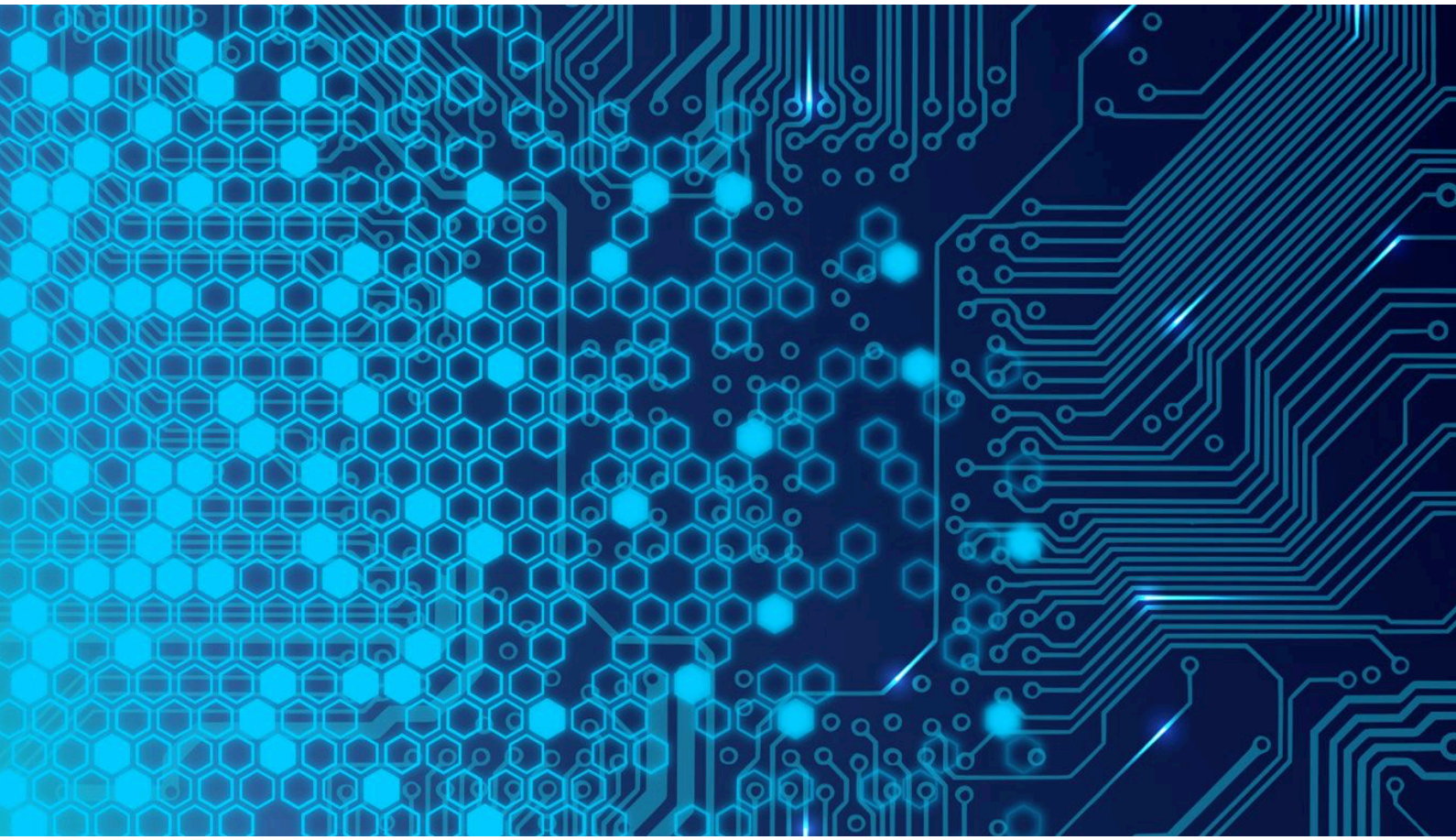
# INCREASING AVAILABILITY OF THE AEPU BY IMPROVING THE UPDATE PROCESS

MASTER'S THESIS

M.H. (Maikel) Coenen



July 2019

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
MSc Embedded Systems
Chair: Computer Architecture of Embedded Systems

Graduation committee
Drs. A. van Leeuwen
Dr.Ir. A.B.J. Kokkeler
Ir. E. Molenkamp
Dr.Ir. P.T. De Boer

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

nedap

# Acknowledgement

First of all, I would like to thank my supervisor Arthur van Leeuwen from Nedap Security Management, for his professional support and input. It was gratifying to have many meetings with him discussing useful insights.

I want to thank my supervisors André Kokkeler and Bert Molenkamp from the Univerisity of Twente for their time and patience to listen each meeting to what I had done and what the problems were. Thanks for giving me space and freedom to shape my research and come up with my own solutions.

Besides my supervisors, I would like to thank Robert Krikke, Gerard Koskamp and Wouter Baks for their insights into the controller and the low-level functionality. Thanks for supporting me during this project and answering my questions about complicated stuff. Also thanks to other colleagues at Nedap for hosting and supporting me during this project.

Finally, I would like to thank my friends and family for their comments on earlier versions and the hours of watching soccer-games and movies to give me some distractions.

# Abstract

Nedap needs to improve the update process of the Access Control controller to make their product highly available with only seconds of downtime each update. The current update process uses a straightforward approach which downloads the update, checks the files, stops the application, overwrite all files and reboots into the new system. After a full reboot, the access control application starts including fetching all authorisations from the server and initialising all connected hardware. This results in at least 3 minutes of downtime which can increase to 23 minutes due to the number of authorisations and the complexity of the system.

This research aims to determine if the update process can be improved by implementing existing update techniques to update the kernel and filesystem within seconds and additionally add fail-safe measures to revert to the last working system in case of a failed update. The time measured as improvement indicator is the relative speed up between the downtime occurring to the access control software during an update. The downtime starts from the point in time the application is killed and stops when it is fully up and running again.

Based on the insights in the old update process and two Design Space Explorations to kernel update techniques and checkpoint and restore techniques, we propose a new update process. This new process implements a second partition to store the update, uses Kexec to load and execute a new kernel directly from the running one and uses CRIU to create a checkpoint of the access control application which can be restored after a reboot. Additionally, a watchdog is implemented to reset the device in case the update fails and reboot into the last working system by using the second partition.

By using the new update process, a kernel and file system update is performed with only seconds of downtime. After performing tests on a full system emulation tool a system update is performed with 13.8 seconds of downtime. Comparing this to the old update process results in a relative speed up of factor 5.6 to 11.

# Contents

# Acronyms

**ACaaS** Access Control as a Service

**AEbridge** AEOS bridge

**AEmon** AEOS monitor

**AEOS** Advanced Enabling Organic System

**AEpu** AEOS processing unit

**API** Application Programming Interface

**BLCR** Berkeley Lab Checkpoint/Restart

**CPU** Central Processing Unit

**CRIU** Checkpoint Restore In User space

**DMTCP** Distributed MultiThreaded Checkpointing

**DSE** Design Space Exploration

**DSU** Dynamic Software Updating

**DUSC** Dynamic Updating through Swapping of Classes

**EABI** Embedded Application Binary Interface

**FD** File Descriptor

**IPC** Inter-process Communication

**JVM** Java Virtual Machine

**LXC** Linux Containers

**MMU** Memory Management Unit

**MTCP** MultiThreaded Checkpointing

**OS** Operating System

**PID** Process Identifier

**PTY** Pseudo Terminal

**RAM** Random-access memory

**RCU** Read-Copy-Update

**scp** secure copy protocol

**TCP** Transmission Control Protocol

**VM** Virtual Machine

**VMA** Virtual Memory Area

**VMM** Virtual Machine Monitor

# Chapter 1

# Introduction

Nedap Security Management focusses more and more on Access Control as a Service (ACaaS). ACaaS removes most of the hardware on-premise and delivers the same service from the cloud. Benefits from this approach are fast provisioning of products and the ability to always be up to date and run the latest software. Apart from migrating from on-premise products to the cloud, the fast-growing number of global customers of Nedap Security Management is notable. These customers have multiple offices around the world with thousands of doors to secure and employees to authenticate. While availability is of importance for these customers, maintenance of the access control solution is a complex task. Combining the ACaaS trend and the increment of global customers, the demand to be highly available all the time is increasing. To provide a high degree of availability, the time a system is unable to provide its functionality, called downtime, must decrease. This most often consists of decreasing the time to recover from crashes and decreasing the time required to update the software of products.

Even though the availability is of importance for Nedap, the demand of updating has also increased due to the publicity of the hack of the Mifare Classic cards back in 2008 [56]. Scientists of the Radboud University performed reverse engineering to decode the full algorithm of the chips used in the cards and therefore became able to clone the cards or change the data on it. In the Netherlands alone, already 2 million of these cards were used for example by public transportation. Currently, public transportation cards are replaced, but world-wide hotels, police offices, companies, and government buildings are still easy to hack [27]. One of the reasons for not updating the old cards is the complexity of replacing them. Switching to newer card versions is not as easy as replacing all cards but adds the requirement to update the access control system to support the new cards.

To be highly available throughout the year while remaining secure by performing an update, the telecom industry introduced the five nines (99.999%) uptime requirement [1]. This requirement translates to hardware downtime of 5 minutes and 15 seconds a year. For software, the percentage is lower, namely 99.95%, which is 1 day, 19 hours and 48 seconds of downtime a year. Almost two days of downtime can be acceptable to telecom providers but for the global customers of Nedap Security Management with 24/7 activity, even a minute of downtime per year is too much. Their security must be continuously functional, and every second of downtime can endanger the security. Therefore every second of downtime needs to be organised and carried out carefully.

## 1.1 Problem description

One of the biggest challenges to increase the availability as stated before is decreasing the downtime of an update. Currently, the update process of the embedded controller of the Nedap Security Management takes minutes, and during the biggest part, the software is unavailable and performs no authorisations. Many customers resolve this by updating their controllers overnight. They can, for example, prepare it during the day by copying the update to the controllers and set everything up but delay the actual update to execute it during the night. For global customers with a lot of controllers and especially customers with 24/7 of activity, this is not feasible. To perform a successful update, these customers must plan the update with great care and arrange guards to secure the doors during an update. In the end, this involves high costs by planning, hiring guards and performing the update.

## 1.2 Background

This section presents more information about the solutions of Nedap Security Management to understand the fundamentals of this research. Section 1.2.1 explains the software-based security management platform introduced back in 2000. Part of the access control platform is the controller, an embedded hardware piece, responsible for authenticating people and subsequently grant access to areas. This research particularly focusses on the update process of the controller and therefore Section 1.2.2 presents more details about the controller.

### 1.2.1 Advanced Enabling Organic System (AEOS)

As a solution to access control, Nedap introduced AEOS as a modular hard- and software system. In recent years several features are added such as intrusion detection, vehicle identification, graphical alarm handler, video surveillance and locker management. AEOS consists of four hardware layers, which are all installed on-premise. Figure 1.1 depicts these layers.

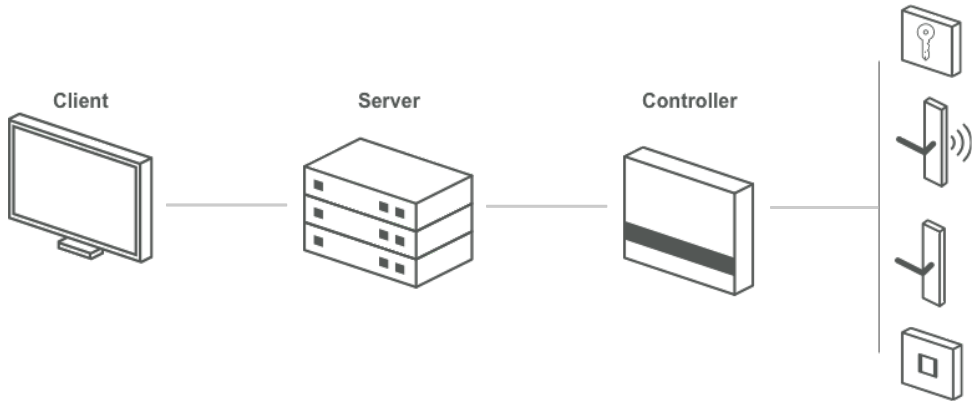



Figure 1.1: AEOS architecture

The first layer is the **client**, most often located at the reception of a building. Via the web-based application, the user can add or remove employees and visitors. For all these employees and visitors, the user can change or append information like telephone numbers, access control cards and authorisations. For example, the user can add an employee who is only authorised during the day and only to the office he or she is working. Besides the administrative tasks, the user can also view events such as an alarm or if someone is trying to access a room without the required credentials.

The second layer consists of the AEOS **server**. The server is located at the customer and acts as the hearth of the platform. On one side, it hosts the web-based application for the client, which the users can use as stated before. On the other side, it establishes a connection to all controllers to share authorisations and information. All the data from the client and controllers is stored securely in a database, and the server synchronises new authorisations frequently with all controllers.

The third layer contains the **controllers**, known as the AEOS processing unit (AEpu). The AEpu is an embedded device introduced to reduce the response time and to remain functional during network losses. This research focusses mainly on this embedded platform, and therefore more information about the AEpu is provided in Section 1.2.2.

The last layer contains all the **readers** connected to a controller. The controller is developed to be flexible and handle various kinds of readers. For example, it is possible to connect card readers with or without a keypad, fingerprint readers and palm-vein readers. Many other brands made their readers compatible to connect to the AEOS platform and are fully functional within the access control platform.

### 1.2.2 AEpu

The previous section presented the AEpu as the connection layer between the readers and the server. This section gives more insight into the newest version of the AEpu called Blue, see Figure 1.2. The controller integrates two hardware boards in one case:

1. *Control board:* The central processing unit which contains a Marvell Sheeva ARMv5TE microcontroller from the 88F6000 Kirkwood series. At least 256MB DDR2 Random-access memory (RAM), 16MB NOR flash and 2GB NAND flash are available on the control board. The actual specifications differ per customer.
2. *AEOS bridge (AEbridge):* The connection board to facilitate connections for connecting readers, locks and emergency buttons. Additionally, it provides an asynchronous serial point-to-point interface (RS-485 [40]) to communicate with other controllers. Using the RS-485 bus, one single AEpu can control multiple other controllers to extend the number of available connectors.


Figure 1.2: Blue AEpu

Each AEpu contains two input and two output connectors on the left side of the controller to control locks and buttons. Additionally, each controller can connect up to two readers with the connectors on the right side. With using the RS-485 connection and additional controllers, an AEpu can control up to 32 readers over a distance of 1200 meters.

Several software components are mandatory to boot the blue AEpu. The boot procedure starts with the bootloader. Currently, Das U-boot 1.1.4 is running on the AEpu. The bootloader initialises the hardware and starts the kernel. The kernel takes over control and functions as the communication layer between the user space and the hardware. On the AEpu the Linux kernel 2.6.34 is running. After the initialisation of data structures, drivers and devices, the kernel calls the initialisation scripts of the user space. The user space mounts file systems, sets up connections and starts services, such as the ssh-daemon and finally the Java access control software.

## 1.3 Goal

The goal is to research and develop an update process that decreases the downtime of the AEOS controller. The new update process must be fit for all controllers from the current hardware version up to future products. Because several other measures are mandatory before the new update process can run on the current controller, such as a modified bootloader and kernel, the proof of concept is implemented on a hardware emulation tool with the same processor architecture as the AEpu.

The main research question for this research is:

*Is it possible to combine existing update techniques to improve the controller's availability and perform a full device update within a contiguous application downtime of half a second?*

Four requirements are introduced to address this challenge and align with the expectations of Nedap:

1. Updating should involve as less programmer effort as necessary.
2. An update should not affect the current behaviour of running software.
3. The software must be open-source and highly supported.
4. The software must support at least the current version of the AEpu.

## 1.4 Scope

Up till now in literature, implementations are presented for each part of the update process individually. Studies propose interesting techniques but often focus on only updating the kernel or updating the file system. Additionally, most studies neglect the fail-safety; they propose a new update technique but do not implement recover mechanisms to resolve severe errors like a corrupt kernel. Given the urge to improve the availability of the product, it is of importance to improve the complete update process, including fail-safe measures to recover from crashes. Particularly, this research focusses on the update techniques of the controller software to address the challenges of updating the software within seconds. The new update approach should recover automatically from failures during the update process to ensure the controller remains functional.

A full controller update, including fail-safe measures and with minimal downtime is challenging. Due to the time constraints and the priority of Nedap Security Management to update the kernel and file system in coming releases to implement new features such as IPv6, this research focusses on a new fail-safe update approach of the kernel and file system with minimal downtime. This implies that a proof of concept for updating the Java application and bootloader is out of scope. Because the system does not continuously use the bootloader during runtime, it is possible to update it without downtime. Furthermore, the Java application requires more insights in the application to determine how to update it most efficiently. Application specific behaviour such as RAM-usage and flash usage is essential and further research is necessary before implementing an update solution.

As stated in Section 1.3, the new update process should be able to operate on the current newest controller, Blue. Due to the old kernel version which misses many valuable functions, a one-time update using the old update process must be performed to install required applications and scripts for the new process. Because a one-time old update is necessary after all, the new update process can be developed using new scripts and applications without dependencies to old applications. During this research, the least necessary kernel version is determined, required for implementation.

## 1.5 Report outline

The structure of the rest of this research is as follows. Chapter 2 provides insight into the update process for the controller currently used by Nedap. It introduces five phases required to perform a full update and gives details about what is executed each phase and its duration. In the end, it introduces the minimal and maximal downtime of the update process. Chapter 3 contains a literature research to relevant update techniques, similar to the solution proposed in this research. With the knowledge of the current update process and relevant techniques used in literature, Chapter 4 performs a Design Space Exploration (DSE) to kernel updating techniques. A DSE is a systematic analysis to research a specific system. Because the specifications and metrics of interest are often complex to deal with,

DSE uses a trade-off analysis between certain parameters, such as timing, resource usage and costs. The chapter first discusses several techniques, and finally, it compares these techniques using a trade-off table. Chapter 5 has a similar structure and performs a DSE to checkpoint/restore techniques to persist application states over a reboot. Based on the DSE of the kernel update techniques and the checkpoint/restore methods, Chapter 6 proposes the first steps towards a new update process. This chapter explains the options of the chosen methods and the required changes made to make the methods applicable to the AEOS controller. Chapter 7 subsequently introduces the additional phases of the new update process, such as fail-safe methods. After the explanation of the full new update process, Chapter 8 presents the results of the new update process in terms of a relative speed up. It analyses the timings of the phases which cause downtime of the old update process and compares it to the downtime phases of the new update process. Finally, Chapter 9 presents possibilities for future work and Chapter 10 finishes with the conclusion.

# Chapter 2

# Current update process

The first step to know how to improve the update process of the AEOS controllers is to examine the current process. This chapter provides the required insight into the current update process. First, it explains the different possibilities for an update and how to start it. After that, it introduces a visualisation of the current update process with five phases, whereafter it discusses each phase. Finally, this chapter concludes with the duration of the update process including the minimal and maximal downtime required for an update.

The current update process is a straightforward approach. The controller downloads the update files, checks them, overwrites the existing files and finally reboots the entire controller. The user can initiate this update process by using AEOS monitor (AEmon); a graphical application developed to configure all AEOS controllers inside a network. With the use of AEmon, users can configure different behaviours per door. For example, a system is set up with several doors connected to one single controller. To be able to use, for example, a reader with a touchpad on one door and a tag-reader on the other door, the controller needs configuration. All these configurations are constructed beforehand in AEmon and afterwards deployed to the controllers.

Additionally, using AEmon, users can view loggings, view reports and start an update. Currently, to update the AEpu several options are possible. The first option is whether to update all controllers or only groups or individual controllers. The second option enables the ability to update only parts of the system, such as libraries, the application or the complete system. With the third option, users can choose to upload the update files, upload and update on a specific time or perform the upload and update immediately.

When the user chooses the last option to perform the upload and update immediately, five phases are executed, depicted in Figure 2.1. In case a user decides to delay the update, the process halts after phase 1 and phase 2 starts after the delay. In the next part, each phase is examined to determine if an improvement in decreasing downtime is possible.

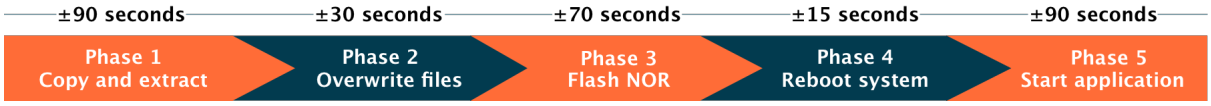



Figure 2.1: Current update phases

**Phase 1** starts with downloading all files from the computer running AEmon to the AEpu. Because validation of the files takes place before transferring, no extra integrity or security measures are implemented and downloading takes place using the secure copy protocol (scp). Transferring of the update files consists of three steps:

1. Copying the Linux system files to the update directory on the AEpu. When transferring is complete, the controller checks the archive with the use of MD5-hashes. After validating, AEmon initiates an

extraction to the update directory.

2. Copying the new Java runtime files as an archive. This archive is also validated and consequently extracted in the same update directory.
3. Copying the new Java application, validating and extracting to the update directory.

After phase 1, all update files are present on the controller, ready to overwrite the existing files. First, the AEOS Java application needs to be shut down. When the application is down, all connected doors are in a default state, which is dependent on the hardware used, normally open or normally closed.

**Phase 2** consists of copying new files over the old files. Before copying the new files, some clean-up takes place. The clean-up makes sure no dependency conflicts occur after an update. At least it deletes the zone information, the SQL library and all AEpu application files. After the clean-up, the new file tree is being copied over the existing root, overwriting all old files.

After setting up the new file system, **phase 3** takes care of updating the bootloader, bootloader configuration and the Linux kernel. It first checks the integrity using MD5-hashes. Additionally, a file on the controller contains all the hashes of previous updates to determine if the controller needs an update or not. After the checks, the corresponding NOR flash partition is erased and rewritten. Respectively, for the bootloader, bootloader configuration and kernel. When all writes are complete, the update process restores the certificates and updates the file permissions and encryptions.

To use the new bootloader and kernel, the system must reboot, illustrated in **phase 4**. An option exists to update only the application and dependencies. In that case, a full reboot is not necessary, and the AEpu application can start directly. A full reboot of the device takes approximately 15 seconds, depending on the initialisation of hardware in the kernel and starting of services in user space.

Whether the system has performed a reboot or not, the Java application needs a restart. **Phase 5** contains the full boot of the complete Java application. Several services are implemented to give flexibility to the software platform. For example, intrusion detection uses different services in comparison to locker management. The downside of this approach is the complexity of the initialisation phase. The 90 seconds depicted in phase 5 of the figure, contains a full reboot of a clean application without any configuration. When global customers with many authentications and complex configurations updates their controllers, a restart of the application can take up to 20 minutes.

With all phases analysed, the total downtime becomes visible. Calculating it consists of summing all the durations from phase 2 up to the end of phase 5. This results in a total downtime for a full controller update of 205 seconds. Because all durations are the minimal durations, this total downtime is also minimal. Using the maximum restart time of the Java application, the total downtime can take up to 23 minutes.

In summary, the current update approach consists of five phases, while four introduces downtime. Copying and overwriting (phase 2 and 3) takes 100 seconds, the reboot takes 15 seconds, and the reboot of the application takes at least 90 seconds. Conclusively, this takes at least 205 seconds of downtime to update the whole controller. This downtime can increase to 23 minutes due to the complexity of the configuration. The rest of this report contains the research to improve the current update phases by decreasing the downtime and adding fail safety.

# Chapter 3

# Relevant update systems

Most of the literature related to update techniques consists of a solution for one single problem, such as updating the kernel without a reboot. Nonetheless, for Nedap Security Management, all steps of the update are important and only combining the most promising solutions resolves the downtime issue. This chapter describes literature which implements a full solution to deliver a complete update process. First, it analyses two research articles which do implement a similar solution as the proposed solution of research. Secondly, it discusses OSs, which delivers a special feature to abstract the interface and implementations from each other to decrease downtime during updates. While this chapter mainly focusses on a complete update solution, Chapters 4, 5 and 9.1 focusses on a specific part of the update process.

## 3.1 KUP

Kashyap et al. [43] proposes an instant updating technique with the use of CRIU as checkpoint/restore mechanism and Kexec to update the kernel with a partial reboot. KUP consists of six stages responsible for the checkpoint and the restore process:

1. Checkpoint all running applications to restore after an update.
2. Store the checkpoint in persistent memory to fetch it after an update.
3. Switch from kernel and skip bootloader stage.
4. Boot the new kernel.
5. Initialise system services because no checkpoint is available for these services.
6. Restore all applications.

To prevent system failures, they implement a safe fallback method. Before switching the kernel, KUP also loads the old kernel into memory. If a fault occurs during an update on kernel level or application level, KUP is able to switch directly back to the old kernel. This method resolves issues due to a new kernel on both system level and application level, but it implements no measures to resolve issues due to failing checkpoints and restores. In case the checkpoint or restore of an application fails, the update continues, and KUP is not able to restore the application afterwards.

Additional to implementing checkpoint/restore and kernel execution techniques, the KUP system also optimises them by introducing two methods to optimise the use of Kexec and one method to optimise CRIU. The first method ensures that Kexec only starts a processor-core when it is required upon reboot. This method could save up to six seconds for systems with 80 processor-cores. For the AEOS controller with only one core, this optimisation is not beneficial. Secondly, KUP skips polling unused PCI slots during execution of Kexec, saving 8.5 seconds on a 16-core machine. To optimise CRIU, KUP uses persistent storage over reboot to decrease the fetching time. This technique does not store the checkpoints in

flash but keeps it in RAM which removes the time to transfer the data from flash to RAM before usage. The results of this optimisation method differ per size of checkpoint but can decrease the checkpoint and restore time by seconds.

By using Kexec and CRIU and implementing the optimisations, KUP can perform a full update in seconds, dependent on the running applications and system. Unfortunately, the authors never published the source code and therefore, no support is guaranteed, which was one of the requirements in Section 1.3. Nevertheless, analysing the methods used in their article is valuable and can provide insight into promising methods.

Contrary to the KUP system, the proposed solution in this research implements an update process for embedded architectures with minimal resources. Therefore no daemon is required to check for failures, and no second kernel is stored in memory to switch back. This results in less overhead and fewer memory requirements.

## 3.2 Seamless kernel update

Siniavine and Goel [58] proposes a solution based on the same principle of creating a checkpoint of an application and performing a kernel update whereafter it restores the checkpoint. To checkpoint and restore the application, they implement their own system, which saves data structures and resources of any application. For each resource, it saves the address and the entry point in the checkpoint to a *Save table*. If a resource already exists, it creates a pointer to the value of the hash table.

The system preserves the checkpoint in memory during a reboot by reserving memory pages during the boot process. This ensures that the boot process does not use these pages and the checkpoint can be used directly from memory after reboot. The restore process creates a *Restore table* and on each successful resource restore, it writes the corresponding identifier to it. Checking this table for completeness results in a successful restore with all resources. Besides the resources, the proposed system restores thread states, memory states, open files, sockets, pipes, Inter-Process Communications (IPCs) and terminals.

Because the focus of the authors lies in reinitialising user space applications, this system does not implement a new kernel update mechanism to decrease downtime. Instead, they perform a full reboot after creating the checkpoint. This full reboot ensures the system starts the new kernel, and after that, it restores the application. During the update process, no fail-safe measures are implemented, and because of the full reboot, the downtime is just above 10 seconds on a 3 GHz dual-core with 2GB of RAM.

Unlike the seamless kernel update method, the solution in this research does implement a kernel update technique to reduce the downtime further. Because one of the goals is to improve the availability of the controller, fail-safety is essential. The seamless kernel update technique does not implement any fail safety measures while this research does. A failure during an update using the seamless kernel update methods possibly bricks the system.

## 3.3 Migration OSs

Besides techniques to update existing modules as the bootloader, kernel and file system, the ability exists to design a new OS to make abstractions between interfaces and implementations. This adds the ability to replace components without disruptions. For example, exokernel designed by MIT enables the applications to communicate more easily with the hardware by using a microkernel [33]. Because it implements most of the hardware communication on application level, updating or swapping components is possible without a reboot. Other possibilities which use the same approach are Proteos [34], Barrelfish [13] and LibOS systems as Drawbridge [12].

Sprite [53] and LOCUS [63] are OSs specialised for migrations of processes over network setups. Additionally, MOSIX [11] proposes the same solution on library level. With the use of these techniques, processes can be moved from a source machine to a destination machine of the same architecture. Using a combination of a checkpoint/restore and a migration technique, the methods can migrate a process within milliseconds of downtime. After the migration, system calls are forwarded or redirected to the new system. For systems with a high availability requirement, these OSs deliver a practical solution.

However, unlike the OS techniques, the system proposed in this research does not modify the kernel or running system to enable faster updates. It optimises the current process by switching kernels without losing the state of applications. This implies less programmer involvement for updating and the ability to use the mainstream Linux kernel and application versions without modifications including the new updates.

# Chapter 4

# Kernel update methods

Regarding the update process explained in Chapter 2, a part of the downtime is due to the reboot of the system. This full reboot is mandatory to make use of the updated Linux kernel. This chapter first presents an insight into the functioning of the Linux kernel and the necessity of a reboot. After that, it explains two possibilities to update a kernel, whereafter several techniques are presented of both flavours. Finally, the approaches are compared on four criteria using a trade-off table.

The Linux kernel is the connecting layer between the hardware and the applications running in user space, Figure 4.1. The kernel currently used by the controller is monolithic, which means it is fully responsible for device drivers, file system, memory management, network stack and IPC in contradiction to micro-kernels which executes most functionality in user space. In monolithic kernels, an application makes use of system calls, to access the hardware. The kernel consists of the syscall interface, the generic kernel code and an architecture dependent layer. The kernel code is the same for all systems, independent of the underlying hardware. To support specific hardware, users can configure the architecture layer. This layer consists of drivers which can be replaced to communicate with the underlying hardware.

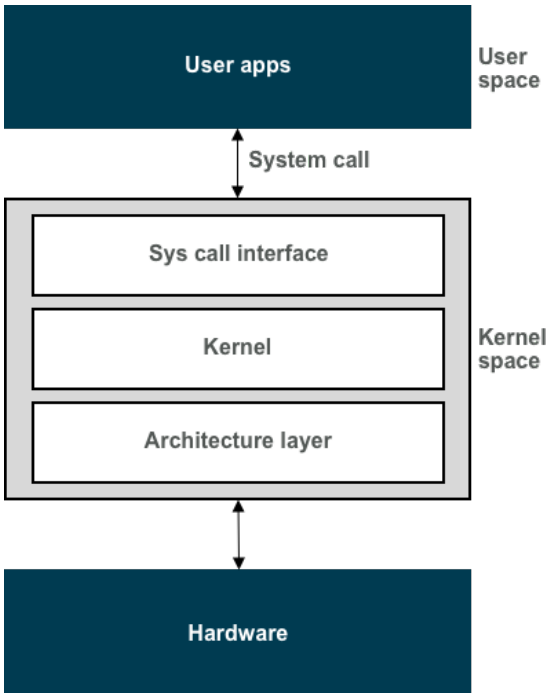



Figure 4.1: Linux kernel

Updating the Linux kernel interrupts the connection between the hardware and the user space. Because the file system, process management and memory management are part of the kernel and crucial for running applications, it is not possible to swap the kernel and continue running applications. Hence, to update the kernel, the system has to reboot and restart all applications.

Two flavours of kernel updating mechanisms are present in literature, patching and soft reboot mechanisms. Patching techniques are commonly used by desktop versions of Linux nowadays. For example, Canonical Livepatch for Ubuntu systems. This service delivers an approach to perform critical updates without requiring a reboot. In general, the patching software bundles the differences in source code and save it as a file called a patch. For a kernel patch, techniques combine these files into a kernel module which can be loaded while running. The patching software then does the actual update by redirecting calls from old functions to the functions from the new kernel module. The downside of this

approach is the increasing kernel size. Every kernel patch, the software adds a kernel module including the updated code to the existing code, resulting in a bigger kernel. Advantage of patching is the little downtime. Depending on the size of the patch, only microseconds of downtime is necessary.

Soft reboot techniques do not use kernel modules to add code, but the techniques are more like the conventional way of updating the kernel. These techniques perform a reboot but skip some non-crucial parts or reboot a virtual machine aside of the running one. Both are resulting in less downtime and a completely new kernel. The downside of some of these approaches is the resource usage of running two systems simultaneously. Skipping some phases of a boot can also be problematic because it skips checks and initialisations. The most significant advantage of using a completely new kernel instead of performing a patch is the smaller risk of crashes due to changing interfaces and systemcalls over updates. Besides, after an update, the kernel is running an exact copy compared to other systems whereby patching increases differences.

Coming sections explain the internals of several kernel update techniques of both flavours. To give a more comprehensive overview, Table A.1 of Appendix A presents a comparison which shows the features of each technique.

## 4.1 Ksplice

Ksplice is one of the oldest patching techniques for Linux kernels and was formerly open-source. Back in 2011 Oracle bought the complete source code of Ksplice and made it available for Premier Support Customers only [42]. Initially, the community developed Ksplice for x86 architectures, but they add ARM support from version 0.9.0 with a minimal version 2.6 of the Linux kernel [8].

According to Arnold and Kaashoek [7], Ksplice uses the object code of the kernels instead of the source code to create a patch. It uses two techniques to accomplish this: pre-post differencing and run-pre matching.

Pre-post differencing uses the original kernel source code and the patched code to build two working kernels. After that, it compares the object code and metadata of both kernels to extract all changed functions. Finally, pre-post differencing stores each function individually in object files which are combined to create a kernel module. The running kernel can load this module without interrupting running user space applications.

After loading the patch module, it is not functional yet. Ksplice first needs to resolve all memory addresses of the functions to swap. Run-pre matching takes care of this by comparing each byte of the running kernel to the module. Additionally, it checks if no unintentional changes take place by patching.

When finished resolving and checking the memory, it is ready to perform the actual update. To ensure no system calls are executed during updating, Ksplice uses `stop_machine`. This function captures all the available Central Processing Units (CPUs) and runs the Ksplice function on one single core. If the function is not able to capture all CPUs, Ksplice pauses and tries again after a delay. Because of this implementation, Ksplice is not able to patch functions which are always on the call stack within the kernel and therefore never inactive. As a result, it only supports 88% of the security patches from May 2005 to May 2008.

## 4.2 kGraft

In response to the acquisition of Ksplice by Oracle, Section 4.1, Linux distributions SUSE and Red Hat cooperated to develop an open-source alternative. In the end, this results in two almost identical approaches, kGraft and Kpatch. SUSE developed kGraft, and it is available from SUSE Enterprise 12 distributions, including kernel version 4.0. SUSE enterprise requires an 64-bit architecture of AMD, Intel, IBM or ARM [59].

SUSE [60] presents the internal functionality of the open-source application. In comparison to Ksplice, it uses the same technique to compare the running kernel and the patched kernel, resulting in a module with all changed functions. Switching from the running functions to the ones bundled in the module is possible due to compiling with function profiling enabled. This option allocates five bytes in front of each function containing a call instruction. After the patch starts, kGraft replaces the first byte with an `INT3` (breakpoint) instruction to provide atomicity for replacing the rest of the bytes. Then it uses `ftrace` to replace the other four bytes by the address to the new function. Finally, the first byte is replaced by the `JMP` instruction to call the new function instead of the old one.

kGraft applies an approach similar to Read-Copy-Update (RCU) and so-called trampolines, to prevent the kernel from crashing due to changes in function interfaces. The trampoline function is called on each kernel entry and checks if the kernel should use an old or new function. This decision is based on a per-thread flag to determine if the new function already can be used. After deciding, it jumps to the called kernel function. Conclusively, this makes sure an old function calls only old functions, and new functions only new ones. When all thread flags changed successfully to the new functions, patching is complete, and kGraft removes all flags and trampolines.

Because all CPUs remain operational and all applications can continue running, the downtime is negligible. Only the trampolines introduced to avoid kernel crashes introduce delays due to checks and jumps.

## 4.3 Kpatch

As a result of the cooperation of the SUSE and Red Hat communities, Red Hat introduces Kpatch available from Red Hat Enterprise Linux 7 with kernel version 4.0 [37]. Red Hat Enterprise Linux supports x86-64, IBM Power, IBM z systems and from version 7.4 it supports ARM64 architectures [38].

Because the community for development is almost identical to kGraft, the internal functionality is similar. According to Conference [18], creating the patch is entirely identical to kGraft, and therefore it compares the kernel code and creates a module with the changed functions.

The actual patching differs little and is a combination of the techniques used by Ksplice and kGraft. It uses `stop_machine` to halt all CPUs except the CPU which is running Kpatch. This technique guarantees that no system calls are possible during the patch. When all CPUs are stopped, it changes the addresses of the old functions to the addresses of the new functions included in the module. Similar to kGraft, it uses `ftrace` to change these addresses.

## 4.4 KernelCare

KernelCare is also a patching technique but differentiates from the previous techniques by offering their product as a service. The service of CloudLinux provides patches for all architectures [17] and is available from Linux kernel version 2.6.18. Because they deliver the full process as a service, CloudLinux supports all patches [9]. The additional effect of the service is the closed source integration with only the Linux kernel module as open-source code [16], resulting in almost no information about the internals. According to the available information, the service automatically downloads new kernel patches to the system and applies them with use of their kernel module.

The kernel module takes care of loading the patch into address space, handle relocations from old functions to new ones and making sure no system calls are executed during patching. Because all patches are developed and applied by CloudLinux, they can customise each patch to ensure support for all updates and all architectures.

## 4.5 Kexec

Relative to the previous patching techniques is Kexec a technique which enables a system to load and boot a new kernel directly from user space. This results in a full update of the kernel with a partial reboot. Initially, Kexec supports only x86 architectures and is available from mainline kernel version 2.6 onwards [54]. From 2007, the kernel adds the required configurations for the ARM architecture, and therefore Kexec also supports the ARM architectures. Normally, during boot, the bootloader loads the kernel, but Kexec skips this stage and directly executes the new kernel. Providing a fast reboot but introduces consequences which the user must take into account. With a full boot, the hardware initialisation resets all devices into a *sane* state. Because the initialisation is part of the bootloader stage, Kexec skips it, and therefore the user must take care of resetting devices.

This section summarises an overview of the internals of Kexec provided by Nellitheertha [50]. According to his information consists Kexec of two components. The first component is *kexec-tools*, which is the user space application to load the kernel and restart into it. The second component is a kernel module used by the user space component to perform the actual switch between kernels.

To actually load and restart into a new kernel directly from a running kernel, Kexec uses three stages:

1. Copy the new kernel into memory.
2. Move the kernel into dynamic kernel memory.
3. Copy to the final destination and start the new kernel.

Loading the kernel implements the first two stages, in which Kexec parses the input file and constructs the segments for each kernel part dependent on the architecture. For example, the ARM architecture uses two segments, one for the kernel and one for the device tree blob. Each segment consists of the addresses of the buffers in user space memory and kernel memory and their sizes. After parsing and constructing the segments, Kexec loads them into the user space memory. Thereafter, stage two executes the system call `sys_kexec` to copy the segments into the kernel pages and additionally allocates memory for the `reboot_code_buffer`.

The third stage is rebooting into the new kernel. To start this stage, the `sys_reboot` function is called with a special flag, `LINUX_REBOOT_CMD_KEXEC`. This ensures it transfers control to the function `machine_kexec`, which is dependent on the architecture used but in general, it stops all interrupts, loads the device tree, copies the assembly code to the allocated buffer and jumps to this code.

The assembly code finally copies the new kernel over the running kernel and jumps to the address of the new kernel forwarding the kernel option parameters given by the user.

## 4.6 ShadowReboot

A much newer reboot technique is ShadowReboot by Yamada and Kono [65]. The authors propose a solution to shorten the downtime of a kernel update by making use of Virtual Machines (VMs). The running system must support VMs which is available for ARM from mainline Linux kernel version 2.6.21 [25]. ShadowReboot is only experimentally tested on x86-64 systems, and therefore ARM support is not guaranteed.

ShadowReboot implements a Virtual Machine Monitor (VMM) to spawn a reboot-dedicated VM parallel to the running system. Since only the reboot-VM reboots, the applications on the original system can continue without interruption. When the reboot-VM is up and running again using the new kernel, it has to transfer the running applications from the original system to the VM.

Transferring the applications and their states from the original system to the VM is made possible by taking a snapshot of the system. This snapshot contains a complete file system, including the application states at the time of the snapshot. After the creation of a snapshot, ShadowReboot restores it into the rebooted VM. Resulting in an identical system running on a new kernel.

Because ShadowReboot does not shut down the applications during snapshot creation, states can change between creation of the snapshot and before the restore. The system does not transfer these changes to the VM and therefore are lost. The advantage of keeping the applications alive is the reduction of downtime. The downtime for applications consists of the VM fork and the restore time of a snapshot. ShadowReboot is tested on a Dell OptiPlex with a 3 GHz dual-core and 4 GB RAM, running five Linux distributions with kernel version 2.6.34. The authors varied the memory sizes of the machines to 256, 512, 1024, 2048 and 2560 MB. The results show a 96,6% shorter downtime in comparison with a normal reboot for 256 MB of memory. Overall the average downtime of a kernel update is about 5 seconds, varying from 1,9 to 9,86 seconds.

## 4.7 Dwarf

Dwarf is the newest technique which is introduced in 2018 by Terada and Yamada [62]. They propose an approach based on multiple VMs. Dwarf is experimental and tested on a Linux desktop from kernel version 2.6.39. The technique is experimental, and the authors did not test Dwarf on ARM architectures.

In comparison to ShadowReboot, Section 4.6, Dwarf loads the update into a new VM but does not take a snapshot of the running system but transfers the applications including memory pages and process files to the new VM. The Dwarf hypervisor is crucial for transferring the application. It spawns new VMs and takes care of copying and transferring control between the systems.

The VMs used for Dwarf only use virtual CPUs and memory. All other I/O devices are not virtualised but are in control of the hypervisor. When the hypervisor transfers control of the applications, it also de-attaches the I/O from the old VM and attaches it to the new machine.

Dwarf can only handle updates which are backwards compatible, and it is not able to update the structure of memory mappings because memory is transferred between machines and requires the same structure. Because Dwarf uses a hypervisor to switch between VMs, it is continuously active and updating the hypervisor itself is not considered in the literature. Hence, a full reboot is required. Dwarf results in an average downtime of 2.0 to 2.6 seconds by performing a full kernel update on a quad-core processor with 16 GB of memory.

## 4.8 Comparison

This section consists of a DSE of all techniques. First, it explains each criterion based on the requirements, and after that, a score is given to each requirement. Finally, it presents the results of the comparison with an explanation.

### 4.8.1 Criteria

A set of general requirements is given in Section 1.3. Based on the requirements, criteria on which the parts are analysed are set up. Each kernel update technique discussed in this chapter is compared according to these criteria. It is essential that the proper technique is chosen to elaborate further on in this research and later on in the proof of concept.

The comparison in this chapter uses the following criteria to find the best update technique for the new update process:

1. *Full update*: Performs the technique a full update or does it patch the running kernel?
2. *Open-source*: Is the proposed technique open-source?

3. *Kernel version*: Which kernel version is mandatory to use the technique?
4. *Hardware compatibility*: Is the technique able to run on the AEOS controller?

Each criterion has a specific weight based on its importance within the overall update approach. With the weight and the numerical interpretation of the signs in Table 4.1, the final score can be derived. For each technique, the score $s_i$ is multiplied by the associated weight $w_i$. Adding all these results together gives the final score of a technique, denoted by Equation 4.1.

Table 4.1: Scoring table

| **Sign** | + | 0 | - |
|---|---|---|---|
| **Score** | 1 | 0 | -1 |

$$\sum_{i=1}^{4} s_i * w_i \tag{4.1}$$

### 4.8.2 Results

Important to note is the limited resource availability in the controller during the update. Besides the physical devices, an update must be easy to roll out, in contrast, to adjust the update code to support a kernel update. Both results in a high weight of the criteria full update and hardware compatibility. Because patch techniques add kernel code each update, the limited flash storage can be a problem over several updates.

Additionally, patch techniques are not always able to perform a specific update. Programmer involvement is required to, for example, change variable types during runtime. Therefore a full update is more applicable for the controllers of AEOS. Implementation of VMs to switch between kernel versions is interesting for desktops and servers with high availability requirements. Because the controller consists of a single core processor, VMs have to share the processing time resulting in low performance. This eventually even increases the downtime tested by the authors of ShadowReboot and Dwarf.

The criteria about the kernel version and if the technique is open-source are less relevant and therefore weighted low. The kernel version is included to know which minimal kernel version is required for using the update technique. Because, as stated in Section 1.4, the assumption can be made that customers have to perform a one-time update with the current approach before the new approach is usable, it is less important for implementation.

Table 4.2 shows the trade-off table for the kernel update techniques. Next, the results are discussed.

Ksplice is a powerful technique for Oracle desktop and server users. The patching approach is beneficial for critical kernel updates, and with downtime less than a second, almost no interruption is noticeable.

Table 4.2: Trade-off kernel update techniques

| | Full update | Open-source | Kernel version | Hardware compatibility | **Total** |
|---|---|---|---|---|---|
| Weight | 3 | 1 | 1 | 3 | 8 |
| Ksplice | - | - | + (2.6) | + | 0 |
| kGraft | - | + | - (4.0) | - | -6 |
| Kpatch | - | + | - (4.0) | - | -6 |
| KernelCare | - | - | + (2.6) | + | 0 |
| Kexec | + | + | + (2.6) | + | 8 |
| ShadowReboot | + | + | + (2.6) | - | 2 |
| Dwarf | 0 | + | + (2.6) | - | -1 |

Because the usage is limited to Oracle Premier customers and patching is less valuable for embedded devices, Ksplice is in comparison to the other techniques less applicable for this research.

kGraft and Kpatch are similar but both available for their own Linux distribution. They are relatively new, and therefore at least kernel version 4.0 and a 64-bit architecture is required. Including the fact that they are both patching techniques, they are less applicable to run on an AEpu.

KernelCare is different in the provision of patches. It delivers its application as a service with excellent support. Therefore it can ensure to patch all critical updates available. Because it is a patching technique and it is a service product including the support advantages, it scores average in comparison to the other techniques.

Kexec scores maximally because of the open-source full update techniques used and a good fit for embedded devices. It implements an approach to fully update a kernel with a partial reboot. This ensures each update takes no extra memory and it can run on embedded devices with limited resources.

ShadowReboot is an approach using VMs to switch between systems and therefore use a newer kernel in only seconds of downtime. Because two systems have to run simultaneously, the controller of AEOS is too limited in processing power.

Dwarf is similar to ShadowReboot but scores less on the full update criterium because it is not able to update memory mappings. This limits the capabilities in which Dwarf is usable, and therefore, it scores less than ShadowReboot.

# Chapter 5

# Checkpoint and restore methods

Performing a full kernel update requires shutting down all the running applications. As Chapter 2 presents, the time required to start the Java application is significant. Checkpoint/restore techniques are interesting to subdue this problem. Creating a checkpoint consists of saving register set, address space, allocated resources and other process private data. These stored data can be restored after a reboot with the same state as before the checkpoint. The advantage of checkpoint/restore is, therefore, the ability to skip initial tasks and resume running at the same state as before the creation of the checkpoint.

Kadekodi [41] classifies current checkpoint/restore techniques by their scope. Depending on their level of operation, two classes can be defined: checkpoint/restore on application or system level.

An application-level checkpoint system checkpoints one specific application, and it can be adjusted to store parts of the application at a predefined timestamp. The downside of this approach is the need to rewrite the application to predefine when to store application states during execution of the application. The advantage is the ability to decide what to store when, resulting in a more efficient way of making a checkpoint.

The system-level approach creates checkpoints at OS level. It does not depend on a specific application, and the user can update the checkpoint application without affecting other applications. The downside is that a checkpoint does not take application specific details into account and therefore making a checkpoint can result in higher downtime and bigger files.

This chapter presents checkpoint/restore approaches based on application and system level. Each section explains the internals of an approach, and finally, Section 5.6 compares them.

## 5.1 DMTCP

Ansel, Arya, and Cooperman [2] propose DMTCP, which is a checkpoint/restore mechanism developed on application level. The application to checkpoint should be linked to the provided DMTCP library before use. The authors base the implementation on previous work called MultiThreaded Checkpointing (MTCP), which proposes a technique for making checkpoints of individual processes [3]. DMTCP adds the ability to checkpoint and restore socket and file descriptors, and other artefacts of distributed software.

Creating a checkpoint with DMTCP consists of sevens stages:

1. *Normal execution*: Wait until the checkpoint is requested.
2. *Suspend user threads*: Suspend all threads and save the owner of File Descriptor (FD).
3. *Elect FD leaders*: Elect the leader for each FD. With misusing the flag `F_SETOWN` of function `fcntl` the owner of an FD can be changed. All processes change the owner of the FD, and therefore, the last process wins the election.

4. *Drain kernel buffers*: The leader of each socket flushes the socket by reading until no more data is available. After that, it writes the connection information table to the disk.
5. *Write checkpoint to disk*: Save all user space memory to a checkpoint file.
6. *Refill kernel buffers*: Send back the drained data to the socket buffers.
7. *Resume user threads*: All applications can resume their execution because the checkpoint process is finished.

To perform a restore, also seven stages are required:

1. *Reopen files and recreate Pseudo Terminal (PTY)*: Reopen all FDs, excluding socket connections to remote processes.
2. *Recreate and reconnect sockets*: Find the new address of the process and after that re-establish the sockets.
3. *Fork processes*: The restore application forks into $N$ processes, where $N$ is the number of processes to restore.
4. *Rearrange FDs for user process*: Rearrange all FDs with use of `dup2` and `close` to agree on the arrangement prior to the checkpoint.
5. *Restore memory and threads*: Restore all local process memory and treads of the application.
6. *Refill kernel buffers*: Fill all buffers to resume as if it just finished making a checkpoint.
7. *Resume user threads*: Continue executing the application as before a checkpoint.

DMTCP creates a wrapper around the system call *connect* and *accept* which recreates and reconnects sockets. This wrapper stores information about sockets such as globally unique socket ID which remains constant even over relocations.

The downside of this approach is the fact that DMTCP only can checkpoint applications which do not use libraries and kernel Application Programming Interfaces (APIs), like `inotify` to monitor file system events. Additionally, due to intercepting and forwarding requests, performance issues may arise during execution.

DMTCP supports Linux distributions with kernel version 2.6.9 upwards. It is ported to the ARM architecture by using Embedded Application Binary Interface (EABI) API but no port for ARMv5 is currently available [29]. Because the AEOS controller supports EABI API, it is possible to port it to be functional on the AEpu [26].

## 5.2 BLCR

Duell [31] proposes BLCR, an open-source system level checkpoint/restore mechanism designed for high performance applications. It is developed as a kernel module and a user space library.

The application to checkpoint has to register a threaded callback, which blocks in the kernel until the application requests a checkpoint. When the utility initiates a checkpoint, it opens a particular file and calls `ioctl` on it containing a structure argument, a target identifier and a file handler to which the utility should write the content. The structure argument and target identifier is most often a process and its process ID.

The `ioctl` also unblocks the callback thread which runs the thread-based callback functions in user space. Once all threads complete their callback functions, one thread is chosen to write the header of the checkpoint file, including information about all threads, their parent/child relationships and all shared resources. After that, the kernel module *VMADump* is invoked to write the registers, signal information and Process Identifier (PID) of all threads to the checkpoint file. After dumping this data, it marks the checkpoint completed, and it kills all processes.

The restart utility uses the `ioctl` function to create the parent and as many children as it needs for the threads. Upon creation, all children read their information from the checkpoint file using the kernel module *VMADump*. One of the threads also reads the header and uses this information to restore the PIDs and the relationships between parent and children.

The downside of this mechanism is the requirement to modify the application to checkpoint and restore. The application must load a library which implements the callback functions. Because this technique is developed with the focus on high-performance applications, support for Transmission Control Protocol (TCP) and UNIX sockets is not added. BLCR supports Linux kernels from version 2.6 and ARM support is currently experimental [46].

## 5.3 CRIU

CRIU [22] is a system level user space application to checkpoint an application to several files in memory and restore it at the same state as before the checkpoint. The files in memory include memory pages, FDs and inter-process communication.

Creating a checkpoint consists of three stages [21]:

1. *Collect process tree and freeze it*: By using the given PID, CRIU walks through the virtual file system `/proc` to collect all the threads and children. With use of `ptrace` it freezes all tasks.

2. *Collect tasks' resources and dump*:

   (a) Read and dump Virtual Memory Areas (VMAs), mapped files, FDs and core parameters from `/proc`.

   (b) Inject bytes for `mmap`. Run an injected system call with `ptrace` to allocate memory for parasite code.

   (c) Copy code to allocated memory and set address to parasite code.

   (d) Dump credentials and contents of memory from parasite context.

3. *Clean-up*: Drop all parasite code and restore original code. CRIU detaches to continue running the application without modifications.

Restore consists of four stages [21]:

1. *Resolve shared resources*: Read image files and find processes which share their resources.

2. *Fork the process tree*: Call fork as many times as needed to recreate all processes required.

3. *Restore basic task resources*: Open files, prepare namespaces, map memory areas, create sockets and call `chdir` and `chroot` to change current directory and root for the process.

4. *Switch to context, restore and continue*: Use the restorer code to `munmap` and `mmap`. This restores the memory mappings, credentials and threads. Finally, it restores all timers to make sure they are not fired too early.

According to their website, CRIU is mainly implemented in user space but to provide some access to process credentials and inject parasite code it relies on the system call `ptrace`. Additionally, it is used to dump open files, credentials, registers and task states into image files. During creating a checkpoint of a process tree, CRIU checkpoints each connected process child individually.

If the PID is set correctly to the same PID at the time of the checkpoint, CRIU restores the application in the same state as it was while creating the checkpoint, including open files, memory and other information from the image files. When finished, CRIU removes the injected code and returns the control to the process.

Currently, CRIU only supports ARMv6 and onwards because of usage of the atomic instructions `ldrex` and `strex` which are not part of the ARMv5 instruction set [19].

## 5.4 OpenVZ

OpenVZ is a mechanism to create isolated Linux containers [51]. Containers are like stand-alone devices including root access, user management, IP addresses, memory, processes, files, system libraries and configuration files. In comparison to VMs, containers are much smaller and less resource intensive because it only containerises applications while VMs virtualises a complete OS.

Because setting up a container involves configuring users, IP addresses and other configurations, it increases the programmer complexity. Additionally, to support OpenVZ, a custom kernel must be used, and this implies updates to the mainline kernel are not directly supported.

Because the application to checkpoint is fully isolated from the system underneath, it is possible to create a checkpoint of a container including the running applications state and restore it on-demand. Currently, OpenVZ makes use of CRIU to checkpoint and restore containers. Therefore, compared to CRIU, only the ability to use containers is added by this technique [23].

## 5.5 LXC

LXC is similar to OpenVZ but supports the mainline kernel to containerise applications. It combines *cgroups* and isolated namespaces for Linux [47]. Cgroups are kernel features which can limit or isolate the resource usage of a collection of processes. Namespaces are also kernel features to provide a different view of resources to different processes. A process can have therefore the same PID as another process as long as they are not in the same namespace.

Furthermore, LXC uses CRIU for creating a checkpoint of a container and hence, it only adds the advantages of using containers.

## 5.6 Comparison

This section uses DSE to compare the proposed techniques. First, based on the requirements, it shows the criteria and a corresponding score for each one of them. After that, it presents the results of the comparison with an explanation.

### 5.6.1 Criteria

The general requirements are given in Section 1.4. Based on these requirements, the criteria for checkpoint/restore mechanisms are set up. Each checkpoint/restore technique discussed in this chapter is compared according to these criteria. To build a new update approach, it is of importance the best fit is chosen and used later on according to the requirements given. Because all the proposed techniques are open-source, this criterion is not used in the comparison.
The comparison of the checkpoint/restore techniques in this chapter uses the following criteria:

1. *Mainline kernel*: Does the technique use the mainline Linux kernel, or does it implement its own?
2. *Transparent*: Is the technique transparent to the application it has to checkpoint/restore? Are application adjustments required to use the technique?
3. *Sockets*: Is the technique capable of creating a checkpoint of UNIX sockets such as TCP?
4. *Kernel version*: Which kernel version is mandatory to use the technique?
5. *Hardware compatibility*: Is the technique able to run on the AEOS controller?

Each criterion has a specific weight based on its importance within the new update approach. The method of scoring is identical to the one used in Section 4.8 even as the equation used to calculate the final score of each technique.

### 5.6.2 Results

The final proposed update process must improve the overall way of updating. One of the requirements is decreasing the downtime but also the programmer effort of an update is important. Programming should be about fixing bugs and adding features instead of making sure the update performs well on each controller separately. Therefore the criteria about using the mainline kernel and transparency to the application are important, hence have a high weight. Implementing the new update approach in current products is also of importance, and therefore, the hardware compatibility has a high score. The support of creating checkpoints of sockets is less important than others. The controllers are developed to deal with network losses and system failures. If a socket does not close during a checkpoint, the application resolves a reconnect as soon as possible. On the other hand, if a technique supports creating checkpoints of socket states, less disruption of the normal operation takes place and therefore, scores average. The kernel version is less important, as stated in Section 4.8 because Nedap updates it before this approach is released.

Table 5.1 presents the trade-off table of the checkpoint/restore techniques discussed in this chapter. Next, each technique is discussed according to the criteria.

Table 5.1: Trade-off checkpoint/restore techniques

| | Mainline kernel | Transparent | Socket support | Kernel version | Hardware compatibility | Total |
|---|---|---|---|---|---|---|
| Weight | 3 | 3 | 2 | 1 | 3 | 12 |
| DMTCP | + | - | + | + (2.6) | + | 6 |
| BLCR | + | - | - | + (2.6) | + | 2 |
| CRIU | + | + | + | - (3.11) | 0 | 7 |
| OpenVZ | - | + | + | - (3.11) | - | 0 |
| LXC | + | + | + | - (3.11) | - | 4 |

DMTCP is a good fit for creating a checkpoint on the AEOS controller. It uses the mainline kernel to ensure critical updates can be applied as a new kernel version is released. Because it is an application level approach, it uses a library, and therefore, the application has to be adjusted. Additionally, this decreases the performance due to wrappers around system calls. DMTCP introduces wrappers to checkpoint sockets, but applications which use external libraries are not supported. Because kernel version 2.6 includes the system calls of DMTCP and the AEOS controller supports EABIs used by this technique, it scores above average.

BLCR is a system level checkpoint/restore technique but the user must couple it to the application, to register callback functions. Due to the system level approach, the checkpoint and restore process takes mostly place in kernel space and on OS level. It supports mainline kernels from version 2.6 and a port to the ARM architecture is available but experimental. Due to the focus on high-performance applications, support for socket checkpoint is not available resulting in a lower total score.

CRIU is a full system-level technique which is fully transparent to the application to checkpoint including creating checkpoints of their FDs and sockets. It supports devices from the mainline kernel version 3.11 and ARM architectures which supports atomic instructions. Because the controller uses an ARMv5 architecture without atomic instructions, it is not supported. However, it is possible to port atomic instructions to older ARM architectures by using other instructions. Therefore, the technique scores high.

OpenVZ is an effective technique to isolate applications from the system and other applications. By using containers, the security and migration capabilities increases and containers require fewer system resources than VMs. Because the controller application already makes use of a Java Virtual Machine (JVM) and extra isolation is not required, containers are less applicable for the new update approach. OpenVZ is part of the Virtuozzo community which uses their kernel and underneath uses CRIU, therefore further implementation is less beneficial for this research.

LXC is similar to OpenVZ but makes use of the mainline kernel. Therefore it scores higher, but because containers are less usable and it only increases the resource usage, it is less applicable overall.

# Chapter 6

# Implementation of the methods

As a result of Chapters 4 and 5, it can be concluded that a combination of Kexec and CRIU is the best option to elaborate further on. Kexec uses several kernel options to overwrite the running kernel with a new kernel resulting in less downtime. CRIU can checkpoint the application in front of an update and restores it after the partial reboot, to skip the initialisation on the controller. In this chapter, both techniques are further explained to give more details about the options which are available for both techniques and how CRIU is ported to the ARMv5 architecture.

## 6.1 Kexec

Implementing Kexec consists of configuring the kernel and installing the user space tool. From kernel 2.6 the Kexec kernel options are merged and need enabling before compilation [35]. The user space tooling, Kexec-tools, can be downloaded as a package or as a source from the official website.

After installing Kexec-tools and configuring the kernel, the user can load a new kernel. Loading the kernel consists of providing the file path of the kernel and additionally, the new kernel command options and possible other configurations like the location of *initrd* or memory configurations. After loading, the kernel is stored in kernel memory but not yet on the right address, which ensures the system can continue without effect. It is also possible to unload the kernel.

Executing the direct reboot can be performed the *nice* way by synchronising the file system, shutting down interfaces and execute the Kexec reboot. Alternatively, the user can force a reboot to directly start the new kernel without synchronisation and shutting down. Without carefully synchronisation and resetting interfaces, states are unknown, and devices might fail to operate after reboot.

After rebooting, the kernel and user space are initialised and operational as before. The new kernel is started and stored on the kernel address resulting in a fully operational system with a new kernel.

## 6.2 CRIU

This section explains the port of CRIU to the ARMv5 architecture. Additionally, it describes the different possibilities of CRIU, for example, using TCP connections, memory tracking and lazy restore.

### 6.2.1 ARMv5 port

As Chapter 5.3 describes, CRIU is currently not available for the ARMv5 architecture. To port CRIU, the atomic instructions need to be expanded with instructions supported by the architecture [20]. Looking closer to the source code of CRIU, it uses the instructions `LDREX` and `STREX` multiple times to exclusively load and store. Unfortunately, architectures before ARMv6 do not support these instructions, and hence, a new implementation must be found. Additionally, CRIU currently uses memory barriers to prevent reordering of instructions. This memory barrier is not compatible with ARMv5, and another approach should be implemented.

**Atomic instructions**
The first question to answer is if the controller requires the atomic instructions. The controller contains a single core processor, and therefore no processes are executed in parallel. However, CRIU can checkpoint a process tree containing multiple threads. The situation can occur that multiple threads are restoring the same data structure, for example, an IPC socket. By restoring, the data structure can change, which results in race conditions. In the end, the requirement of atomic instructions persist.

The next question is how to solve the atomicity requirement. Multiple solutions are possible for this. The first is the most straightforward solution which the kernel currently uses, namely to store interrupt states, disable the interrupts, and afterwards enable and restore them. The second solution is to implement the atomic functions by implementing the same approach as currently used for ARMv6 and above but by using ARMv5 capable instructions. A third possible solution is to use kernel helper functions. The kernel code contains these functions to implement newer instructions into older architectures, for example, atomic functions as `atomic_add`.

To port CRIU to the ARMv5 architecture, this research proposes a solution based on the simplicity and the number of instructions. Because disabling interrupts including saving and restoring the states is a complex task in kernel space and even more complicated in user space [45], this research does not elaborate this solution further. Using kernel helper functions is interesting because it implements most functionality by configuring the kernel properly [44]. This reduces the complexity significantly but increases the kernel size by enabling all helper functions and also the amount of instructions to use is relatively high because of the requirement of multiple functions to get the same implementation. Therefore the best fit for the new update approach is to modify the existing atomic functions to use instructions available for ARMv5 architectures. In the end, this also enables other users to use CRIU on an ARMv5 architecture by publishing a patch to the community.

CRIU implements the atomic load and store instructions to prevent race conditions while restoring multiple threads. Changing the instructions while preserving the functionality, consist of a synchronisation mechanism which ensures multiple threads are not adjusting data at the same time. Computer scientist commonly uses semaphores to provide this functionality. A semaphore implements a variable to control access to a shared resource [57]. In older ARM architectures the semaphore behaviour is commonly implemented by using the `SWP` instruction [30]. This instruction atomically copies data from a memory address to a register and stores a value from a register to the memory address.

Listing 6.1 shows the current instructions to exclusively load, add a value, and exclusively store.

```
1: ldrex   %0, [%3]          ; exclusive load (%r0 ← M[%r3])
   add     %0, %0, %4        ; addition (%r0 ← %r0 + %r4)
   strex   %1, %0, [%3]      ; exclusive store (M[%r3] ← %r0 , %r1 ← status)
   teq     %1, #0            ; test equal (%r1 = 0?)
   bne     1b                ; branch not equal (1 before)
```

Listing 6.1: Instructions for atomic add for ARMv6

Using the `SWP` instruction instead of the exclusive load and store, it is possible to port the code for ARM architectures from version 6 to the ARMv5 architecture. Listing 6.2 shows the ported code.

```
   mvn   %1, #0             ; move not (%r1 ← NOT 0)
1: swp   %0, %1, [%3]       ; swap (%r0 ← M[%r3], M[%r3] ← %r1)
   cmp   %0, %1             ; compare (%r0 = %r1?)
   beq   1b                 ; brange equal (1 before)
   add   %0, %0, %4         ; addition (%r0 ← %r0 + %r4)
   swp   %1, %0, [%3]       ; swap (%r1 ← M[%r3], M[%r3] ← %r0)
```

Listing 6.2: Instructions for atomic add for ARMv5

As Listing 6.2 presents, the first instruction stores a value in register 1 to act as a lock flag. The value is set to the maximum possible integer value, ensuring no deadlocks occur due to using a flag which

is possibly equal to the data stored on the memory address. After swapping the flag to the memory address and retrieving the content, a compare instruction (line 3) executes a compare to check if the data is not in use. If it is currently in use by another thread, it keeps trying until the other thread unlocks the data. If the data is unlocked, the thread can perform the addition, and it atomically stores the new value back in memory by using the `SWP` instruction again.

For subtraction, the same approach is implemented but with a `sub` instruction instead of the `add` instruction.

**Memory barrier**
Compilers often refactor the code to improve the performance of the execution of applications. For ARM architectures this also implies reordering instructions to hide pipeline latencies or take advantage of pipelining. Because the order of instructions in the atomic functions is highly important, no reordering optimisation should occur. The ARM architecture prevents this by implementing memory barriers.

CRIU implements memory barriers for ensuring that no reordering takes place in the atomic functions. Currently, CRIU uses the following instruction at the start and end of the function:

```
#define smp_mb() __asm__ __volatile__ ("mcr p15, 0, %0, c7, c10, 5" : : "r" (0) : "memory")
```

This function executes a functionality not present in the core processor but part of a co-processor, in this case, a function of the Memory Management Unit (MMU). By using the right registers this instruction prevents reordering and thus implements a memory barrier [5][4].

The ARMv5 architecture does not implement the memory barrier functionality by using the MMU command. Thus a compatible equivalent should be used. Looking at the migration guide from ARMv5 to ARMv7 [6], a GCC inline assembler memory clobber can be used. This assembly code indicates that the instructions change memory, and no reordering is allowed across the barrier. The memory clobber is implemented as follows:

```
#define smp_mb() __asm__ __volatile__ ("" : : : "memory")
```

Besides the atomic instructions and the memory barrier, the makefiles are patched to add compatibility for ARMv5 architectures. Finally, the compiled application is tested.

### 6.2.2 CRIU configuration

The implementation of CRIU into the new update approach consists of two stages, the checkpoint before a reboot and the restore afterwards. For both stages, this section discusses the different options available.

**Checkpoint**
To create a standard checkpoint, CRIU only needs the PID of the application, and it saves the checkpoint consisting of several files at the current directory. For most of the applications, this approach is sufficient, but if the application uses TCP connections, CRIU must enable repair functions.

Creating a checkpoint including the TCP connections and states, the user must enable the option `--tcp-established` by creating the checkpoint but also by restoring. This ensures it creates a new file for storing all the established TCP connections. The option uses the `TCP_REPAIR` option from the kernel to store the current state, and restore it while letting the protocol repair the data sequence. One particular thing to note is the requirement that the IP addresses do not change between the checkpoint and restore, otherwise CRIU is not able to restore the TCP connections.

Besides a standard checkpoint, CRIU can create a checkpoint while keeping the application running. After that, CRIU can detect changes in memory pages over time to checkpoint these changes. This incremental checkpoint can decrease the freeze time of an application by getting the memory of an application and start writing it into files without an application freeze. So it only freezes the application

to get changes in memory. To use incremental checkpoints, the architecture must provide memory tracking. ARM does not support memory tracking, and therefore it is not considered further in this research [48].

If the controller can track memory changes, the application can continue after creating a checkpoint. CRIU can add the changes during a second checkpoint. However, without memory tracking, the continuation of the application can be fatal if, for example, after creating a checkpoint, the application establishes a new TCP connections. If the user does not take this into account, the restore might fail, or the behaviour of the application is affected.

**Restore**
Restoring an application consists of pointing CRIU to the image files and providing the corresponding options. For example when a checkpoint is generated with using `--tcp-established`, the restore process must also contain `--tcp-established`.

Besides the standard options, CRIU implements the ability to perform the restore not at once but only restore the parts which are required to get the application running again. This technique is called *Lazy restore* and makes use of functionality introduced in kernel version 4.11 [24]. A daemon controls all the memory pages and checks if any page fault is occurring. If an application tries to access a memory page which is not yet restored, it causes a page fault. The daemon notices this and populates the memory with the memory page from the checkpoint. Currently, all systems with `userfaultfd` enabled, do support lazy restoring. Because it is experimental, shared memory, mapping one page to two places and synchronisation between pending forks and transfers is not yet implemented.

One general limitation of checkpoint/restore is the requirement that the application must have the same PID after the restore as before the checkpoint. This can be a problem when another process is using this PID during the restore resulting in an error. Killing this process can resolve the issue, but sometimes the process is in use, or it is a system process, and killing does not solve the issue. As a solution, the new update approach uses the PID namespace. The PID namespace isolates the ID of a process from the rest of the system. The namespace itself gets an available PID from the system, and inside the namespace, the PIDs start from 1 again and therefore the application can get the same PID again. Figure 6.1 shows an example of the use of PID namespaces, PID 5 starts a namespace, and inside the namespace, the IDs of the processes starts from 1 again.

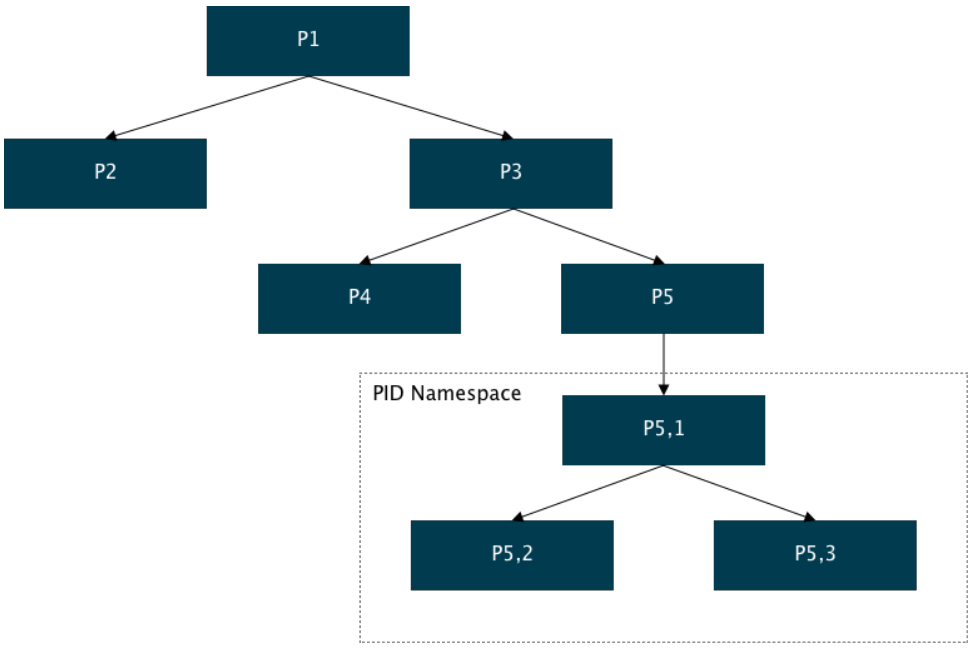



Figure 6.1: PID namespace example

# Chapter 7

# New update process

This chapter describes the design and implementation of the new update process based on the conclusions of Chapters 4, 5 and 6. Of every step, the design is discussed, followed by the overall implementation. However, first, the overall design is discussed.

## 7.1 Overview

As described in Section 1.3, the goal is to research a new update process for an AEOS controller to decrease the downtime. To meet this goal, the following update process is proposed consisting of downloading the update files to the controller, checking the integrity, loading the new kernel, creating a checkpoint of the application, restarting into the new kernel and restoring the application. Figure 7.1 presents the new update process divided into six steps. The steps indicated in orange are the steps which cause downtime while the blue steps can be performed before stopping the access control application. Details about the design of each step are discussed next.

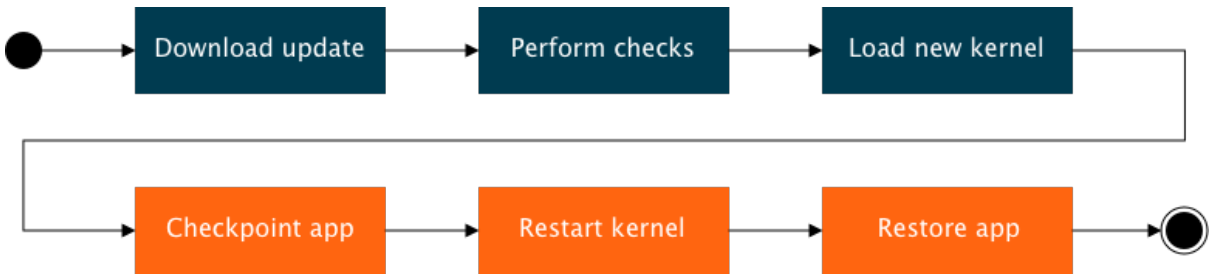



Figure 7.1: Overview of the new update process

### 7.1.1 Download update

Before updating, the update application must transfer all files to the controller. Because this step does not imply downtime, and it checks all files before transferring, this step is equal to the first phase of the current update process. Considering this is a proof of concept, AEmon is not used and downloading update files consists of manual transferring the image using scp. After transferring, the remaining part of the proof of concept executes automatically.

### 7.1.2 Perform checks

The proof of concept also implements the integrity checks by using MD5-hashes of step two. This checks the transferred image files before using them. The most important difference from the current update process is that the new process does not stop the application after downloading all files, but keeps it functional by using multiple partitions explained in Section 7.2.

### 7.1.3 Load new kernel

The next step is to load the new kernel with the use of Kexec. As Section 6.1 presents, the user must provide the new kernel and additional options when executing Kexec load. After that, Kexec segments the new kernel, stores it in user space memory and copies it to the kernel pages. Because only the kernel is loaded and the running system remains intact, the application can continue executing.

### 7.1.4 Checkpoint application

While the new kernel is successfully stored and ready to start, the new update process creates a checkpoint of the application to persist the state of it over a reboot. The new update process only creates a checkpoint of the AEOS Java application, including all its processes, because other running processes do not introduce significant initialisation time. Creating a checkpoint and restoring them would, therefore not reduce the downtime. To know which application to checkpoint by CRIU, the new update process must find the PID of the application. The Linux command `ps` reports a snapshot of current processes with PID, executable name and cumulated CPU time. Searching through this list results in the corresponding PID. If the correct PID is found, the update process runs CRIU using the PID-namespace script as described in Section 6.2. Finally, after the checkpoint, the script checks the log-file to determine if the creation is succeeded and therefore if the update can proceed.

### 7.1.5 Restart kernel

With the creation of the checkpoint, the application is shut down, which implies downtime. The main goal of this research is to ensure as less downtime as possible. Therefore the new update process restarts the system as soon as possible after the checkpoint. Because the kernel is already present in the kernel pages, the process synchronises the disks, shuts down interfaces and quits the system. Executing Kexec transfers control to the reboot process of Kexec and starts the new kernel as described in Section 4.5. Because Kexec sets the kernel and corresponding settings during the load stage, no further actions are required and only firing Kexec is sufficient.

### 7.1.6 Restore application

After a successful reboot, the system is running on a new kernel and with a new file system. The checkpoint process before the reboot stores the image files on the second partition and therefore they are directly accessible after the reboot. The restore process consists of executing CRIU using the same options as provided during the creation of the checkpoint. After a successful restore, the application continues in the same state as during the checkpoint. The access control application is running again after an update of the kernel and file system.

## 7.2 Fail-safe methods

Most of the returned controllers are broken by a failed update. Users are often not able to recover from these issues and are required to send the controller back to Nedap. Nedap flashes new software on the chip and returns it to the user. To make sure the controller remains functional over a failed update, the new update process needs to be atomic. An update process is atomic if a transition takes place from old software to a fully operational new version, with no other state possible, even if power losses occur. The system always boots a valid new version of the software if it succeeds or a valid old version if the update fails [64].

This section proposes two techniques to ensure update atomicity. The first technique is to use multiple partitions to prevent overwriting working code and the ability to reset your device. The other technique is to use a hardware watchdog to monitor the behaviour of the system and act upon. The new update process uses both to guarantee an atomic update process.

### 7.2.1 Multiple partitions

The first technique to provide fail safety for an update is using two or more partitions [64][14]. Currently, the controller uses several partitions on the NOR flash to separate the bootloader, bootloader configurations and the kernel and one partition on the NAND flash, Figure 7.2. The AEOS application is only using a small part of the partition on the NAND flash and therefore using multiple partitions is possible.


Figure 7.2: Current partition layout

Figure 7.3 shows the partitioning of the flashes of the proof of concept. Each controller consists of a minimum of 2 GB NAND flash; hence, each partition is 1 GB. Additionally, each partition consists of a full system, including kernel files and the file system. Depending on the maximum size of AEOS, the partition layout can be adjusted. For example, system setup is imaginable consisting of four partitions; one for system A, one for system B, a configuration partition and a data partition. This adds the ability to reset the AEOS Java application by erasing the configuration partition. It is even possible to remove all data and perform a full data fetch in case the saved data is corrupt. An additional benefit of using multiple partitions is the decrement of flash degradation. NAND flash can only handle a finite number of erases due to the implementation of the semiconductor layer. Over time, the bit-error rates increases and the flash becomes unreliable. By using multiple partitions, erasing is only required for one partition per update resulting in a longer flash lifetime.


Figure 7.3: New partition layout

To perform a fail-safe update, the system solely uses one partition for the new version and the other partition for the running system. During the update, the following steps take place:

1. Check which partition the system currently uses.
2. Mount the second partition to be able to write the update files.
3. Download the new file system to the second partition.
4. Check the integrity using MD5-hashes.
5. Change kernel parameters to ensure the system reboots into the second partition.
6. Reboot the system.

### 7.2.2 Watchdog

The previous technique ensures no working code is overwritten during the update but does not guarantee the device reboots automatically into the last working partition if a failure occurs. For example, the checkpoint of the Java application has become corrupt during writing and therefore, does not start. Adding a watchdog can resolve this particular issue and also add extra fail-safe measures for other issues.

A watchdog is a hardware or software system to automatically detect software anomalies during runtime and reset the system in case such anomaly occurs [49]. Generally, the system sets a counter with the initial watchdog value and counts down from that value to zero. The software system detects the watchdog value and periodically resets it. If the counter reaches zero before a reset, the system reboots. Especially for embedded systems which need to be self-reliant, these fully autonomic technique is commonly implemented. A general setup is shown in Figure 7.4

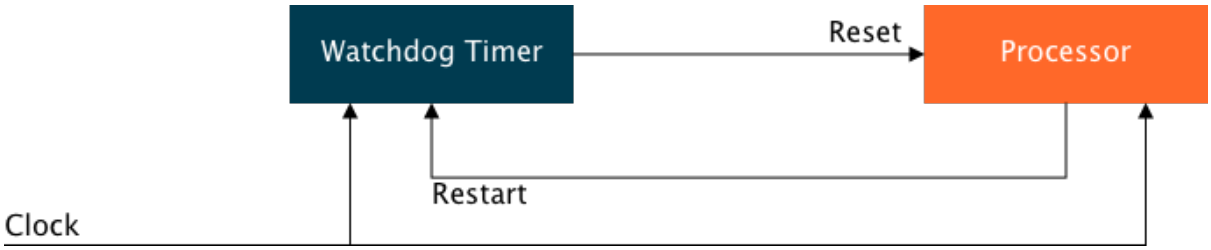



Figure 7.4: Watchdog setup

Implementing a watchdog starts with opening a watchdog device in `/dev/watchdog`. When open, the watchdog starts counting from the initial value to zero. To keep the system from resetting, the process must perform a watchdog reset at least once a minute [28]. If no write to the device is performed before the deadline, the system resets. When the new software system is running successfully, the watchdog can be disabled by closing the device.

Resetting the system might not be enough to remain functional. When an entire file system is corrupt, or the application is broken, the system must switch back to the last working state. The bootloader U-boot implements a solution for this by implementing `bootcount` [32].

The `bootcount` is a variable located in the bootloader configuration which increments at each reboot. A second variable implements `bootlimit` to ensure the bootloader only tries to boot a specific configuration (kernel and file system) a predefined number of times. When the bootloader reaches the `bootlimit` value, it executes the configuration stored in `altbootcmd`.

The new update process sets the `bootlimit` value to 1 to ensure the bootloader switches immediately to the `altbootcmd` setup in case of an update failure. The low limit decreases the downtime by ensuring the system only resets once to boot directly back to a functional system. In case the system boots successfully in an updated version, the process changes the `altbootcmd` and `bootcmd` to point to the new working system. To change the bootloader configuration, the kernel system call `fw_setenv` can be used.

## 7.3 Implementation

Combining these techniques into one process delivers a new update process with as less downtime as possible and on the same time ensuring fail-safe operation by using two partitions and enabling the watchdog to reset the device to the last working state. Figure 7.5 presents the complete new update process including fail-safety.

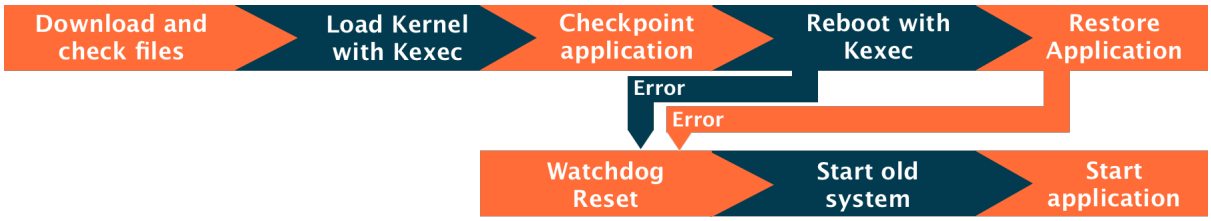



Figure 7.5: Implementation of new update process

The new update approach normally runs only the five phases of the upper part of the figure. These phases are required to download, check the files, checkpoint an application, reboot and restore the application. As stated in Section 7.1, only the phases to checkpoint the application, reboot into the new kernel and restore the application, causes downtime if the update runs successfully. If the update fails in one of the first three phases, the application continues running without downtime. This ensures that the access control system is still functioning, and the user must resolve the error before trying to update again.

If one of the last two phases fails, the system is already down and can only revive by returning to the old system. Therefore, the system triggers the watchdog; the system reboots into the old system and starts the application using the conventional approach. This fail-safe mechanism introduces extra downtime by resetting the whole device and starting the application without restore but ensures the system is always able to revert to the last working state.

Appendix B contains the implementation of this approach. It is implemented in two shell scripts to meet the coding languages Nedap Security Management uses. One script contains the phases from 1 to 4, and the second script contains the last phase. This last phase contains restoring the application and is triggered automatically on each boot of the system by using the initialisation manager of the Linux system.

# Chapter 8

# Results

In the previous chapters, the new update process and its techniques are explained. This chapter covers the results of this new update process in terms of relative speed up between the old process and the new process. Because the focus of this research is to increase availability by decreasing downtime, this chapter focusses on the comparison of the downtime phases of the old and the new update process. The other phases can be performed before stopping the application or after the restart of the application to avoid downtime.

Running the new update process on the AEOS controller requires a minimal Linux kernel version 3.11 and a file system containing Kexec and CRIU. Nedap Security Management is currently developing a new bootloader, kernel and file system but, unfortunately, these are not yet available for testing the new update process. Therefore, QEMU, a full system emulation tool is used to imitate the architecture of the controller. The first section of this chapter explains the differences between the hardware and the emulation system. Because the differences introduce performance discrepancies, the old update process is also analysed in the emulation system to get a relative speed up. This is explained in the second section. After that, the new update process on the emulated system is explained, and finally, they are compared to give a conclusive relative speed-up.

## 8.1 Differences between hardware and emulation

ARMv5TE 800MHz
16 MB NOR flash
4GB NAND flash
32-bit RISC architecture
U-boot bootloader

QEMU
Full-system emulation

ARMv5TEJ
No NOR flash
2x 4GB NAND flash
32-bit RISC architecture
No bootloader

Figure 8.1: Differences in systems

As stated at the beginning of this chapter, the new update process requires a new kernel and file system, including Kexec and CRIU. For testing the process, a new Linux build is developed with the use of Yocto, consisting of Linux kernel 4.18, a Poky distribution and all required applications and scripts. For more information about Yocto, the new Linux build and how to rebuild the system for future usage, inform appendix C.

Besides the difference in Linux and the OS of the current controller and the emulation, also the hardware is not equal. Figure 8.1 shows the most important hardware specifications for both the AEOS controller and the emulated system. Section 1.2.2 already explained the hardware specifications of the AEpu but important to note is that it consists of an ARMv5 architecture, the combination of NOR and NAND flash and the use of the U-boot bootloader. Comparing these specifications to the system used for the emulation tool, three important differences show up.

The first difference is the uncertainty of the maximum processing power of the system in the emulation tool. Instead of running the applications on one core with a maximum of 800 MHz clock frequency, such as the AEOS controller, QEMU makes use of dynamic translation [61]. Dynamic translation converts the application code of the emulated architecture on demand to instructions supported by the host. After that, it runs directly on the host architecture. This implies that QEMU is not able to limit the processing power, which results in performance discrepancies with regard to the controller. Ultimately, this also affects the required downtime of the update process.

The second difference is the absence of the NOR flash in the emulation. The reason for this is that the development board used as architecture in QEMU does not support NOR flash and only can use NAND flashes. Other hardware boards are available, but these use different processor architectures. To test the functionality of CRIU, the ARMv5 architecture is more important, and therefore the NOR flash is omitted. This results in a system with only NAND flashes and to simulate the two partitions of the new update process, the emulation includes two flashes. One for partition A and one for partition B. In the end, QEMU loads the kernel from the host system directly into the RAM of the emulated system, executes it and starts the user space from one of the NAND flashes.

The third difference is that QEMU does not use a bootloader upon boot. Because all the emulated hardware is virtual, QEMU manages the states and initialisation. During a boot, QEMU skips the hardware detection and initialisation, normally part of the bootloader, and directly runs the provided kernel.

The three differences in hardware do not influence the functionality of the software compared to running it on the AEpu. However, the performance can vary due to the differences in processing power and peripherals. In the end, the functionality of the new update process can be analysed, but to provide results in terms of speed up between the old and new process, both have to be implemented on the emulator. By analysing and comparing both the old and the new update process on the emulator, it is possible to measure a relative speed up.

## 8.2 Old update process in emulation

The old update process consists of five phases, as Chapter 2 explains. It downloads the update, copies and checks the update files, flashes the NOR with the new bootloader and kernel, reboots the complete system and starts the new application. Because downloading the update does not introduce downtime, this phase can be skipped from the test. Because the NOR flash and the bootloader are both not implemented in the emulation, giving one conclusive measurement of the downtime of the old update process in emulation is not possible. However, to be able to compare the old and new process, this section introduces a pessimistic and optimistic bound of the old update process.

### 8.2.1 Pessimistic bound

The pessimistic bound is introduced as the maximum downtime the old update process introduces on the emulated system. Figure 8.2 shows the five phases of the pessimistic bound of the old update process. The *copy and checks* phase, *kernel and file system* phase and *start application* phase are all analysed running in the emulation. Because of the lack of a NOR flash and a bootloader, the second and third phase cannot be measured. The measurements of flashing the NOR and running the bootloader are therefore reused from Chapter 2 and because the emulator uses hardware with more processing power, the assumption is made that flashing the NOR and executing the bootloader takes at least the same time or less on the emulated system.


Figure 8.2: Old update approach pessimistic

The time measurement starts after the update process finishes downloading all the files to the emulated controller. It starts with checking the integrity of the new file system, whereafter it overwrites all the files, and finally, it checks the integrity of the new bootloader and kernel. At the second phase, it delays for 70 seconds equal to the time the AEpu requires to erase and rewrite the NOR flash. The third phase includes the first part of the reboot, but because QEMU does not use a bootloader, the test delays the process by 3.8 seconds, equal to the time U-boot requires to initialise and start the kernel on the controller. After this phase, the emulator reboots, which includes starting the new kernel and booting the new file system. Finally, the AEOS Java application starts and the update is finished.

The old update process takes at most 151.3 seconds of downtime on the emulated system. As stated before, this is the upper bound of the old update process because it uses virtual hardware with better performance.

### 8.2.2 Optimistic bound

The optimistic bound is introduced as the minimal downtime the old update process can take on the emulated system. Figure 8.3 shows the three phases of the optimistic bound. Comparing this bound to the pessimistic one results in the lack of the phases which are not measured on the emulation. Because the assumption is made that the emulator is faster than the AEOS controller, it can at most be that fast that flashing the NOR and running the bootloader takes no time. With zero downtime of these two phases, only three downtime phases remain, which consists of the *copy and checks* phase, *kernel and file system* phase and the *start application* phase.

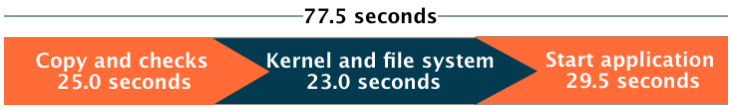



Figure 8.3: Old update approach optimistic

Because the remaining three phases are equal to phase 1, 4 and 5 of the pessimistic bound, also the timings of these phases are equal. This results in an optimistic bound of 77.5 seconds.

### 8.2.3 Comparison of pessimistic and optimistic bound

Removing the phases of the NOR flash and the bootloader results in a significant difference in downtime of 73.8 seconds. The assumption is made that the downtime of the old update process in emulation takes between the optimistic and pessimistic bound of respectively 77.5 and 151.3 seconds. Because there is no possibility to measure or calculate the actual downtime, only this range can be given to compare the update processes.

## 8.3 New update process in emulation

Figure 8.4 shows the new update process fully simulated in QEMU. Because the new process uses two partitions, it introduces no downtime by copying the update files to flash. Additionally, it performs all checks and measures to set up the update while running the AEOS Java application, which reduces the downtime even further. Ultimately, this results in three phases which introduces downtime, the *checkpoint*, *Kexec reboot* and the *restore* phase. For the remaining phases required to perform a full update using the new update process, see Section 7.3.

The new update process uses CRIU and Kexec, which both are responsible for the downtime. Whenever the update process is ready to restart into the new kernel, CRIU is used to create a checkpoint of the AEOS Java application. During this creation, CRIU kills the application, which starts the downtime of the system. When it finishes the creation and saving of a checkpoint in the first phase, the system is ready for a reboot. Kexec performs this partial reboot by directly starting the new kernel from the kernel space of the currently running kernel. When both the new kernel and the new file system of the second partition are successfully started, CRIU restores the Java application. At the end of this phase, the system is running using a new kernel and file system while using the Java application as if it was never stopped.

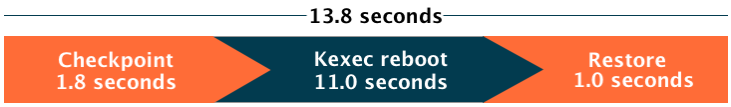



Figure 8.4: New update process

Figure 8.4 also includes the downtime introduced by each phase. The checkpoint, Kexec reboot and restore phase are all executed on the emulation system and introduce downtime to the access control application. The first phase saves the states of the application and stops it to store also the registers and stack. From the point in time it stops the application until it finishes the complete checkpoint process, which takes 1.8 seconds. Directly after that, Kexec reboots the system into the new kernel and file system, introducing the most significant part of the downtime, namely 11 seconds. After a reboot, the initialisation script automatically restores the application with all its connections, memory and states resulting in a downtime of 1.0 second.

Conclusively, the complete new update process introduces 13.8 seconds of downtime, where the checkpoint and restore phases are responsible for 2.8 seconds and the Kexec reboot for 11 seconds.

## 8.4 Comparison and discussion

As can be seen in previous sections, most of the phases of the old update process are removed by using checkpoint/restore and kernel execution. The phases to copy, check, and flash the NOR are still required but can be performed without downtime due to introducing multiple partitions and booting into the new kernel directly from the old kernel.

Table 8.1 shows the timings of each phase of the old processes and the new process. What stands out is the high-performance gain of the phases before and after a reboot. By introducing multiple partitions and Kexec which loads the kernel directly from any location instead of using the NOR flash, we remove all downtime before stopping the Java application. The old process required at least 25 seconds for copying and checking the files and up to 70 seconds for flashing the NOR.

After all update files are written, the old update processes initiate a reboot. The new process, however, requires an additional step because it does not quit the application before copying the files. The new update process kills the application at the very least point before the reboot. This killing is performed by CRIU while creating a checkpoint of the application and takes 1.8 seconds of downtime.

Table 8.1: Results

| Phase | Old (pessimistic) (s) | Old (optimistic) (s) | New (s) |
|---|---|---|---|
| Copy and checks | 25.0 | 25.0 | 0 |
| Flashing NOR | 70.0 | 0 | 0 |
| Checkpoint | 0 | 0 | 1.8 |
| Bootloader | 3.8 | 0 | 0 |
| Kernel and file system | 23.0 | 23.0 | 11.0 |
| Restore | 0 | 0 | 1.0 |
| Start application | 29.5 | 29.5 | 0 |
| **Total downtime** | 151.3 | 77.5 | 13.8 |

The old update processes perform a full reboot of the system to start using the new bootloader, kernel and file system which takes from 23 to 26.8 seconds. This time is required to detect and initialise the hardware and boot the kernel and file system. The new update process skips the bootloader entirely and directly starts the new kernel. Because all hardware is already initialised and no reset is performed, the reboot takes only 11 seconds.

A big part of the downtime of the old update process is introduced by restarting the application. In the emulation, this takes 29.5 seconds, but depending on the configuration and number of authentications, this can take up to 20 minutes. Because before the reboot a checkpoint of the application is created, the new update process uses this to restore the application. After a restore which takes 1.0 second, the state of the application is equal to the state before the reboot, and the application can continue running.

The results of table 8.1 suggests that the new update process updates the kernel and file system within 13.8 seconds in comparison to the 77.5 to 151.3 seconds of the old update process. This leads to a relative speed up of factor 5.6 to 11.0. Because emulation is used instead of the actual controller, the possible differences between the emulation and the implementation on the AEOS controller are discussed next.

As stated earlier, the hardware in emulation differs from the hardware of the AEpu. Comparing the old update process on the controller and the old update processes of the emulation, several differences emerged. First of all, due to the higher processor power, copying and checking the update files is faster in emulation. Moreover, starting the AEOS Java application is much faster on the emulated system than on the controller. Comparing the boot processes of the AEpu and the emulation shows that the actual hardware is faster. This can be due to the configuration of the kernel and the file system. When optimisation takes place to decrease the number of drivers and processes to start during boot, the boot time also decreases. These differences introduce inaccuracy in the test results, and it is essential to note that the speed up is only relative and therefore not conclusive for the final implementation on the AEOS controller. When Nedap successfully ports the new bootloader and kernel to the controller, a conclusive speed up can be measured.

# Chapter 9

# Future work

During the research, some beneficial options and techniques emerge which Nedap should investigate in further research. Some of these techniques do not apply to the current hardware but are valuable for future products of Nedap Security Management. First, the results of research to possibilities of updating the AEOS Java application are described consisting of a Dynamic Software Updating (DSU) and a handing over technique. Finally, several other general research topics and possible techniques are described.

## 9.1 AEOS software update

The new update process currently updates the kernel and file system using techniques to checkpoint and restore the Java application and directly run the new kernel from the old one. This approach reduces the downtime of the update process significantly. However, this approach does not implement updating the Java application in comparison to the old update process.

To determine a new update technique for the AEOS application, research is performed to Java update techniques. However, to implement a proof of concept of these techniques, more research is required. This research should focus on the ability to run two Java applications simultaneously for a short term and the number of application changes required for the techniques proposed in this chapter. Because the Java application update is not part of this research and requires more insights into the application itself, this chapter presents two promising techniques which can be researched further.

The first update strategy is a commonly used technique called DSU. This technique creates a patch for the code to update and takes advantage of the class loading ability of a JVM. The second technique is a more recent one used to hand over program execution from an old version to the new one.

### 9.1.1 Dynamic software update

DSU mechanisms migrate a running software system to a new version without stopping the application. The Java bytecode inside a JVM consists of classes, stacks and the heap. Updating a Java application during runtime consists therefore of locating old classes and update them to the newer ones. After that, the update application locates all old objects in the heap to replace them with new objects.

Figure 9.1 presents a common way of DSU. At the start, the new and old code are compared to create a patch. This patch normally contains all new or changed classes and transformer objects which tells the updater how to transform existing variables or functions to new ones. When a patch is created, the update performs a request to install. From that point, the running software decides when to update, commonly specified in the control flow. When the application is ready to update, it installs the new patch and continues running.

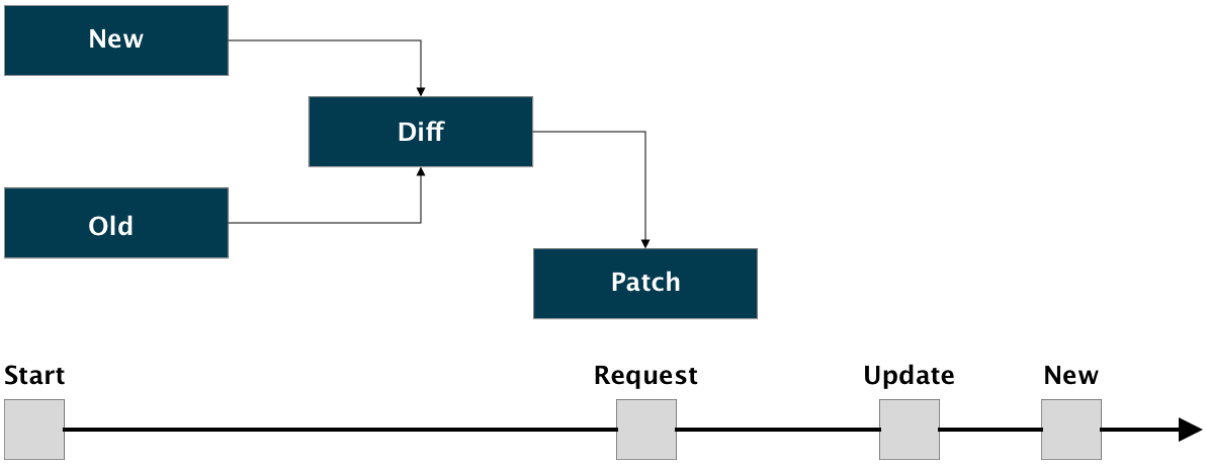



Figure 9.1: Dynamic software update

Gu et al. [36] present Javelus, a DSU mechanism which updates changed code during suspension and migrates objects on-demand after resuming. It is implemented on the Java HotSpot JVM and supports arbitrary changes of Java classes with a disruption of milliseconds.

Javelus consists of two phases: offline preparation and online updating. The offline preparation phase is comparable to the phases in Figure 9.1 from the comparing to the actual patch. The patch contains a transformer object which the user must adjust to ensure that Javelus can convert the current code to the new code. The online update phases are the stages from the request to the end of the update and consists of four phases: patch loading, safe point checking, code updating and objects updating. The first phase loads the patch and prepares the metadata. The second phase waits for a safe point, which is a state in the program where all the threads are suspended, and all stack frames can update. Javelus waits and retries to find such point and otherwise fails after certain retries. The third phase updates all the classes and refreshes all inline or compiled code. Lastly, the fourth phase resumes the application and intercepts access to objects which are not yet updated, to make sure Javelus updates them before use.

Javelus can make use of two different modes. The first mode is eagerly updating, which is updating all classes at once, resulting in a variable downtime due to the size of the patch. The second mode is lazy updating. Lazy updating is a method to update the crucial classes during the actual patch but postpone the rest of the classes to point in time when the new class is required. This increases the number of classes which have to be updated on-demand but decreases the pausing time of the application. From their results, both the eager and lazy approach takes a pausing time less than a second.

Orso, Rao, and Harrold [52] present Dynamic Updating through Swapping of Classes (DUSC), which does a similar thing but with using proxy classes to swap the updated classes. With the use of the proxy classes, no involvement of the running application is necessary. DUSC transforms each class in a bundle of classes, consisting of an implementation class, an interface class, a wrapper class and a state class. The update manager contacts the wrapper class upon update request to swap. Because it only swaps the classes, the actual pausing time is reduced, and their results show an increment of the execution time of less than half a second.

### 9.1.2 Handing over

Concurrently running multiple instances of an application was presented in the 1970s by Chen and Avizienis [15]. Introduced initially to minimise the risk of having the same bug in multiple versions of an application. By voting in case of control flow divergence, the most promising results are determined.

Hosek and Cadar [39] proposes Mx, a system for multi-core processors to target crash bugs. It enables two versions of an application to run concurrently without modifications. Mx performs the synchronisation at system call level, and it can restart crashed applications by checkpoint/restart. Mx first analyses both binaries and constructs a mapping of the control flow of the old application to the new one. After that, it starts both applications, synchronises them and monitors divergences. Mx discovers crashes, restores the crashed application, patches it to resolve bugs and continues executing.

Several other frameworks are proposed already [8][16][44]. The main thing in common is the goal of increasing the reliability of software in multi-processor systems. The AEpu is, however, a single core embedded system and has limited resources. Continuously running two versions of an application and additional monitor software is not applicable. Nevertheless, the idea about running versions concurrently is still interesting to apply for short terms.

Balena [10] presented the idea of the hand-over update strategy further at Dockercon 2016. They used their balenaOS to update a drone while flying, resulting in a downtime of 50 to 200 ms. The main loop of the application must be adjusted to check if a new instance of the application is running, for example, by creating and writing to a UNIX socket. When the second instance starts, it tries to connect to this socket and reads from it. Since a connection is established, the first instance knows a newer version is running, and it transfers necessary data. When finished transferring, it notifies the supervisor it is ready to shut down. The supervisor kills the old version of the application and the second instance recreates sockets and connections, and continues running.

Because the AEOS Java application is already running inside a JVM, the advantage of this can be taken. To change from the old version of the software to a newer one, it is possible to run both versions simultaneously. If the old software version detects the new version, it transfers the configuration and authorisations to remove the requirement for fetching initialisation data from the server. If all the data is successfully transmitted, the old JVM can quit, and the new JVM can establish all the necessary connections and continue running. Because the old application can send everything during the start of the new application, always one of the instances is in control. This reduces the downtime significantly, and only a small amount of downtime is left when the new instance establishes connections after the old application stopped.

Some software modifications are necessary to use this technique. First, both applications are required to look for other instances of the application. If the update starts a second application, the first application must establish a connection. Secondly, the first instance must transfer the data via this connection, and finally, the first instance closes the socket and shuts down. When the first closes the connection, the second instance can initialise to continue running. This initialisation consists for example of establishing connections to the server and readers.

In Figure 9.2, the process of handing over is visualised. The second application starts simultaneously but halts after initialisation. The old application notices the new instance and begins transferring necessary data. After transferring, the old instance quits, and the updated version establishes all necessary connections and continues the program execution. Because the execution of this approach depends on the application, timings of the data transfer and downtime due to re-initialisation cannot be determined in front. Because transferring data occurs during the execution of the old application, the timing is not interesting for downtime. However, the initialisation of interfaces introduces downtime, which is dependent on the number of initialisations required and the execution time it takes to initialise.

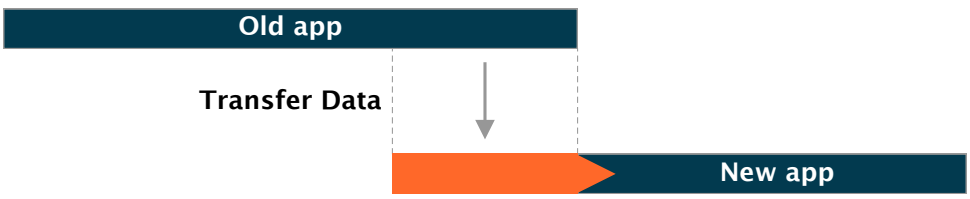



Figure 9.2: hand-over update strategy

## 9.2 Other research topics

While a new update technique to update the Java application further reduces the downtime, several other improvements and techniques emerge during this research. This section describes these topics which should be considered in further research and development.

- Implementation of a reset functionality to erase all configuration and data to restore a fresh installation. By using a partition for data and one for the configuration, completely erasing those partitions should be sufficient.
- Perform incremental updates of the file system instead of flashing or copying a full new file system over an old one. By using the differences between versions, an update only includes the affected files. This requires less bandwidth, which enables big customers to speed up the rollout. Besides it also decreases the flash degradation because less write cycles per block are required.
- Introduce persistent memory over a partial reboot to eliminate the time required for fetching checkpoint files from the flash. This decreases the downtime of a checkpoint and restores due to removing the necessity of writing to and reading from the flash. Important to investigate is the required memory for saving a checkpoint and the requirements for booting the system.
- The incremental checkpoint and lazy restore are already mentioned in Chapter 6 but cannot be used on the current hardware. Developing future products should include research on the ability to support these functionalities. By using incremental dumps and lazy restores, the downtime of the checkpoint and restore process decreases, which decreases the overall downtime of an update.
- Analyse the usage of CRIU and Kexec in case of an update failure to speed up the reboot into the last working partition. In the proposed solution in this research, the full system is rebooted using the conventional method, including usage of a bootloader and fully restart the Java application. The possibility exists to skip these phases to decrease the downtime of a failed update.
- Analyse the ability to use Mender to update the controllers. Mender is an open source update manager for embedded devices. While Nedap Security Management tends to go for cloud-based products, the process of updating the controllers change. By using an update manager with support of secure image transfer, multiple partitions and one management server, the process of updating cloud-connected controllers is simple to manage and fail-safe. Besides Mender, Nedap should consider possible other update management techniques.
- By introducing new hardware for the controller, the ability to use multiple cores should be researched. By using multiple cores, the update process should take advantage of this by, for example, using multiple virtual machines to switch between versions of software without noticeable downtime.

# Chapter 10

# Conclusion

Currently, Nedap Security Management uses a straightforward update process which downloads the update, checks the files, stops the application, overwrites all files and reboots into the new system. After a full reboot, the application starts again, including fetching all authorisations from the server and initialising the connected hardware. Due to several trends in the access control market and the urge to update the Linux environment to a newer version to support new features, the update process must be improved to guarantee high availability of the AEOS controllers. This research explored the design of a new update process to improve the current update process and guarantee high availability using as less downtime as required and introduce fail-safe measures to remain functional over a failed update. Hence, the research question: Is it possible to combine existing update techniques to improve the controller's availability and perform a full device update within a contiguous application downtime of half a second?

To answer this question, the update process currently used by Nedap is analysed. Besides the phases of the update process, this also resulted in insight into timings of each phase and the downtime of the overall update process. As a result of this part of the research, the update process is divided into five phases. These five phases provide a full update including overwriting the application, rebooting the system and starting the application. The downtime of these five phases is at least 3 minutes but depending on the configuration and number of authorisations to fetch from the server, the downtime can take up to 23 minutes.

A critical part of the old update process is the kernel update which requires a full reboot. During this reboot, the old kernel stops and the new kernel takes over control. To overcome this full reboot, this research introduces an investigation into kernel update techniques without a full reboot. An DSE is performed to compare seven different update techniques and, in the end, Kexec is the most valuable fit for the new update process. Kexec starts the new kernel directly from the currently running one while skipping the bootloader during reboot. The most beneficial features are that Kexec already supports the AEpu hardware, is open-source and updates a full kernel instead of patching the kernel which is not always possible and if it is possible it adds code to the kernel.

A second important observation from the old update process is the restart of the Java application, which can take up to 23 minutes. To make sure the application can resume its process after a reboot without initialisation, research to checkpoint/restore techniques is performed. This research analysed five techniques, which all create a checkpoint of an application and can restore it on demand. Conclusively, CRIU is the most beneficial one due to its ability to also checkpoint sockets and FDs. Unfortunately, it did not support the controller architecture yet. By introducing swap instructions instead of the exclusive load and store, the code is changed and ported to the ARMv5 architecture. Additionally, memory barriers are added to ensure the compiler does not interchange instructions in atomic functions to improve performance. Making sure the application can always be restored even with a taken PID, the PID namespace is introduced. The PID namespace is a Linux feature to isolate the ID of a process from the rest of the system. The namespace itself gets an available PID from the system and inside the namespace, all PIDs starts from 1 again and therefore a restored process can always take the same PID as it had before.

Finally, a full new update process is introduced consisting of five phases to download and check the update, load the new kernel with Kexec, create a checkpoint of the application, reboot into the new kernel and restore the application afterwards. To ensure the system even remains functional over a failed update, two additional techniques are implemented. The first technique is using multiple partitions to separate the currently working code and the updated code. This ensures the system is always able to revert to the last working system, and additionally, it also removes the requirement to stop the application before writing the update to its destination. The second technique is the watchdog, which is an automatic reset triggered on a counter. With carefully resetting the timer, the watchdog can detect system hangs and other problems. When for example, the system does not boot correctly, or the application cannot start, the watchdog triggers a reset, and by using the bootloader-environment, it ensures the last working system is booted.

The newly implemented update process is simulated in QEMU, a system emulation tool, to compare the speed up between the old approach and the new process. Because not all parts of the system can be simulated, a deviation in results occur. Therefore a pessimistic and optimistic bound is introduced. Conclusively, the new update process is executed with a downtime of 13.8 seconds, which is a speed up between factor 5.6 and 11 compared to the old update process with downtime between 77.5 to 151.3 seconds.

By implementing this new update approach, the system can update within seconds and with fail safety measures. Only the application running on the system is not updated with this new approach. To update the Java application with minimal downtime, more insight is required. Two possible techniques are therefore proposed. The first one is DSU, which patches the running application. The second technique runs two instances of the application simultaneously to switch from the old version to the new version. Additionally, several future topics are proposed, which requires more research. These topics include incremental updates, incremental checkpoints, lazy restore and persistent memory over a reboot. Before Nedap investigates these topics, the new update process proposed in this research, needs to be tested and adjusted to the hardware and configuration of the AEOS controller. With this implementation, the downtime of the update process can be reduced to seconds, and the number of devices returned due to a failed update reduces.

# Bibliography


[1] M. Ahmad. “Reliability Models for the Internet of Things: A Paradigm Shift”. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 2014. DOI: 10.1109/ISSREW.2014.107.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”. In: *2009 IEEE International Symposium on Parallel and Distributed Processing* (2009).

[3] Jason Ansel, Michael Rieker, and Gene Cooperman. “Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux”. In: (2006).

[4] ARM. *ARM11 MPCore Processor Technical Reference Manual - Cache Operations Register*. URL: https://developer.arm.com/docs/ddi0360/e/control-coprocessor-cp15/register-descriptions/c7-cache-operations-register (visited on 05/08/2019).

[5] ARM. *MCR, MCR2, MCRR, and MCRR2*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204h/Cihfifej.html (visited on 05/08/2019).

[6] ARM. *Migrating a software application from ARMv5 to ARMv7-A/R ¿ Replacing ARMv5 barriers with equivalent ARMv7*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/BABDIAGA.html (visited on 05/09/2019).

[7] Jeff Arnold and Frans Kaashoek. “Ksplice: Automatic Rebootless Kernel Updates”. In: *Proceedings of the 4th ACM European conference on Computer systems* (2009).

[8] Jeffrey Brian Arnold. *Ksplice: ARM support*. 2008. URL: https://lkml.org/lkml/2008/9/13/10 (visited on 03/26/2019).

[9] Hans-Joachim Baader. *KernelCare verspricht Patches ohne Neustart*. URL: https://www.pro-linux.de/news/1/21062/kernelcare-verspricht-patches-ohne-neustart.html (visited on 01/07/2019).

[10] Balena. *How we updated a drone while flying - and how you can too!* URL: https://www.balena.io/blog/how-we-updated-a-drone-while-flying-dockercon2016/ (visited on 12/03/2018).

[11] A Barak, A Shiloh, and R Wheeler. “Flood prevention in the MOSIX load-balancing scheme”. In: *IEEE Computer Society Technical Committee on Operating Systems Newsletter* (1989).

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: USENIX Association, 2014.

[13] Andrew Baumann et al. “The Multikernel: A New OS Architecture for Scalable Multicore Systems”. In: ACM, 2009.

[14] Gilad Ben. “Building Murphy-compatible embedded Linux systems”. In: *Linux Symposium* (2005).

[15] Liming Chen and Algirdas Avizienis. “N-version programming: a fault-tolerance approach to reliability of software operation”. In: *Proceedings - Annual International Conference on Fault-Tolerant Computing* (1978).

[16] CloudLinux. *KernelCare*. URL: https://www.kernelcare.com (visited on 01/07/2019).

[17] CloudLinux. *Supported distributions and kernels*. URL: https://www.kernelcare.com/supported-distributions-and-kernels/ (visited on 03/26/2019).

[18] Linux Plumbers Conference. “Kernel Patching Microconference Notes”. In: 2014. URL: https://blog.linuxplumbersconf.org/2014/wp-content/uploads/2014/10/LPC2014_LivePatching.txt (visited on 01/07/2019).

[19] Christopher Covington. *CRIU possible on ARMv5?* URL: https://lists.openvz.org/pipermail/criu/2014-July/014958.html (visited on 02/18/2019).

[20] Christopher Covington and Pavel Emelyanov. *[CRIU] CRIU possible on ARMv5?* 2014. URL: https://lists.openvz.org/pipermail/criu/2014-July/014955.html (visited on 01/03/2019).

[21] CRIU. *Checkpoint Restore*. URL: https://criu.org/Checkpoint/Restore (visited on 01/07/2019).

[22] CRIU. *CRIU*. URL: https://criu.org/Main_Page (visited on 01/04/2019).

[23] CRIU. *OpenVZ*. 2017. URL: https://criu.org/OpenVZ (visited on 01/03/2019).

[24] CRIU. *Userfaultfd*. 2018. URL: https://criu.org/Userfaultfd (visited on 05/21/2019).

[25] Christoffer Dall and Jason Nieh. “KVM for ARM”. In: (2010). URL: https://systems.cs.columbia.edu/archive/pub/2010/07/kvm-for-arm/.

[26] Debian. *ArmEabiPort*. 2018. URL: https://wiki.debian.org/ArmEabiPort (visited on 01/07/2019).

[27] Hanne Decre. *Toegansbadges hotels, politiecommissariaten, bedrijven of steden gemakkelijk te kopieren*. 2019. URL: https://www.vrt.be/vrtnws/nl/2019/05/02/toegangsbadges/ (visited on 04/20/2019).

[28] die.net. *watchdog(8) - Linux man page*. URL: https://linux.die.net/man/8/watchdog (visited on 05/06/2019).

[29] DMTCP. *DMTCP: Distributed MultiThreaded CheckPointing*. 2017. URL: http://dmtcp.sourceforge.net/FAQ.html (visited on 01/07/2019).

[30] DOULOS. *Implementing Semaphores on ARM processors*. URL: https://www.doulos.com/knowhow/arm/Hints_and_Tips/Implementing_Semaphores/ (visited on 05/19/2019).

[31] Jason Duell. *The design and implementation of Berkeley Lab’s linux checkpoint/restart*. 2005.

[32] Denx software engineering. *Boot Count Limit*. URL: https://www.denx.de/wiki/view/DULG/UBootBootCountLimit (visited on 04/05/2019).

[33] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr. “Exokernel: An Operating System Architecture for Application-level Resource Management”. In: *SIGOPS Oper. Syst. Rev.* (1995).

[34] Cristiano Giuffrida, Anton Kuijsten, and Andrew Tanenbaum. “Safe and Automatic Live Update for Operating Systems”. In: (2013).

[35] Vivek Goyal and Maneesh Soni. *Documentation for Kdump - The kexec-based Crash Dumping Solution*. URL: https://www.kernel.org/doc/Documentation/kdump/kdump.txt (visited on 12/03/2018).

[36] Tianxiao Gu et al. “Low-disruptive dynamic updating of Java applications”. In: *Information and Software Technology* (2014).

[37] Red Hat. *Kernel Administration Guide*. 2018. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/kernel_administration_guide/#chap-Documentation-Kernel_Administration_Guide-Working_With_Kpatch (visited on 02/26/2019).

[38] Red Hat. *Red Hat Enterprise Linux*. URL: https://access.redhat.com/ecosystem/search/#/ecosystem/Red%20Hat%20Enterprise%20Linux?category=Server (visited on 03/26/2019).

[39] Petr Hosek and Cristian Cadar. “Safe Software Updates via Multi-version Execution”. In: 2013, pp. 612–621. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486869.

[40] Texas Instruments. *RS-485 Reference Guide*. 2014. URL: http://www.ti.com/lit/sg/slyt484a/slyt484a.pdf (visited on 06/06/2019).

[41] Saurabh Kadekodi. “Compression in Checkpointing and Fault Tolerance Systems”. In: *ACM Transactions on Embedded Computing Systems* (2013). URL: http://saurabhkadekodi.github.io/checkpoint_compression.pdf.

[42] Chris Kanaracus. “Oracle buys Ksplice for Linux ’zero downtime’ tech”. In: (2011).

[43] Sanidhya Kashyap et al. “Instant OS Updates via Userspace Checkpoint-and-Restart”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 605–619. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kashyap.

[44] kernel.org. *Kernel-provided User Helpers*. URL: https://www.kernel.org/doc/Documentation/arm/kernel_user_helpers.txt (visited on 05/09/2019).

[45] kernel.org. *The Userspace I/O HOWTO*. 2006. URL: https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html (visited on 05/09/2019).

[46] Berkeley Lab. *BLCR Frequently Asked Questions (for version 0.8.5)*. URL: https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html (visited on 01/07/2019).

[47] Canonical Ltd. *Linux Containers*. URL: https://linuxcontainers.org (visited on 01/03/2019).

[48] Xavier Merino. *CRIU 3.7 on Raspberry Pi 3: Can”t dump*. URL: https://github.com/checkpoint-restore/criu/issues/446 (visited on 05/21/2019).

[49] Niall Murphy and Michael Barr. “Watchdog Timers”. In: (2001). URL: http://homepage.cem.itesm.mx/carbajal/Microcontrollers/ASSIGNMENTS/readings/ARTICLES/murphy01_watchdog_timers.pdf.

[50] Hariprasad Nellitheertha. *Reboot Linux faster using kexec*. 2004. URL: https://web.archive.org/web/20130121033946/http://www.ibm.com/developerworks/linux/library/l-kexec/index.html (visited on 01/08/2019).

[51] OpenVz. *Main page*. 2018. URL: https://wiki.openvz.org/Main_Page (visited on 01/03/2019).

[52] *A Technique for Dynamic Updating of Java Software*. 2002.

[53] John Ousterhout et al. “The Sprite Network Operating System”. In: (1988).

[54] Andy Pfiffer. “Reducing System Reboot Time With Kexec”. In: (2003).

[55] Yocto project. *GETTING STARTED: THE YOCTO PROJECT® OVERVIEW*. 2018. URL: https://www.yoctoproject.org/software-overview/ (visited on 05/19/2019).

[56] Mimitri Reijerman. *Mifare-chips eenvoudig volledig te kraken*. 2008. URL: https://tweakers.net/nieuws/52381/mifare-chips-eenvoudig-volledig-te-kraken.html (visited on 04/20/2019).

[57] Ashish Sharma. *Operating System — Semaphores in operating system*. URL: https://www.geeksforgeeks.org/semaphores-operating-system/ (visited on 05/19/2019).

[58] Maxim Siniavine and Ashvin Goel. “Seamless kernel updates”. In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013).

[59] SUSE. *SUSE Linux Enterprise Live Patching*. URL: https://www.suse.com/nl-nl/products/live-patching/ (visited on 03/26/2019).

[60] SUSE. *SUSE Linux Enterprise Server 12 SP4 Administration Guide*. 2018. URL: https://www.suse.com/documentation/sles-12/singlehtml/book_sle_admin/book_sle_admin.html#pre.sle (visited on 01/07/2019).

[61] QEMU team. *Translator Internals*. 2016. URL: http://people.redhat.com/pbonzini/qemu-test-doc/_build/html/topics/Translator-Internals.html (visited on 06/12/2019).

[62] Ken Terada and Hiroshi Yamada. “Shortening Downtime of Reboot-Based Kernel Updates Using Dwarf”. In: *IEICE Transactions on Information and Systems (IEICE T INF SYST)* (2018).

[63] Bruce Walker et al. “The LOCUS Distributed Operating System”. In: *SIGOPS Oper. Syst. Rev.* (1983).

[64] Karim Yaghmour, Jonathan Masters, and Gilad Ben. *Building Embedded Linux Systems, 2Nd Edition*. Second. O’Reilly Associates, Inc., 2008.

[65] Hiroshi Yamada and Kenji Kono. “Traveling Forward in Time to Newer Operating Systems Using ShadowReboot”. In: *SIGPLAN Not.* (2013). URL: http://doi.acm.org/10.1145/2517326.2451536.

# Appendices

# Appendix A

# Comparison tables

Table A.1: Comparison of kernel update techniques

| | **Ksplice** | **kGraft** | **Kpatch** | **KernelCare** | **Kexec** | **ShadowReboot** | **Dwarf** |
|---|---|---|---|---|---|---|---|
| Architecture | IA32, x86-64, SPARC, ARM | x86-64, IBM Z, IBM Power, AArch64 | x86-64, IBM Z, IBM Power, AArch64 | All | i386, x86-64, ppc64, ia64, ARM, AArch64 | x86-64 | x86-64 |
| Linux distribution | Oracle Linux 7 | SUSE Enterprise 12 | Red Hat Enterprise Linux 7 | All | All | Gentoo, Fedora, Cent, Ubuntu, and SUSE | Ubuntu |
| Kernel | 2.6 | 4.0 | 4.0 | 2.6.18 | 2.6 | 2.6.21 | 2.6.39 |
| Patching | Yes | Yes | Yes | Yes | No | No | No |
| Stops processes | Yes | No | Yes | ? | Yes | Yes | Yes |
| Reboot required | No | No | No | No | Yes (partial) | No | No |
| Virtual Machine | No | No | No | No | No | Yes | Yes |

Table A.2: Comparison of persistent application state techniques

| | **DMTCP** | **BLCR** | **CRIU** | **OpenVZ** | **LXC** |
|---|---|---|---|---|---|
| Architecture | x86, x86-64, ARM | x86, x86-64, PPC(32), ARM | x86-64, ARM, AArch64, PPC64 | x86, x86-64, IA64, ARM, SPARC | x86, x86-64, ppc(64), ARM, AArch64 |
| Kernel | 2.6 | 2.6 | 3.11 | 3.11 (OpenVZ) | 3.11 |
| Run without pre-loading | No | No | Yes | Yes | Yes |
| TCP sockets | Yes | No | Yes | Yes | Yes |
| UNIX sockets | Yes | No | Yes | Yes | Yes |
| Shared resources | Yes | No | Yes | Yes | Yes |
| Multiprocess | Yes | Yes | Yes | Yes | Yes |
| Containers | No | No | Yes (OpenVZ and LXC) | Yes | Yes |

# Appendix B

# Script

This appendix shows the instant update script. It is implemented in shell conform the other scripts used within Nedap Security Management.

It first sets up a new directory tree for the update files and log files. Thereafter it checks the architecture of the device and optional subtypes. After checking the new kernel file using MD5-hashes, it checks if all required applications are installed on the system and which partition is currently in use.

After all checks, it mounts and flashes the second partition with the new file system and creates a directory for application checkpoint. Now the system is fully set up, Kexec loads the new kernel with corresponding kernel parameters on line 93. On line 95 the script searches the PID of the application and uses this to create a checkpoint with CRIU. The *criu-ns* command is an executable which automatically uses PID-namespaces as described in chapter 6.

After the checkpoint, old files are removed and it copies last modified files to the second partition. After a sync and an unmount, the system is partially rebooted by Kexec.

```
1  #!/bin/sh
2
3  # load handy functions
   . /etc/functions
5  . /etc/sysfunctions
6
7  ARCH=`getarch`
   #SUBTYPE=
9  IMAGE="$1"
   TARGET_DEVICE="/dev/vma"
11 SOURCE_IMAGE="/images/$IMAGE"
12
13 if [ -d /update ]; then
   rm -Rf /update;
15 fi
16
17 mkdir /update /update/log /update/img
18
19 # we currently only support ax8008 architecture
   [ "$ARCH" == "ax8008" ] || { exit 0 ; }
21
   # get optional ax8008 subtype
23 SUBTYPE=`getax8008subtype`
   if [ "$SUBTYPE" == "unknown" ]; then
25   echo "Unknown ax8008 board subtype!"
     exit 1
27 fi
28
29 if [ $# -ne 1 ]; then
   echo "Usage: $0 <kernel-image>"
31 echo "e.g.: $0 zImage.bin"
```

```
exit 1;
fi

echo " "
echo "Checking ${SOURCE_IMAGE} - $(date)"
echo " "

# check the checksum of the image file for upload consistency

if ! md5sum -c /images/"$IMAGE${MD5_PREFIX}".md5; then
echo "Source '${SOURCE_IMAGE}' checksum failure!"
exit 1
fi

echo " "
echo "Checking if programs are installed"

if ! criu_loc="$(type -p "criu-ns")" || [[ -z $criu_loc ]]; then
echo "ERROR:  -> CRIU not found!"
exit 1
else
echo "  -> CRIU installed"
fi

if ! kexec_loc="$(type -p "kexec")" || [[ -z $kexec_loc ]]; then
echo "ERROR:  -> Kexec not found!"
exit 1
else
echo "  -> Kexec installed"
fi

echo " "
echo "Check which partition is currently in use"
partition="$(awk '/^root/ {print $1;}' /proc/cmdline)"
partition=${partition:5}
echo "Partition in use: '${partition}'!"
echo " "

if [ "$partition" = "/dev/vda" ]; then
new_partition="/dev/vdb"
else
new_partition="/dev/vda"
fi

# Do flashing thing here!!!

# Make filetree on new partition
if ! [ -d /second ]; then
mkdir /second
fi
mount ${new_partition} /second

if [ -d /second/update/ ]; then
rm -Rf /second/update/;
fi

mkdir /second/update/ /second/update/log /second/update/img

# Load new kernel in memory
cmdline="$(cat /proc/cmdline)"

kexec -l ${SOURCE_IMAGE} --append="root=${new_partition} rw highres=off console=ttyS0 mem=256M
     ip=dhcp console=ttyAMA0,115200 console=tty" > /dev/null 2>&1 # load the kernel with the
    correct parameters

# Get PID of running application
if ! ps axf | grep test | grep -v grep | awk '{print $1}'; then
echo "ERROR: Application to dump not found, update kernel anyway!"
else
pid="$(ps axf | grep test | grep -v grep | awk '{print $1}')"
```

```
echo "PID of application is '${pid}'"
echo " "

# Dump application to new partition
criu-ns dump -t "${pid}" -D /update/img/ -vvv -o /update/log/dump.log > /dev/null 2>&1

if ! grep "Dumping finished successfully" /update/log/dump.log; then
echo "ERROR: Criu dump of application '${pid}' failed!"
exit 1
fi

echo "CRIU dump successful"
echo " "
touch /update/DUMPED
fi

# Remove old files
if [ -f /second/home/root/test.log ]; then
rm -f /second/home/root/test.log
fi
# Copy last application and logs
cp -rf /home/root/test.log /second/home/root/test.log
cp -rf /update/ /second/

sync

sync # sync all of the disks so as not to lose data
umount -a # make sure all disks are unmounted
kexec -e # reboot the kernel
```

Restoring is automatically fired on a system reboot. The script checks the CRIU application and the checkpoint files and subsequently restores the application. If a restore fails or a checkpoint is not found, this script automatically fires the application from scratch.

```
#!/bin/sh

SOURCE_UPDATE="/update/"
SOURCE_IMAGE="/update/img/"
SOURCE_LOG="/update/log/"

if ! criu_loc="$(type -p "criu-ns")" || [[ -z $criu_loc ]]; then
echo "ERROR: CRIU not found!"
exit 1
fi

if [ -f ${SOURCE_UPDATE}DUMPED ]; then
# Restore found, so continue restoring
criu-ns restore -D ${SOURCE_IMAGE} -vvv -o /update/log/restore.log

if grep "Restoring FAILED" /update/log/restore.log; then
echo "ERROR: Criu restore of application failed!"
exit 1
fi

echo "CRIU restore successful"
touch ${SOURCE_UPDATE}RESTORED

else
echo "No restore found!!"
setsid /home/root/test.sh < /dev/null &> /home/root/test.log
fi

echo "Kernel update finished successful at '$(date)'"
```

# Appendix C

# Linux build

A new Linux system is required including Kexec and CRIU, to be able to test the new update approach. This new build includes Linux kernel version 4.19 which is the latest LTS Linux version and a poky distribution built with Yocto. This appendix first explains the Yocto project and thereafter describes the steps required to reuse the environment made for this research.

## C.1 Yocto

Yocto is a project which bundles open source projects to create Linux systems regardless of the hardware underneath. It includes tools to share software layers, configuration tools to customise the build and a compile tool to create the system automatically including bootloader, kernel and file system. Applications can be installed and added to the file system upon request such as CRIU and Kexec for this research.

The project makes use of layers and receipts. Each layer can contain several receipts which are the applications. For example, the layer Python implements the dependencies to run Python applications on the system. Additionally, in the overall configuration hardware can be set to build for a specific board or processor. Switching between platforms is as easy as changing the configuration and compiling. For each image a list of applications is present to determine the applications of the final system. Figure C.1 shows the steps to build a system. With the source, user configurations and hardware configurations, Yocto builds a package feed to combine it finally in an image. Important to note is that the Yocto project automatically performs all steps if the user provides all required configurations and files.

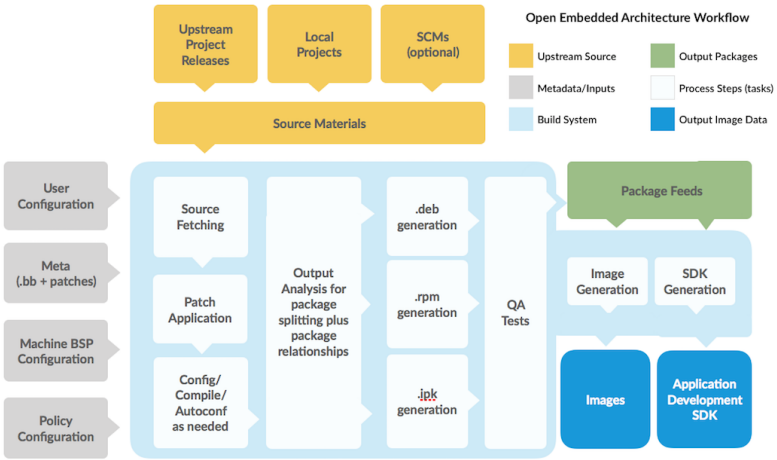



Figure C.1: Yocto process [55]

## C.2 How to use

To speed up the process of creating a new image for the AEOS controller and to test the implementation of this research, a full environment is developed and can be reused or appended for further use. The compilation of the system is performed in a docker to be independent of the compilation system. Figure C.2 shows the required steps to compile a Linux file system including the kernel.

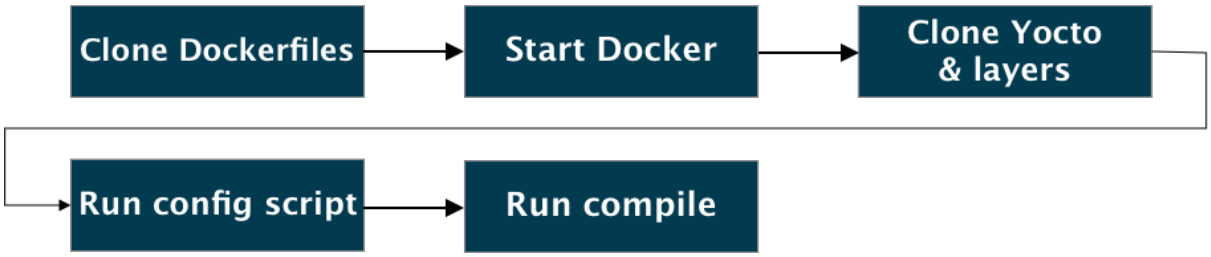



Figure C.2: Steps to build Linux kernel and file system

The first step is to clone the files to create a docker. If a docker is not desirable, step 1 and 2 can be skipped but additional changes in configuration can break the compilation in further steps. This repository consists of a dockerfile which creates a new docker with the required tools and the scripts to flash the new Linux file system to the hardware.

The second step is to start the newly created docker, it starts a console on the new system which is currently a Ubuntu 16.04 distribution.

The third step is to clone the Yocto project and the required layers for our system. This also includes the `nedap-ax8008` layer which contains the ported CRIU application and the configuration files for the hardware.

After everything is set up correctly, the user must run the configuration script to copy the configuration files to the required locations and to set up the environment.

The last step is to run the compile script which automatically download necessary tools and compiles a complete file system including the Linux kernel.