability to change that label. Smack uses but do not require extended attributes (xattrs) to store the labels on the file system objects. If the xattrs support is lacking a default label can be used for the whole file system[50]. In order to change labels of a file the process also must have `CAP_MAC_ADMIN` (see `capabilities(7)` in the Linux manual).

The labels have different access rules that are in a predefined order and are defined in text files located under different directories below `/etc/smack` The precedence and rules are[50]:

- *** - star:** The star label is set on some objects that needs to have universal access. A process with this label does not have access to other objects (even including others with star label). Vice versa all other labels have access to objects with a star label.
- **_ - floor:** This is the default label for system files and system processes. Every process has read access to objects with this label.
- **^ - hat:** Processes with hat label has read access to all other objects.
- **Matching labels:** If a process has the same label as an object the process will get access (except the star label).
- **Unmatching labels:** Since users can define their own accesses explicitly one could give access to process and object labels without them matching. So if there exist such an access defined it gets access, otherwise not.

Smack works on any Linux distribution and is used on embedded systems with Tizen (open source mobile operating system based on Linux) and Intel IoTs open source project although in modified versions. It can control access to files, IPC, sockets and processes.

5.1.3 SELinux

SELinux, short for Security-Enhanced Linux is a LSM that provides MAC, enforcement and Bell & LaPadula sensitivity for Linux and has been in the kernel since version 2.6 [51]. SELinux puts a label with each program that states the security characteristics of a process that runs that specific program and supports learning mode. The way labels are set on the objects is in the file system xattrs field and supports restrictions on file system, network, IPC and more.

By default SELinux denies access to objects. A subject or object is governed by a security context divided into three different parts; user, role and domain:

- **User** in SELinux is separated from the Linux DAC and contains information of what privileges the user has. A user in this context is either a user or a daemon.
- **Role** can be seen as Linux equivalent of group, i.e., a user can be a member of several roles. But SELinux only allows one role at a time and switches between different roles must be explicitly allowed for that user.
- **Domain** can be seen as a kind of namespace, any combinations of subjects and objects within in the same domain will be able to interact with each other.

A system that enforces SELinux must take into account applications that run other applications and processes. The profile for these applications must have all the sub-processes permissions set in the profile as well in order to fully function. This can thus lead to large profiles to include all needed resources. This can be a drawback for embedded systems. If we take Busybox (a software that has several stripped-down Unix tools in a single binary) as example this means that it has to be given all the rights for what the program can do and the policy must then

be programmed to take into consideration these aspects which can create massive policies that are hard to incorporate to third party programs (since Busybox can be changed when updated). The policies for these binaries will generally be large (excess of 800 000 lines is not uncommon) and when the binaries are changed the policies and file system might need an update.

5.1.4 TOMOYO

TOMOYO uses MAC with support for learning and supervisor mode for restriction of applications. Configuration files are used to fine-grain policy for every application on the system. TOMOYO sets restrictions of the application based on the path to the executable. It comes in two different versions, version 1.8 and 2.5, both being very different from each other in terms of modification of the kernel [52].

Version 1.8, which is not classed as a LSM but modify the kernel in a way that requires recompilation of the kernel to extend the possibilities for it to run. Support for kernel versions prior the introduction of LSM is possible and version 1.8 can be implemented on kernel version 2.4.37 [53]. Version 2.5 is developed as a LSM without having to modify the kernel and is included in the kernel from version 2.6.30 [54]. This dissertation will focus on the latter since upstream LSMs in the kernel is preferable compared to patch the kernel to support version 1.8.

TOMOYO can apply access control on files access and modifications, mount, network, dbus and IPC and TCP-wrapper-like TCP/IP packet filtering. The latter can be based on both address and ports that could operate as a firewall for each connection being made from running applications. It even allows restriction of the environments variables names that are passed to system calls so that dangerous environment variables (like `LD_PRELOAD` to `execve()`) won't be passed around freely. One downside could be the overhead, however according to their own tests the amount of overhead is acceptable in most cases [55], but of course our own tests were made.

TOMOYO uses path name based access control, which can be discussed to be seen as less secure than label based access control. For example if a binary changes name the access control will fail. This disadvantage exists for other techniques that also use path based access control. In this dissertation AppArmor is an example of that. It can be difficult to maintain and update policies if and when renaming of files in the system occur. TOMOYO has some assists to this and the damage it may have on the system by restricting path names, which each application can request but it still remains a possible drawback.

5.2 Sandboxes

This section will describe some sandboxing implementations developed for Linux with the use of the kernel capabilities mentioned in the previous chapter. Observe that the versions used for the different techniques might not be the absolute latest. The reason for that are limitations in the build system described in Section 3.2.1. A brief comparison that can be used as an overview for the techniques is shown

in Figure 5.2 and 5.3 and each row is described below:

- **Version** - specifies what version each technique used throughout this whole dissertation.
- **License** - that applies for each techniques, see Section 2.7 for description.
- **Isolates network** - separates the network from the underlying network so no applications with isolation of the network can see packets outside of this network. Could be to only show a private loop-back interface.
- **Requires root** - states if the sandbox needs to be run as root to function.
- **Configurable auditing** - states if the technique support individual configuration for each application, isolate network for example or only allow read/write access from some specific path.
- **Requires separate file system** - for the sandbox to function, i.e., a complete separate operating file system.

Name	LXC	Firejail	Bubblewrap	Docker
Version	2.0.0	0.9.38	0.1.6	1.13
License	LGPL 2.1+	GPL v2	LGPL 2+	Apache v2
Isolates network	Yes	Yes	Yes	Yes
Requires root	Partially	Yes	No	Partially
Configurable auditing	No	Yes	Yes*	No
Requires separate file system	Yes	No	No	Yes

Table 5.2: Matrix comparison over the covered sandboxes.
*: With support of seccomp and SELinux policies (these techniques must then be activated in the kernel)

Name	Systemd-nspawn	Systemd-service	Rkt
Version	231	231	1.25
License	LGPL 2.1+	LGPL 2.1+	Apache v2
Isolates network	Yes	Yes	Yes
Requires root	Yes	Yes	Partially
Configurable auditing	No	Yes	No
Requires separate file system	No	No	Yes

Table 5.3: Continuation of the matrix comparison over the covered sandboxes.

5.2.1 LXC

LXC, or Linux containers, is an application that uses namespaces (see Section 2.5.5) to achieve process or complete virtual OS isolation. LXC was added to

the Linux kernel in 2.6.32 but required then to be privileged to start a container. Support for unprivileged mode was added in version 3.13 of the Linux kernel [56] [57]. LXC is developed under the license GNU LGPLv2.1+ and is maintained by people working at Canonical Ltd.

The containers made by LXC can, as previously stated, be either privileged or unprivileged, i.e., root or normal user. The privileged containers should only be used in specific environments where unprivileged containers are not available (this is since the container's users get root access to the host which is insecure). For both privileged and unprivileged container Linux Security Modules (see Section 2.5.1) SELinux or AppArmor can be applied as well as seccomp or capabilities to add an extra layer of security [57]. A container can be seen as privileged if UID 0 of the container is mapped to UID 0 of the host. If a container is privileged the security comes from MAC, seccomp filters, namespaces and dropping of capabilities. If an error or bug exists in these security mechanisms it will be possible to escape the isolation and then the application escaped would have root privileges on the host. The privileged containers can typically be used to prevent damage on the host such as reconfiguring hardware, kernel or accessing the file system.

Unprivileged containers are safer by design, as explained in the preliminaries (Section 2.6). The UID 0 inside the container is then mapped to an unprivileged user outside of the container and can thus only access the owners privileges. When creating and starting a container as an unprivileged user, setuid is used for the parts that needs to be run as a privileged user.

As stated LXC does not require root to isolate applications and can use policies from seccomp, SELinux or AppArmor. Since isolation of a complete OS is possible multiple processes can be isolated together under LXC and can provide higher usability and maintainability for inter-process communication. It supports all file systems and is still being developed. On the network part however there can only be disconnection from the network but not filtering. It does lack the possibility to run a trace mode to test the applications system resource usage.

Configuration to achieve isolation

First a container needs to be downloaded that contains a complete OS file system. The application to be executed must then be copied to the container's file system since the container will be isolated from the host's file system. To execute the application in an isolated environment the following command should be used:

```
lxc-start -n name-of-container name-of-application
```

Note that execution of lxc-start is available without any special privileges once LXC has been installed and setup to allow a specific user to handle containers. All the containers are run under the same PID as the user executing the commands. Root is only needed when setting up the container the first time and the sticky bit on the binary achieves this.

5.2.2 Firejail

Firejail is a sandboxing program that tries to restrict the environment that is running untrusted software using Linux namespaces (see Section 2.5.5) and seccomp-

bpf (see Section 2.5.4). To achieve this isolation it requires root to run. Firejail can be used by itself or combined with other isolation programs to add more functionality.

It is lightweight with very few dependencies as it only requires the Linux kernel of version 3.5 or higher to run. It supports, among other things, sandboxing servers, user login sessions and graphical applications. Firejail comes with certain profiles pre-built for various Linux applications such as Chromium, VLC and Firefox, and if no profile is made for an application a generic one is used. The profile that the sandboxing is made according to can also be changed to a custom version where each user in the system can specify their own profile. To launch an application sandboxed by firejail it has to be passed as an argument to firejail [58].

A simple example how Firejail can be used is shown below:

```
firejail --net=eth0 --dns=8.8.8.8 \
  --netfilter=/etc/firejail/nolocal.net \
  --private=/home/firejail/fake-home/ firefox
```

This example launches Firefox but changes so that it uses one of Google's public DNS server [59]. It also adds restrictions to the private network so only connection outside of the local network can be made. `$HOME` is set to `/home/firejail/fake-home/` so no files from the user's home folder can be read by the Firefox binary and the files written will be stored persistently in the same path.

Firejail provides a private view of the global system resources, for example network, processes and mount table. It can be run in an SELinux or AppArmor environment and can wrap anything in seccomp easily even if the underlying application does not support it. It supports all file systems, and is being continuously developed and maintained. Firejail supports tracing, and can show exactly what the application needs access to.

5.2.3 Bubblewrap

Bubblewrap is a tool for wrapping. The main goal for Bubblewrap is to sandbox applications where the access is restricted to parts of the operating system or the user data. To create the sandbox an executable is passed on as an argument and then the executable is executed in a custom namespace. This namespace starts out empty and then the actual sandbox is built from arguments in the command line [60]. Bubblewrap can be run as a normal user and thus privilege escalation will not be as damaging since the user executing Bubblewrap is limited in the system [61]. This leads to some limitations where some techniques in the kernel cannot be used for Bubblewrap, one of those is `iptables` for advanced filtering on the network but denying of non-local network is possible. System resources can be isolated and seccomp-filter is supported. Bubblewrap is continuously being developed.

Each application to isolate needs to be passed as an argument to Bubblewrap (binary is named `bwrap`). To achieve the isolation for one application an example script is shown below.

```
bwrap --ro-bind /bin /bin \
      --ro-bind /usr /usr \
      --ro-bind /lib /lib \
      --ro-bind /lib64 /lib64 \
      --ro-bind /etc/resolv.conf /etc/resolv.conf \
      --chdir / \
      --proc /proc \
      --unshare-pid \
      name-of-application
```

5.2.4 Docker Engine

Docker engine provides a way to run applications in a container, with support for specific versions of libraries for each container. The Docker engine can run on several different host OS (Linux, macOS and Windows) and thus adds capabilities to run a docker instance on several different OSes. The Docker engine described here is the one running on a Linux host and relies on containerd, runC and libcontainer. The latter two is part of the Open Container Initiative (OCI). Containerd is a daemon for controlling the OCI tools. To run, create and manage containers the docker command must be executed as root.

Docker is mainly focused for development and testing and not deployment even though the focus has shifted to deployment. Docker imitates an entire OS and requires space for the container's filesystem. Docker is still under development and is built on running a single process in a single container with its own private network or several images connected to the same isolated network. No system resources are passed on to the container by default so Docker gives complete isolation, and thus cannot provide tracing. Docker states itself as "quite secure" and general opinions has stated that Docker is less secure than competitors [62].

5.2.5 Systemd

Systemd is an init system for various Linux distributions and has the ability to bootstrap all the processes and user space. It runs as PID 1 and starts up the system while providing aggressive parallelization, d-bus and socket activation for starting services, starting daemons in an on-demand fashion, process tracking through cgroups, maintaining mounts and mount points and supports snapshoting and restoring. Its main focus is being an init system but it has some security aspects as well. One is the use of unit files and the other is nspawn, which will be described in the coming sections.

Nspawn

Systemd-nspawn is a part of systemd that provide containerization capabilities and tools to manage these containers. It makes the file system, process tree and inter-process communication namespaced [63].

> Note that even though these security precautions are taken systemd-nspawn is not suitable for secure container setups. Many of the security features may be circumvented and are hence primarily useful to avoid accidental changes to the host system from the container. The intended use of this program is debugging and testing as well as building of packages, distributions and software involved with boot and systems management.

This quote comes from the manual and proves that this tool perhaps not should be used as a function meant for security [64].

This tool supports both booting of complete OSes as well as running a single process within the container. It needs to be run as root and supports system resource isolation as well as simple network filtration with support for exposing specific ports. Several different containers can be combined in the same virtual network. One possible drawback can be that if there is a stability problem with systemd the whole system will crash due to it being used as init.

To sandbox a complete OS (an Alpine image is used for this purpose) the Alpine Linux image needs to be downloaded and extracted to an arbitrary location then the application to be run needs to be moved in to the file system of the Alpine file system. To execute the binary with no network capability the following command can be used:

```
systemd-nspawn -directory=/var/lib/machines/alpine
-private-network /bin/APPLICATION
```

Unit

In systemd a unit is a resource that the system can operate and manage and is configured using configuration files that are called unit files. Systemd has the possibility of also isolating certain things without using nspawn as a container but instead by specifying in the actual unit file what that service is allowed to do. It does so by using capabilities and namespaces and only supports restriction of processes instead of entire OSes as can be the case in nspawn (and shall not be mixed with nspawn even though they use the same basic kernel functions).

When compared to other init systems a unit can be seen as a job but a unit has a more wide definition and supports the abstraction of service, network resources, devices, file system mounts and resource isolation. There exists many types of units, for example `.socket`, `.device`, `.mount`, `.service` and many more. Socket units describes network or IPC sockets, device units describes designated devices that needs systemd management and mount units defines mountpoints on the system. The unit most interesting to this dissertation is the service unit. It describes how an application or service is being managed on the server, which includes how to start it, stop it, if it is to be automatically started, what it should restrict in terms of resources and so on.

For these units there exists various options to modify that unit service, but those most important for the isolation part of the processes are:

`Systemd.exec(5)` supports the loading of SELinux (see Section 5.1.3), AppArmor (see Section 5.1.1) or SMACK (see Section 5.1.2) policies when executing a

service as well as set restricted access to certain system resources. The restrictions in systemd.exec(5) includes but not limited to:

- **Directory and files:** change root and working directory as well a private `/tmp` directory and specify which directories are prohibited and not. Options to set umask for files created by the process. The option to do this is called `RootDirectory`. Through `InaccessiblePaths` there exists options to set up a new file system namespace that can limit process access to the file system hierarchy. The responding for users can be done as well, through `PrivateUsers`, that will set up a new user namespace for the process and makes a minimal user and group for that process.
- **CPU and memory consumption:** set limitations on memory and CPU usage and prioritizing of the process and is done for example by the options `LimitCPU`, `CPUAffinity` and `MemoryLimit`.
- **Devices:** only allow certain devices such as `/dev/sda` and set private network so only a private loop back interface will be accessible to the process. Device management can also use the option `PrivateDevices` to turn off physical device access by the running process, which may add security. There exists an option called `ProtectSystem` that also can make the system more secure through mounting directories as read only that can need the device access to be configured to work as intended.
- **Syscall:** filter on predefined system call sets that are allowed for the process. `SystemCallFilter` is the option used and uses seccomp filtering to do the actual system call control.

An example of an unit file that starts nbench and outputs the results to a file while limiting access to some paths and devices, runs as a new user and only allows system calls for changing resource limits, memory and scheduling parameters is shown below:

```
[Unit]
Description=nbench
InaccessiblePaths=/
ProtectSystem=true
PrivateUsers=true
PrivateDevices=true
SystemCallFilter=@resources

[Service]
ExecStart=/bin/sh -c "nbench 2>&1 > /usr/local/addon/nbench.txt"

[Install]
WantedBy=multi-user.target
```

5.2.6 Rkt

Rkt [rock-it] is a container manager for Linux. The main focuses while being developed was that it should focus on security, simplicity and composability. It is

designed to run individual applications and not a whole OS. It requires root for managing the containers and works together with systemd and can use nspawn for the creation of namespaces for containers. It is still under development. Instead of having a background daemon rkt uses an interface that compromises a single executable. Rkt requires root to run and requires the container to be made of images following the OCI specification. These containers needs to be created from somewhere and tools to build them are for example acbuild, acttool and goaci. Rkt also has the possibility of running Docker images. Even though parameters are defined in the `config.json` in the container image they can be overridden at runtime, and as an example `rkt run` will allow users to input their own execution arguments to images.

Since Rkt is compatible with the OCI standard different containers can share resources such as network to communicate with each other. Rkt itself states that it is designed and developed with the feature of being "secure-by-default" and is bundled with support for running various LSMs, make TPM measurements and having hardware-isolated VMs [65].

Chapter 6

Results

The result part of the thesis is going to be split in three parts, implementations that had no possibility of being run within our specified use case, implementations that had the theoretical possibility of being added and executed on the camera, but because of various reasons could not be added to the final image and lastly implementations that were able to run.

6.1 Non running isolation techniques

There were two implementations that had no possibility to be added to the image at all. Rkt lacks support for the ARMv7-processor that is used and without upstream support addition for ARMv7-processors by CoreOS (developers of rkt) this implementation will most likely never be able to run.

For Docker some recipes already existed in various layers in Yocto. To be able to fully run Docker the layers `meta-virtualization`, `meta-python`, `meta-networking` and `meta-filesystems` had to be added. This proved to be a moderately complex task since it was necessary to rewrite and specify our own manifests (BitBake file that basically states where to fetch various sources from and what to include) to get these layers to be added to the build tree. These layers come stacked with many different things, which also add some overhead in our firmware. The first approach was to split the Docker source recipes from these layers and just add them to an already existing layer. This however proved to be impossible since the main source code is dependent on python version 3 support. Axis does not currently need python 3 for any of its packages. This python support together with dependencies of other recipes lead to the addition of very many stand alone packages which later on has to be maintained by Axis by themselves, so in unison with Axis we choose to have the extra overhead by adding the entire layers (layers caan be maintained by Yocto and does not have to be Axis burden to do so). This addition of all the layers lead to that the image needed 564916224 byte (or roughly 565 Mb) space. With about 221 Mb available for the entire operating system including addons this is about 2,5 times larger than that and could thus not be tested on the camera.

6.2 Techniques with theoretical possibility of execution

Some of the techniques that were covered by the comparison could not be added to the operating system within the limitations of this thesis but still has the theoretical possibility of being added to the image. These techniques could be a better fit than the ones that actually managed to run, so they should not be discarded completely. However in the benchmarking and final results they will be discarded.

LXC can be run as long as namespaces is activated in the kernel, but once the init sequence of the container gets to the part where it is going to connect to a pseudoterminal (a bidirectional communication channel that works like an ordinary terminal but is virtual, also called pty) it fails even though pseudoterminals are included in the build. The actual problem is not the pty but rather in the creation and transfer of the container file system to the camera. The camera system is very limited and Busybox version of xz to unpack a tar-ball, that contains the container file system, lacks support for the majority of the flags that LXC is using after downloading the container file system. So in order to get LXC fully working the need is to find a solution on how to move the container file system fully intact in to the firmware during build time or transfer it to the camera during runtime. It was not possible to use LXC own tools to retrieve a container file system.

Systemd-nspawn is supported through systemd, but the same problem as LXC had with movement and/or creation of a container file system is a factor here. For nspawn to function, namespaces and most likely pseudo terminal has to be enabled in the kernel. The pseudo terminal dependency has not been verified since a boot of a container has not been possible, see Appendix A.7 for kernel flags to enable.

AppArmor did actually run on the kernel and was showed as activated, however to be able to setup it the user space tools have to be installed. These tools were dependent of `swig` and native python which proved to be a difficult task to implement in BitBake, if even possible. The user space tools is for making profiles and enforcing the security and is not required for AppArmor to run - but AppArmor has to be configured through the kernel setup phase if not using them. Since Axis is using BitBake as build tool these kernel changes proved to be overwritten and changed in so many ways that the result did not persist to the actual built image. The user space tools did exist as a recipe in Yocto but the flavors of them were not written for the same version as Axis is using. This recipe would have required massive rewriting and configuration to be able to run since just adding the currently existing layers resulted in compile errors (of the type where the Axis main layers were incompatible with the added layers).
If the actual configuration of AppArmor could have been made by the user space tools or setup during build there would have existed a way to use it but since AppArmor requires a policy for every binary on the system (if it should be constrained by AppArmor) these policy-files must be generated and included in the firmware. How these policy files should be generated for the addons is out of scope of this dissertation but is still something that needs to be taken into consideration when choosing a LSM as a way to isolate each application.

SELinux can be built with the kernel, but since SELinux is using policies for

every application there has to be policies for all of them. These rules will be loaded during boot from a policy database. The main problem with adding SELinux to the kernel is with these policy files. When the policy files are automatically generated by the recipes that exists in Yocto the image gets too large. A possible solution here would be to craft some automatically generated reference-policies that can fit Axis needs, however we did not get that solution to run due to build and configuration problems with BitBake. The user space tools needed for SELinux could be added from a layer called `meta-selinux` but as stated the build failed when enabling the policies. With SELinux enabled the tar ball size was 53835041 byte and when extracted too large to be fit in a cameras image. Worth mentioning is that SELinux with the user space tools but without policies could be run on the image, however since no policies are there no actual improvement of security is made. The same drawback exists as for AppArmor, policies needs to be generated and maintained for all binaries to be isolated.

6.3 Running isolation techniques

The techniques that are up and running within the limitations of our use case is the one that are going to be most elaborated and graphs with benchmark comparisons are shown in Figures 6.1-6.10. When the benchmarks were being made all extraneous services were stopped on the cameras so that there would be only the bare minimum of the system that was running (only Axis own built services was stopped since they can generate various loads on the camera and then the benchmark may differ, for example if there is motions in the video data). In some cases kernel flags had to be changed in the operating system for the techniques to work, exactly what flags were changed can be found in the appendix.

For nbench there was one test that required external files, and that one was neural net benchmarks and those files could not be added to the camera. For this thesis the benchmark was modified in order to run on the destined camera and the part of neural net had to be removed. As explained in the preliminaries the neural net tests does not add that much information to the benchmark so that part of the benchmark was discarded.

Firejail was able to compile with the kernel only by adding a BitBake recipe and a dependency to that recipe, it however needed some kernel configuration for it to be able to run as planned. Namespaces and the belonging flags were not activated in the kernel, so once those flags were activated and built with the kernel Firejail was able to start and function as intended.

Bubblewrap was like Firejail able to be included only by adding recipes, namespaces in the kernel and dependency for the recipe. The recipe had to be made from scratch but did not provide any special difficulties.

Systemd unit isolation does not require any extra binary in the running operating system but namespaces must be activated to fully function and only service files (regular text files) need to be created for each application to isolate. This helps with the resource aspect. The benchmarks change a bit since there are some extra kernel features (i.e., namespaces) that had to be made through and the question whether it is safe enough is still present.

SMACK could be enabled through configuration of the kernel flags and the user space tools needed to be added as a separate layer in BitBake. SMACK is used by Intels IoT open source department and the layer was already developed [66]. There had to be some modifications done in order to make it run on Axis specific build system though (for example some versions of sshd had to be changed and reconfigured).

Since **TOMOYO** is a LSM the activation could just happen through the additions of kernel flags to the kernel during build as done with SMACK (however a completely changed set of flags). TOMOYOs user space tools was pre-added to the `meta-oe`-layer and just needed activation to be able to run. During the benchmarks TOMOYO was executed with policies that enabled all access.

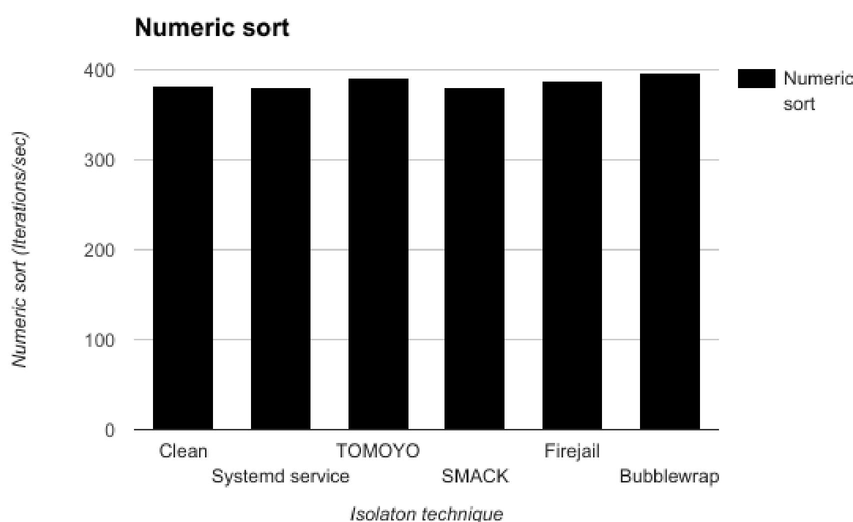

Figure 6.1: Numeric sort performance

The final sizes of the images when running these techniques is shown in Figure 6.11 and the data is taken from the compressed file systems before it is split into different partitions.

6.4 Security evaluation

This evaluation is going to cover only running techniques and is meant to give some insight to the current security aspects of these. The information regarding security concerns is based on reported Common Vulnerabilities and Exposures (CVE) and the number of occurrences should not be seen as an guidance but rather the severity of the errors since the amount could just reflect the size of the project or amount of users that tries to find weaknesses.

systemds unit isolation has the main issue that if a malicious application breaks the isolation the attacker could get access to the entire systemd and root

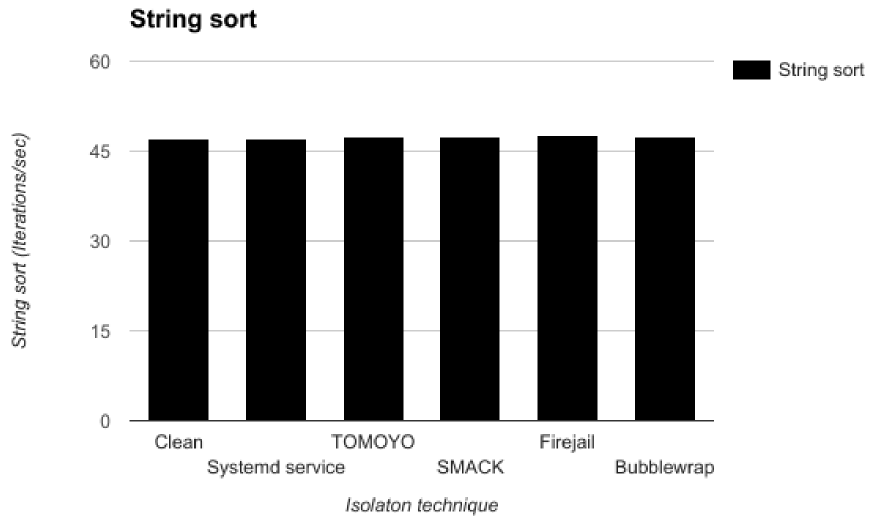

Figure 6.2: String sort performance

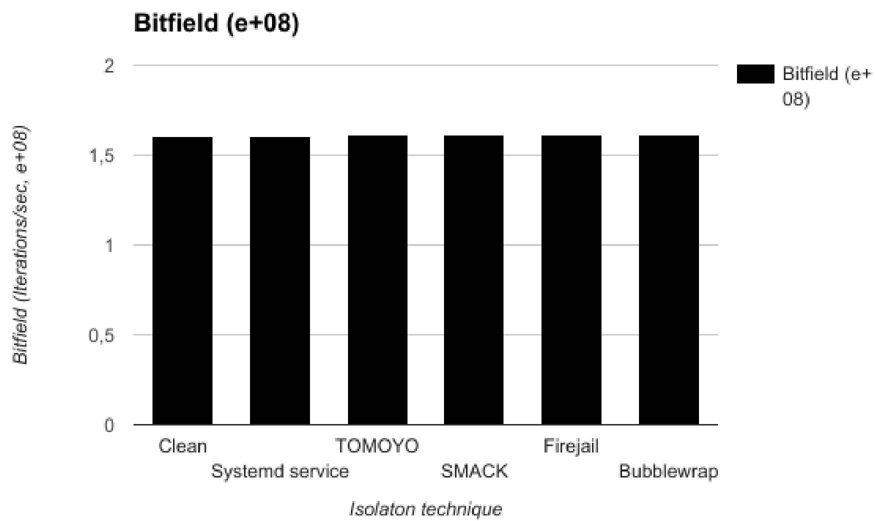

Figure 6.3: Bitfield "bit twiddling" performance

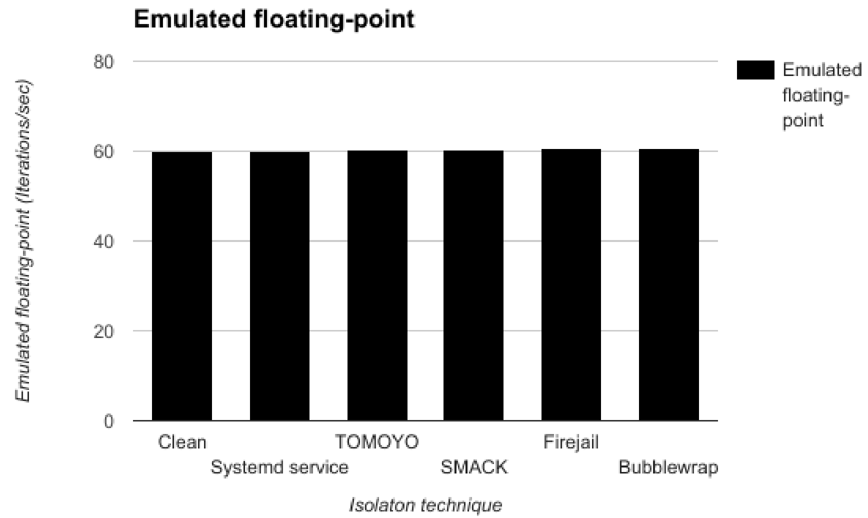

Figure 6.4: Emulated floating-point performance

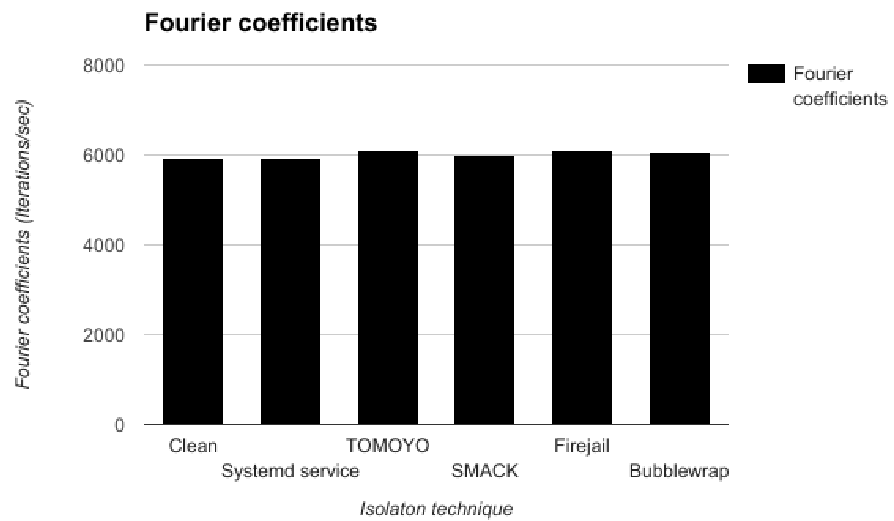

Figure 6.5: Fourier coefficients performance

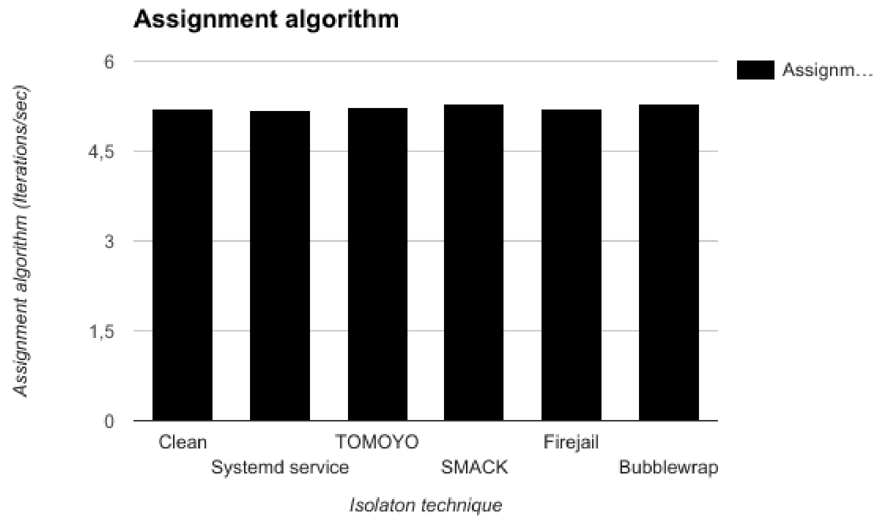

Figure 6.6: Assignment algorithm performance

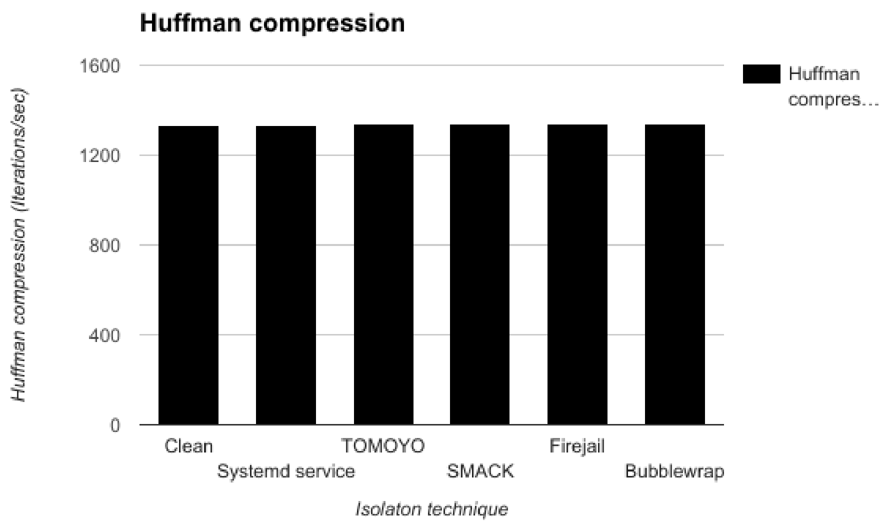

Figure 6.7: Huffman compression performance

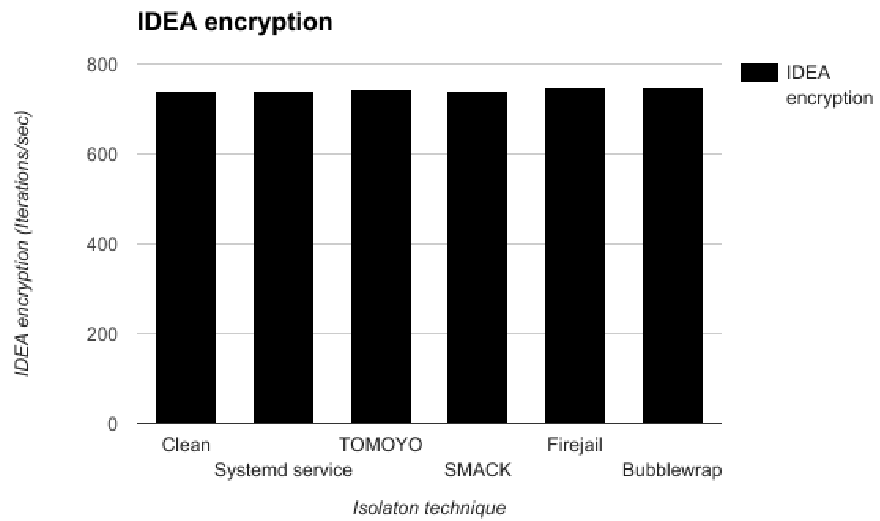

Figure 6.8: IDEA encryption performance

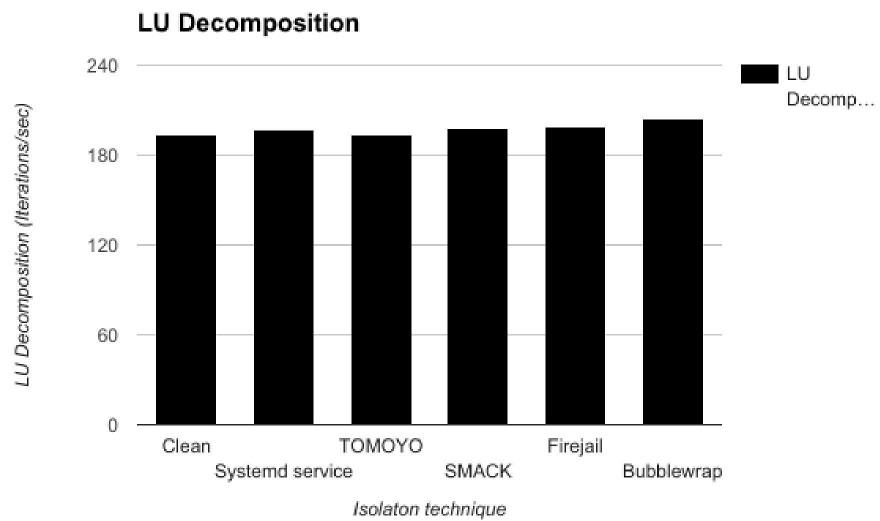

Figure 6.9: LU Decomposition performance

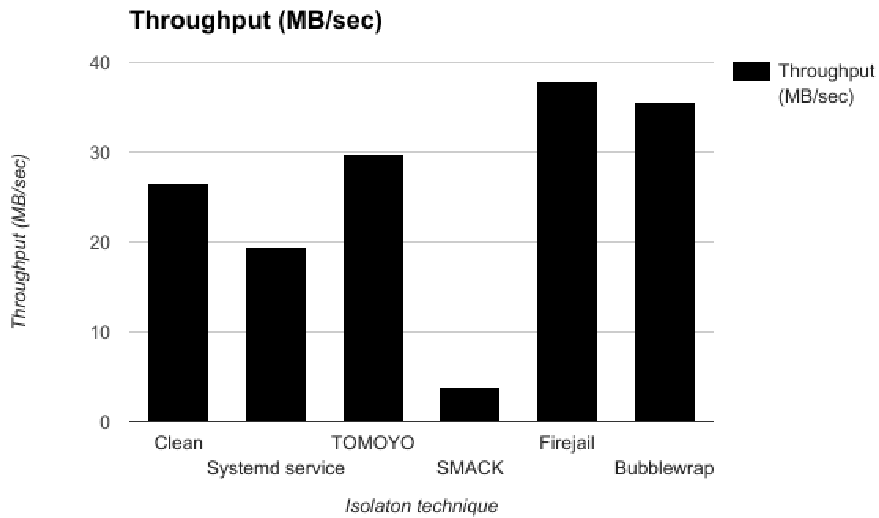

Figure 6.10: The throughput of file write and reads

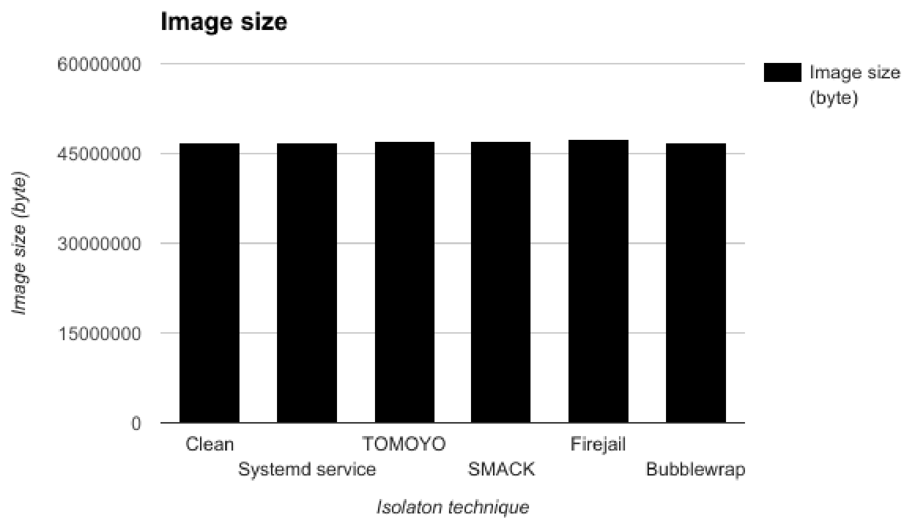

Figure 6.11: Total image size when running the isolation techniques as compared to a clean image.

access to the system since the services are started by root. This brings many possible attack scenarios since systemd is also responsible for the init and various other things. When checking the current CVEs 13 weaknesses exists for systemd overall, however none can be found for just the specific isolation part of services.

Bubblewrap is a small and new technique, so it does not have as many CVEs as systemd. There exists two CVEs as of this day (2017-03-01), one that is a local privilege escalation that only occurs when Bubblewrap is installed with SUID or file capabilities (which it does not need to run) and the other has a problem with TIOCSTI ioctl (which is used to simulate terminal input) where the sandbox can add input to the controlling tty. This seems to still be a factor when writing this text, but there are workarounds for this problem and that is to use seccomp to filter TIOCSTI ioctl or using a new command that is called –new-session (added in v 0.17). The former problem was fixed in version 0.13, so the usage of newer versions is encouraged. The tests were made on version 0.16 since version 0.17 was released during the actual tests and there was no time to update the firmware with the later version.

Firejail is actually affected by a similar problem as Bubblewrap that involves TIOCSTI ioctl and since Firejail requires root it becomes more severe than for Bubblewrap since an attacker can execute arbitrary commands through this security hole that is going to be executed outside the sandbox (CVE-2016-9016). This problem is mainly for version 0.9.38.4 and should be solved in later versions. There exist two more CVEs (CVE-2017-5180 and CVE-2017-5940) of which one is not fixed as of today (2017-03-07). The fix for the first CVE actually lead to the second CVE because of an incomplete fix. The first CVE was regarding a local code-execution vulnerability where a local attacker could execute their own code in within the sandbox context and can lead to denial of service attacks. The fix for this CVE lead to that an attacker could escalate their privileges and as stated is still not fixed. This CVE was released one month before this evaluation was written which one could argue is a long time ago and should be fixed and thus can be seen as a negative thing with Firejail.

The two other isolation techniques that are running on the embedded system is LSMs which was introduced to the kernel in 2008. Even though they were introduced long ago there only exists one CVE for TOMOY (CVE-2011-2518 and has to do with a mount system call not filtering the input which could allow a denial of service attack, and is solved since 2011 with update 2.6.39.2) and not a single one for SMACK. This could be due to various factors, as for example that few people are using these techniques, few people are actively trying to find weaknesses or because of the factor that an LSM is only an added layer of security (which means there is only one more hook before the call is being passed through and the possible errors could be fewer than with a container). Whatever the reason there is not enough substance to make the result only dependent on that there does not exist CVEs.

6.5 Licenses

As can be seen in the matrices 5.1, 5.2 and 5.3 the licenses for the various techniques is GNU GPL, GNU LGPL and Apache V2. All of these licenses can be used with Axis software and are being used today by many of Axis applications.

Chapter 7
Discussion

7.1 Results

The results are based on many iterations, for example in the nbench case the results are based on tests executed several times to obtain statistically meaningful results (and before the actual tests there is a calibration phase) and in the case of dbench the tests are made on roughly 500 000 files which also will provide a statistically meaningful result. That being said, the nbench results are roughly equivalent which should mean that the running techniques do not limit the CPU, caches or FPU of the system. This seems correct since the underlying techniques used in the kernel to achieve isolation runs the calculations on the hardware directly without any hardware virtualization. However the results regarding throughput of disk usage differs quite a bit and the reason for this could be that a lot of system calls to the kernel is being made frequently like creating new files, reading files and so on. The kernel needs to check that the process is allowed to make all the calls and what results that should be reported back (in case of virtual file system for example)

Since the same version of the base firmware was used for all the tests, just the specific isolation technique was added so only small differences in the result was expected. But as can be seen in Figures 6.1-6.10, the results differ and even with isolation activated some benchmarks got better results than a clean firmware. The results from dbench could perhaps be explained by the fact that Firejail and Bubblewrap limits the access to the file system so that the dbench tests just gets "no access" when trying to read/write to files on various locations and does not print an error message about this. In theory the "clean" version should have the highest result. We did try to get some error messages about this in dbench, but it seems like it does not support printing out this kind of information, and in general dbench is possibly not meant to test out the I/O of sandboxes so these kinds of errors may not have been thought of during development of the software. Even though we turned off most of the running services some of the still running services could of course vary the system load, which could explain the varying dbench results. But we did the measurements several times, on top of the benchmarking tools itself that iterates several thousand times, and still got similar results as seen in the figures.

SMACK is the one technique that varied vastly compared to the other tech-

niques when benchmarking the file system throughput. When looking at other embedded systems using SMACK degrades of 12% in performance for file system actions was noted [67]. This is not near as much as the performance loss we measured and reason for this big difference is not clear, the policy activated for the benchmark was set to allow all. UBIFS which is the file system used on the camera uses another compression technique than the tar.gz that we have taken numbers from and compared in figure 6.11. Since compression is used this might not give an exactly equal measurement but a good and close enough comparison between the different techniques use of disc space. Perhaps the underlying file system, UBIFS, has not been taken into account when optimizing for performance on SMACK or some parts with the specific ARM architecture differs that much so caching or other related parts to file handling is changed which SMACK has not been developed for.

7.2 Fault sources, possible solutions

These techniques are sometimes not meant or designed for the embedded world, which means that in some cases they can be very blunt when used for an embedded system. They can require way too much space or presume that the processing, memory or disk space is rather unlimited. Even if some are meant for embedded systems there is no guarantee that it is designed for our system with a single core processor based on ARM architecture. We recommend that tests should be done on more architectures before the performance aspect can be finally put to rest.

LXC and systemd-nspawn has the same basic problem, that there is no feasible way to unpack a container file system. For the future regarding these techniques we recommend some more work as to how to either make it possible to unpack or transfer a complete container file system to the camera. The rsync version we used is not supportive of all the special files that had to be moved. Another solution could possibly be to find a way to include it during build phase, but these are solutions that we could not make work or had the time to fully implement.

Since **Firejail, Systemd units and Bubblewrap** are running the future work could focus on generating profiles/isolation for them to automatically depend on the manifests for the running applications.

Docker is as stated dependent on various levels of middleware software, and will possibly be able to run without having to include the entire Docker Engine. Libcontainer is a possible way to go henceforth and first of all try to add all the various recipes one by one to existing layers instead of trying to add whole layers. We tried to add some of the recipes as singles, but realized there were some major requirements and cross dependencies so it is not certain that this is possible. This also raises the question whether adding single recipes that have to be maintained (and in this case we are talking about at least 40-50 recipes) by Axis is an applicable strategy. There then has to be people assigned to just maintain the functionality of Dockers core functions, which can be debated already is being done through the upstream existing layers containing this technique. Our opinion is that there is too much work to do this and that this solution should be avoided if possible and

instead rely on the complete layers (if and when the cameras hardware reaches the point where Docker can be supported).

Rkt has support for ARMv8 and could possibly be ported to support ARMv7-processors but we cannot say how much work this would require. Even though it possibly could be ported the question whether it can be used is still there as to the size of the download for x64 is roughly 100Mb and it contains various features that can be seen as overhead for this specific purpose. Our opinion and recommendation is to not pursue with Rkt for embedded systems based on the overwhelming features it provides and also because of the few architectures it does support.

Even if it is possible to use **AppArmor** without using the user space tools no new profiles could be applied without flashing a new firmware. This limits the use to only isolate applications included in the firmware or known to be installed in a later stage. Our recommendation is to find a solution to add the user space tools needed to control AppArmor so that if and when a third party software is installed limitations can be set to that application. The technique itself is reliable and widely used so the support and development will most likely continue.

SELinux is still in our opinion a candidate worth exploring since it is used with Android and thus has the possibility of working in an embedded system. However the camera model in our use case is very limited compared to the current cell phones when looking at flash memory and RAM. As stated before SELinux is a rather tricky technique to use and requires a massive amount of understanding before it can be put in to use, which could be a limiting factor. During the build phase only needed and used policies must be included in the firmware in order to reduce the space usage, other than that it should be possible to use the existing tools to generate policies for running software to set proper limitations.

SMACK lacked specially in the area of throughput in the sense of input and output so to be able to look at why this is such a limiting factor would be a good place to start. There could of course have been something wrong with the polices made for SMACK when running our benchmark which accidentally limited throughput, however it is not very likely since the policies were made to allow access to basically everything and consisted of very few lines of rules. Since SMACK is used in other embedded devices we think it is a good candidate to do further work on if the performance aspect can be solved.

Some concerns regarding **TOMOYO**s learning mode was raised during the learning phase, with it not generating the policy and just remained empty as if there was no system calls being made by the programs. The reason for this is unclear since the TOMOYO was activated and enabled and when manually adding the system calls to the polices they worked as destined. The learning mode has to be usable and the generation of polices automatical otherwise it is too time consuming to write all the policies manually. We believe that the future work of TOMOYO in Axis case is to find a fix for the lack of learning mode so that it works since the performance is quite good and the user tools already exists without having to add more layers to the build.

Chapter 8

Conclusions

8.1 Recommendation

When seeing the pure test results the implementations performed roughly equivalent on all but one test, the I/O-performance. Since SMACK decreased the performance of I/O-operations to roughly 10 percent of the normal it will be discarded because of that reason. When looking at systemd unit there are some security concerns about systemd in general [68], and when only using the services as an addon security patch some of the services must be executed as root and thus if broken can result in an adversary gaining root access. The configuration is however simple, and can be used by just adding support for namespaces in the kernel. Firejail also has the "problem" of needing root to be run, which just as systemd unit leads to the fact that if the sandbox is broken the adversary gains root access to the system. If the extra functionality Firejail provides compared to Bubblewrap is needed and the known risks regarding root requirements is worth it Firejail would also be a suitable and recommended implementation. If the implementations should avoid root to any cause it leaves just two techniques - TOMOYO and Bubblewrap. If the fix for TOMOYOs learning mode is an easy sustainable fix this is a really good candidate since the user space tools already exist within Axis layers for BitBake. Bubblewrap could be added with just support for namespaces and a small recipe, which makes it fairly easy to maintain. It achieves the isolation well and the configuration for each application is fairly flexible. The use of an LSM opposed to a container can of course be further evaluated, but both techniques provide the required safety measures while still being small and simple enough to not be a burden to maintain.

8.2 Future work

This thesis was meant to do an exploratory study of which techniques that were suitable for embedded systems with the limitations they provide. When some techniques had been rolled out of the decision due to various factors the benchmarking could begin, and our hope was that this dissertation could point further work in the correct direction straight away and not focus so much on the basics but instead work more in depth. Axis allows third party applications on their cameras and in the manifest for each application there could be a section that described how and

what they will require from the system. The natural next step is in our opinion to extend the study to try to make an automated sandboxing environment based upon these manifests that denies anything outside the scope of the manifest and does so by using Bubblewrap or TOMOYO.

References

[1] Sam Lucero et. al. Iot platforms: enabling the internet of things. Technical report, IHS, 03 2016.

[2] Leonard J. Bell, D. E. ; LaPadula. *Secure Computer Systems: Mathematical Foundations*. MITRE, nov 1973.

[3] H. Lindqvist. *Mandatory Access Control*. Umeå University, may 2006.

[4] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[5] An introduction to virtualization. `https://www.infoq.com/articles/virtualization-intro`. Accessed: 2016-11-16.

[6] Randi Thomas Ian Goldberg, David Wagner and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Sixth USENIX UNIX Security Symposium*, Jul 1996.

[7] Linux sandboxing. `https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md`. Accessed: 2017-03-14.

[8] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, Jun 2001.

[9] P Rubens. What are containers and why do you need them? `http://www.cio.com/article/2924995/enterprise-software/what-are-containers-and-why-do-you-need-them.html`. Accessed: 2017-02-27.

[10] Open container initiative runtime specification. `https://github.com/opencontainers/runtime-spec/releases/download/v1.0.0-rc5/oci-runtime-spec-v1.0.0-rc5.pdf`. Accessed: 2017-03-09.

[11] Linux security modules: General security hooks for linux. `http://www.hep.by/gnu/kernel/lsm/`. Accessed: 2016-11-17.

[12] et. al. C Wright. Linux security module framework. `http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf`. Accessed: 2017-02-27.

[13] S. E. Hallyn and A. G. Morgan. Linux capabilities: making them work. https://landley.net/kdocs/mirror/ols2008v1.pdf#page=163. Accessed: 2017-02-27.

[14] bpf(2) - linux manual page. http://man7.org/linux/man-pages/man2/bpf.2.html, oct 2016. Accessed: 2016-11-15.

[15] seccomp(2) - linux manual page. http://man7.org/linux/man-pages/man2/seccomp.2.html, oct 2016. Accessed: 2016-11-15.

[16] Bpf – in-kernel virtual machine. http://events.linuxfoundation.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf, feb 2015. Accessed: 2016-11-15.

[17] Alexei Starovoitov Jay Schulist, Daniel Borkmann. Linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt, oct 2016. Accessed: 2016-11-15.

[18] Linux 2 6 12. https://kernelnewbies.org/Linux_2_6_12. Accessed: 2016-11-29.

[19] Linux 3.5. https://kernelnewbies.org/Linux_3.5. Accessed: 2016-11-29.

[20] A seccomp overview. Accessed: 2016-11-16.

[21] Linux 2 6 24. https://kernelnewbies.org/Linux_2_6_24. Accessed: 2016-12-06.

[22] M. Ridwan. Namespaces tutorial: Isolate your linux system. https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces. Accessed: 2016-11-17.

[23] namespaces(7) - linux manual page. Accessed: 2016-11-17.

[24] S. Niu. et al. Overview of linux vulnerabilities. In *International Conference on on Soft Computing in Information Communication Technology*, 2014.

[25] 5 security concerns when using docker - o'reilly media. https://www.oreilly.com/ideas/five-security-concerns-when-using-docker. Accessed: 2017-03-16.

[26] A. Bettany. Vulnerability exploiation in docker container environments. https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments.pdf. Accessed: 2017-02-27.

[27] Cve - about cve. https://cve.mitre.org/about/. Accessed: 2017-03-08.

[28] Top open source licenses. https://www.blackducksoftware.com/top-open-source-licenses. Accessed: 2016-12-14.

[29] Free Software Foundation. Gnu lesser general public license v2.1 - gnu project - free software foundation. https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html. Accessed: 2016-11-22.

[30] The Apache Software Foundation. Apache licence, version 2.0. https://www.apache.org/licenses/LICENSE-2.0. Accessed: 2016-11-22.

[31] Nbench-byte results. http://www.math.cmu.edu/~florin/bench-32-64/nbench/. Accessed: 2017-02-21.

[32] dbench(1) - linux manual pages. https://linux.die.net/man/1/dbench. Accessed: 2017-02-21.

[33] Dbench. https://dbench.samba.org/. Accessed: 2017-02-21.

[34] Et. al G. Papaux. Processor virtualization on embedded linux systems, 2014.

[35] P. B. Menage. Adding generic process containers to the linux kernel.

[36] Ye Li, Richard West, and Eric Missimer. A virtualized separation kernel for mixed criticality systems. *SIGPLAN Not.*, 49(7):201–212, March 2014.

[37] Et. al M.S.U. Haq. Design and implementation of sandbox technique for isolated applications, 2016.

[38] Et. al W. Kanda. Spumone: Lightweight cpu virtualization layer for embedded systems. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 144–151, Dec 2008.

[39] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[40] Axis m1065-l network camera. https://www.axis.com/se/sv/products/axis-m1065-l. Accessed: 2017-03-07.

[41] Axis camera management. http://www.axis.com/global/en/products/axis-camera-management/overview. Accessed: 2017-01-19.

[42] Axis camera application platform. http://www.axis.com/ng/en/support/developer-support/axis-camera-application-platform. Accessed: 2016-12-03.

[43] Yocto project | open source embedded linux build system, package metadata and sdk generator. https://www.yoctoproject.org/. Accessed: 2016-12-12.

[44] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Embedded Systems. Springer Netherlands, 2010.

[45] Erik Karlsson. Evaluation of linux security frameworks. http://www8.cs.umu.se/education/examina/Rapporter/ErikKarlsson.pdf, 2010.

[46] Linux 2 6 36. https://kernelnewbies.org/Linux_2_6_36. Accessed: 2016-11-17.

[47] Apparmor core policy reference. http://wiki.apparmor.net/index.php/AppArmor_Core_Policy_Reference. Accessed: 2016-11-17.

[48] Apparmor - ubuntu wiki. https://wiki.ubuntu.com/AppArmor. Accessed: 2016-11-17.

[49] Linux 2 6 25. https://kernelnewbies.org/Linux_2_6_25. Accessed: 2016-11-18.

[50] C. Schaufler. Smack in embedded computing. In *Proceedings of the Linux Symposium*, Jul 2008.

[51] Jaroslav Šoltỳs. Linux kernel 2.6 documentation. http://diplomovka.sme.sk/zdroj/2847.pdf, 2006.

[52] Tomoyo linux functionality comparison table. http://tomoyo.osdn.jp/comparison.html.en. Accessed: 2016-11-30.

[53] Jonathan Corbet. Tomoyo linux and pathname-based security. https://lwn.net/Articles/277833/, 2008.

[54] Chapter 2: Why do i need tomoyo linux? http://tomoyo.osdn.jp/2.5/chapter-2.html.en. Accessed: 2016-11-30.

[55] Whatis - pukiwiki. http://tomoyo.osdn.jp/wiki-e/index.php?WhatIs#w3a12f9d. Accessed: 2016-12-13.

[56] Linux containers - lxc - introduction. https://linuxcontainers.org/lxc/introduction/. Accessed: 2016-11-21.

[57] Linux containers - lxc - security. https://linuxcontainers.org/lxc/security/. Accessed: 2016-11-21.

[58] Firejail. Firejail | security sandbox. https://firejail.wordpress.com/. Accessed: 2016-11-17.

[59] Google. Public dns | google developers. https://developers.google.com/speed/public-dns/. Accessed: 2016-11-17.

[60] Github - projectatomic/bubblewrap: Unpriviledged sandboxing tool. https://github.com/projectatomic/bubblewrap. Accessed: 2016-11-17.

[61] Alexander Larsson. Using bubblewrap in xdg-app - alexander larsson. https://blogs.gnome.org/alexl/2016/04/29/using-bubblewrap-in-xdg-app/. Accessed: 2016-11-17.

[62] Docker security - docker. https://docs.docker.com/engine/security/security/. Accessed: 2016-12-14.

[63] J. M. Harris. Container technologies in fedora: systemd-nspawn - fedora magazin. https://fedoramagazine.org/container-technologies-fedora-systemd-nspawn/. Accessed: 2016-11-21.

[64] systemd-nspawn(1). http://manpages.ubuntu.com/manpages/xenial/man1/systemd-nspawn.1.html. Accessed: 2017-01-23.

[65] coreos/rkt: rkt is a pod-native container engine for linux. it is composable, secure and built on standards. https://github.com/coreos/rkt. Accessed: 2017-02-23.

[66] 01org/meta-intel-iot-security: A collection of loosely related openembedded layers providing several security technologies. https://github.com/01org/meta-intel-iot-security. Accessed: 2017-02-21.

[67] Embedded alley solutions, inc. smack for digital tv. http://www.webcitation.org/6As4D4a0R. Accessed: 2017-03-14.

[68] Argumentation against systemd - without systemd. http://without-systemd.org/wiki/index.php/Arguments_against_systemd. Accessed: 2017-03-07.

Appendix A
Kernel flags

Listed below is the additional kernel flags (needed on the system described in Chapter 3) for each technique to fully function.

A.1 Firejail, Bubblewrap

```
CONFIG_NAMESPACES=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_NET_NS=y
```

A.2 TOMOYO

```
CONFIG_SECURITY=y
CONFIG_SECURITY_TOMOYO=y
CONFIG_DEFAULT_SECURITY_TOMOYO=y
CONFIG_DEFAULT_SECURITY="tomoyo"
CONFIG_SECURITY_TOMOYO_MAX_ACCEPT_ENTRY=2048
CONFIG_SECURITY_TOMOYO_MAX_AUDIT_LOG=1024
CONFIG_SECURITY_TOMOYO_OMIT_USERSPACE_LOADER=n
CONFIG_SECURITY_TOMOYO_POLICY_LOADER="/usr/bin/tomoyo-init"
CONFIG_SECURITY_TOMOYO_ACTIVATION_TRIGGER="/usr/lib/systemd/systemd"
CONFIG_NAMESPACES=n
CONFIG_NETLABEL=y
CONFIG_IMA=n
CONFIG_EVM=y
CONFIG_INTEGRITY=n
CONFIG_INTEGRITY_SIGNATURE=n
CONFIG_INTEGRITY_AUDIT=n
CONFIG_SECURITY_SELINUX=n
CONFIG_SECURITY_SMACK=n
CONFIG_SECURITY_APPARMOR=n
CONFIG_SECURITY_YAMA=n
```

```
CONFIG_IP_NF_SECURITY=y
CONFIG_IP6_NF_SECURITY=y
```

A.3 AppArmor

```
CONFIG_KEYS=y
CONFIG_DEFAULT_SECURITY_APPARMOR=y
CONFIG_SECURITY=y
CONFIG_SECURITYFS=y
CONFIG_SECURITY_APPARMOR=y
CONFIG_DEFAULT_SECURITY="apparmor"
CONFIG_SECURITY_APPARMOR_BOOTPARAM_VALUE=1
CONFIG_SECURITY_NETWORK=y
CONFIG_SECURITY_APPARMOR_HASH=y
CONFIG_NETLABEL=y
CONFIG_IP_NF_SECURITY=y
CONFIG_IP6_NF_SECURITY=y
CONFIG_SECURITY_SELINUX=n
CONFIG_SECURITY_SMACK=n
CONFIG_SECURITY_TOMOYO=n
CONFIG_SECURITY_YAMA=n
CONFIG_INTEGRITY=y
CONFIG_INTEGRITY_SIGNATURE=y
CONFIG_INTEGRITY_AUDIT=y
CONFIG_IMA=n
CONFIG_EVM=n
CONFIG_TCG_TIS_I2C_ATMEL=y
CONFIG_TCG_TIS_I2C_INFINEON=y
CONFIG_TCG_TIS_I2C_NUVOTON=y
CONFIG_TCG_ATMEL=y
CONFIG_TCG_TIS_ST33ZP24=y
CONFIG_SYSTEM_TRUSTED_KEYRING=n
CONFIG_TRUSTED_KEYS=n
CONFIG_INTEGRITY_ASYMMETRIC_KEYS=n
CONFIG_EVM_ATTR_FSUUID=y
CONFIG_TCG_TIS_ST33ZP24_I2C=y
CONFIG_TCG_TIS_ST33ZP24_SPI=y
CONFIG_PKCS7_MESSAGE_PARSER=y
CONFIG_PKCS7_TEST_KEY=y
CONFIG_SIGNED_PE_FILE_VERIFICATION=y
```

A.4 SMACK

```
CONFIG_NAMESPACES=n
CONFIG_SECURITY_PATH=y
CONFIG_DEFAULT_SECURITY="smack"
```

```
CONFIG_DEFAULT_SECURITY_SMACK=y
CONFIG_SECURITY_SMACK_BRINGUP=y
CONFIG_IP_NF_SECURITY=m
CONFIG_IP6_NF_SECURITY=m
CONFIG_SECURITY=y
CONFIG_SECURITY_SMACK=y
CONFIG_SECURITY_SELINUX=n
CONFIG_SECURITY_APPARMOR=n
CONFIG_SECURITY_TOMOYO=n
CONFIG_SECURITY_YAMA=n
CONFIG_TMPFS_XATTR=y
CONFIG_INTEGRITY=y
CONFIG_INTEGRITY_SIGNATURE=y
CONFIG_INTEGRITY_AUDIT=y
CONFIG_IMA=n
CONFIG_EVM=n
CONFIG_INTEGRITY_ASYMMETRIC_KEYS=n
CONFIG_PKCS7_MESSAGE_PARSER=y
CONFIG_PKCS7_TEST_KEY=y
CONFIG_SIGNED_PE_FILE_VERIFICATION=y
CONFIG_SYSTEM_TRUSTED_KEYRING=n
CONFIG_TRUSTED_KEYS=n
CONFIG_SYSTEM_TRUSTED_KEYS=n
```

A.5 SELinux

```
CONFIG_NAMESPACES=n
CONFIG_AUDIT=y
CONFIG_AUDITSYSCALL=y
CONFIG_AUDIT_WATCH=y
CONFIG_AUDIT_TREE=y
CONFIG_NETLABEL=y
CONFIG_IP_NF_SECURITY=m
CONFIG_IP6_NF_SECURITY=m
CONFIG_NETFILTER_XT_TARGET_AUDIT=m
CONFIG_NETFILTER_XT_TARGET_SECMARK=n
CONFIG_FANOTIFY_ACCESS_PERMISSIONS=y
CONFIG_NFSD_V4_SECURITY_LABEL=y
CONFIG_SECURITY=y
CONFIG_SECURITY_NETWORK=y
CONFIG_SECURITY_PATH=y
CONFIG_LSM_MMAP_MIN_ADDR=65536
CONFIG_SECURITY_SELINUX=y
CONFIG_SECURITY_SELINUX_BOOTPARAM=y
CONFIG_SECURITY_SELINUX_DISABLE=y
CONFIG_SECURITY_SELINUX_DEVELOP=y
```

```
CONFIG_SECURITY_SELINUX_BOOTPARAM_VALUE=1
CONFIG_SECURITY_SELINUX_CHECKREQPROT_VALUE=1
CONFIG_SECURITY_SELINUX_ENABLE_SECMARK_DEFAULT=y
CONFIG_SECURITY_SELINUX_AVC_STATS=y
CONFIG_DEFAULT_SECURITY_SELINUX=y
CONFIG_DEFAULT_SECURITY="selinux"
CONFIG_SECURITY_SELINUX_POLICYDB_VERSION_MAX=n
CONFIG_INTEGRITY=n
CONFIG_SECURITY_SMACK=n
CONFIG_SECURITY_APPARMOR=n
CONFIG_SECURITY_TOMOYO=n
CONFIG_SECURITY_YAMA=n
```

A.6 LXC

```
DEVPTS_MULTIPLE_INSTANCES=y
CONFIG_CPUSETS=y
CONFIG_PROC_PID_CPUSET=n
CONFIG_CGROUP_DEVICE=y
CONFIG_VETH=y
CONFIG_MACVLAN=y
CONFIG_MACVTAP=n
CONFIG_MEMCG=y
CONFIG_CGROUP_MEM_RES_CTLR=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
```

A.7 nspawn

```
CONFIG_NAMESPACES=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_NET_NS=y
CONFIG_UNIX98_PTYS=y
CONFIG_INPUT_LEDS=n
CONFIG_INPUT_FF_MEMLESS=y
CONFIG_INPUT_POLLDEV=y
CONFIG_INPUT_SPARSEKMAP=y
CONFIG_INPUT_MATRIXKMAP=y
CONFIG_INPUT_MOUSEDEV=n
CONFIG_INPUT_JOYDEV=n
CONFIG_INPUT_EVDEV=n
```

```
CONFIG_INPUT_EVBUG=n
CONFIG_INPUT_KEYBOARD=n
CONFIG_INPUT_MOUSE=n
CONFIG_INPUT_JOYSTICK=n
CONFIG_INPUT_TABLET=n
CONFIG_INPUT_TOUCHSCREEN=n
CONFIG_INPUT_MISC=n
CONFIG_VT_CONSOLE=y
CONFIG_VT_HW_CONSOLE_BINDING=y
CONFIG_SND_SOC_CS42L56=n
CONFIG_I2C_HID=n
```