Goethe Universität Frankfurt

Institut für Informatik

Infrastruktur und Rechnersysteme in der Informatik

Master Thesis

Analysis of Security Isolation Technologies for HEP-Computing

Supervisor:
Prof. Dr. Udo Kebschull

Student:
Daniel Bilanović

21.12.2017

Abstract:

Virtual machines are for the most part not used inside of high-energy physics (HEP) environments. Even though they provide a high degree of isolation, the performance overhead they introduce is too great for them to be used. With the rising number of container technologies and their increasing separation capabilities, HEP-environments are evaluating if they could utilize the technology. The container images are small and self-contained which allows them to be easily distributed throughout the global environment. They also offer a near native performance while at the same time aproviding an often acceptable level of isolation. Only the needed services and libraries are packed into an image and executed directly by the host kernel. This work compared the performance impact of the three container technologies Docker, rkt and Singularity. The host kernel was additionally hardened with grsecurity and PaX to strengthen its security and make an exploitation from inside a container harder. The execution time of a physics simulation was used as a benchmark. The results show that the different container technologies have a different impact on the performance. The performance loss on a stock kernel is small; in some cases they were even faster than no container. Docker showed overall the best performance on a stock kernel. The difference on a hardened kernel was bigger than on a stock kernel, but in favor of the container technologies. rkt showed performed in almost all cases better than all the others.

Contents

1 Introduction 7
1.1 Motivation . . . 8
1.2 Goal of the Thesis . . . 8

2 Basics 8
2.1 Virtualization . . . 8
2.2 Security Features . . . 10
2.2.1 Capabilities . . . 10
2.2.2 Application Armor . . . 11
2.2.3 Control Groups . . . 11
2.3 Separation . . . 12
2.3.1 Chroot . . . 12
2.3.2 FreeBSD Jails . . . 12
2.3.3 Linux-VServer . . . 13
2.3.4 Linux Namespaces . . . 13
2.4 Container . . . 15
2.5 Differences between Virtual Machines and Containers . . . 15
2.6 Application hardening . . . 16
2.7 Kernel hardening with grsecurity and PaX . . . 17
2.8 High-Energy Physics . . . 18

3 Methodology 19
3.1 Used hardware . . . 19
3.2 Used software . . . 19
3.2.1 Kernel patching . . . 20
3.2.2 CERN software . . . 21
3.2.3 Un-containered Simulation . . . 22
3.2.4 Docker . . . 23
3.2.5 rkt . . . 26
3.2.6 Singularity . . . 28
3.3 Test procedure . . . 31
3.4 Test setup . . . 31

4 Results 33
4.1 Results . . . 34
4.2 Discussion . . . 34

5 Conclusion and Future Work **41**
5.1 Conclusion 41
5.2 Future Work 42

List of Figures

1 Architecture of a Type-1 and Type-2 hypervisor. 9
2 Setup of a container environment and a virtual machine environment. 16
3 Comparison of runtimes without a container, Docker, rkt and Singularity on a stock kernel, in seconds. 35
4 Comparison of runtimes without a container, rkt and Singularity on a hardened kernel, in seconds. 36
5 Average runtimes of the un-containered simulation on a stock and a hardened kernel, in seconds. 37
6 Average runtimes of the simulation inside Docker on a stock kernel, in seconds. 38
7 Average runtimes of the simulation inside rkt on a stock and a hardened kernel, in seconds. 39
8 Average runtimes of the simulation inside Singularity on a stock and a hardened kernel, in seconds. 40

List of Tables

1 Average runtimes of all simulations on a stock kernel, in seconds. 34

2 Average runtimes of all simulations on a hardened kernel, in seconds. 35

3 Runtime difference of the container compared to un-containered simulation on a stock kernel. 39

4 Runtime difference of the container compared to un-containered simulation on a hardened kernel. 40

Listings

1 Applying the grsecurity and PaX patches to the kernel and compiling it. 20
2 Patching AUFS and compiling the kernel. 20
3 Content of the cvmfs config file. 21
4 Command to start a simulation inside the PbPbbench-directory. 22
5 Set all environment variables as in the alien environment. . . . 22
6 Command to save starttime with variable container ID. 24
7 Dockerfile to create the testimage. 24
8 Command to create Docker image from a Dockerfile. 25
9 Command to create Docker container with ID 1 from an image. 26
10 Command to save starttime with fixed container ID. 26
11 Commands to create and modify an ACI container from a Docker image. 27
12 Command to start rkt container. 28
13 Singularity recipe to create the testimage. 28
14 Singularity recipe to create the testimage. 30
15 Command to start Singularity image. 31
16 Commands to fix error of too many levels of symbolic links. . 32
17 PaX error message of un-containered simulation. 32
18 Disabling PaX filesystem protection features preventing rkt from starting a pod. 33

1 Introduction

Efficient usage of available resources is an often sought-afte goal, especially in datacenters. The old model, where one physical server had one task, wasted a lot of resources as these servers rarely used all available resources. Additionally they were difficult to scale. If more requests had to be served, another server would have had to be set up. One physical server could host multiple different services, but this would have two major drawbacks. The first one would be the more complicated upgrades and migrations. The system would have to satisfy the dependencies of all services that were running on it. Not only would it need a lot of libraries, some of them would even be needed in different versions. One would need to carefully set the environment up and maintain it. The second one would be the increased security and availability risk of the running services. If one of the services would crash the system, all the other services would also be unavailable. A programming error or a malicous service crashing the system would prevent the other services from functioning as inteded. It would also be able to read, modify or destroy data from the other services. If one of these services got compromized, the attacker would gain access to all data and services running on that machine.

One solution to these problems are virtual machines (VM). Virtual machines contain fully functional operating systems that are completely isolated from the host system and other VMs running on the same host. Even though virtual machines have a lot of advantages, they also have a major drawback: they need a lot of additional resources. A hypervisor is emulating hardware and executing full operating systems and their services. Namespaces (as called in the Linux kernel) are a possible resource-efficient alternative. They isolate processes on a system from one another without the need to emulate hardware. Technologies like Docker, rkt and Singularity make it easy for a user to create and execute a so-called container with the wanted processes and all their dependencies.

Previous researches have compared the performance loss of VMs in regard to containers [18] [15] [53]. This work focuses on the performance difference introduced by the different container technologies compared to an execution without any isolations. Chapter 2 explains the history and technical components that are used to create container. Chapter 3 describes the test setup, the used software and how it was configured and used. Chapter 4 shows the

testresults and the discusses them. And chapter 5 finally concludes this work and proposes possible directions for future research.

1.1 Motivation

Datacenter operator strive to run as many jobs on their available hardware as possible while eliminating, or at least minimizing, the interference of them running on the system. Virtual machines are a great way to separate the jobs and host machines from each other, but they incur a rather big resource overhead, which often is not feasable for the operator. Containers promise an easy and resource saving way to separate processes on a machine, similar to virtual machines, but with fewer security guarantees.

1.2 Goal of the Thesis

The goal of this thesis is to measure the performance difference of container technologies by measuring the time an application needs to finish 1) outside of any container on an unmodified Linux kernel, 2) inside of different containers on an unmodified Linux kernel, 3) outside of any container on a Linux kernel patched with grsecurity and PaX, 4) and inside of the same containers on a Linux kernel patched with grsecurity and PaX. This work tries to determine if the mere packaging of a physics simulation impacts its runtime.

2 Basics

This chapter describes the technologies used in this work.

2.1 Virtualization

Virtual machines can be traced back as far as 1970. [35] IBM presented a system that could run multiple System/360 operating systems on one physical machine. The software that ran these machines was divided into two parts, the Virtual Machine Control Program (VMCP), and the Cambridge Monitor System (CMS). The VMCP provided time-sharing and resource allocation, while the CMS enabled users to control the virtualized host easily over a console typewriter. Every users had their own virtual machine and could not interfere with the others.

Today the software that runs virtual machines is called Virtual Machine Monitor (VMM) or Hypervisor and it completely manages the virtual machines, their resources, is responsible for the clean resource separation between the host and all virtual machines, and translates privileged instructions into less privileged ones (interpreting instructions) if necessary. This strict separation provides great security and reliability. A virtual machine cannot use more system resources than provided by the hypervisor. Therefore it is not possible, as long as the hypervisor has no errors, for one virtual machine to disrupt another or to access its data. The hypervisor has to maintain proper control of the executing VMs and their resources, otherwise they could execute privileged instructions and gain acces to the host and other VMs on that system.

The privileged instructions of the hypervisor are only needed to manage the memory adresses of the running VMs. If this mapping were invisible to software, a hypervisor would have no need for privileged instructions. [5]

Hypervisors are categorized into one of two types, Type-1 and Type-2. A Type-1 hypervisors, like the CMVP from the seventies or newer ones such as ESXi and Xen, run directly on the hardware without the need of an operating system. They can host on the same hardware more virtual machines than a Type-2 because there is no host operating system that uses resources. Type-2 hypervisors, like Oracle VirtualBox, VMWare Player and Linux KVM, run on top of an operating system. Due to the additional abstraction provided by the host operating system they are more portable and easier to implement but have less resources available for the virtual machines. Figure 1 illustrates the difference between a Type-1 and a Type-2 Hypervisor. [41, Chapter 2]

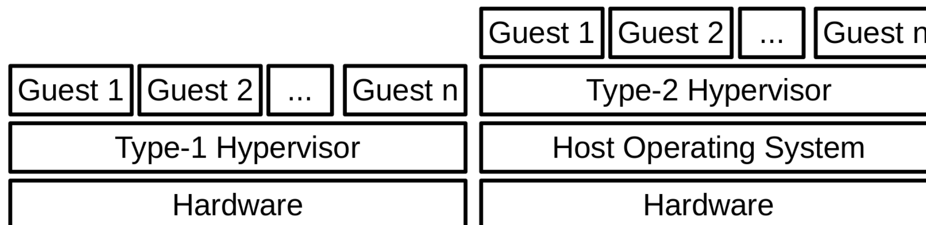

Figure 1: Architecture of a Type-1 and Type-2 hypervisor.

With the reduces hardware costs in the eighties and nineties companies bought more hardware if they needed more resource power, which almost completely eliminated virtual machines except on mainframes. Around the

year 2000 VMWare developed a hypervisor for xommodity x86 processors which enabled IT-Staff to easily create and manage VMs on cheap hardware. This and the increased processing power of the processors (i.e. more cores on one chip and specialized instructions for virtualization) helped virtual machines gain popularity. [6, Chapter 1.5] [41, Chapter 1]

Even though the performance loss from virtualization not as big as it was before, it can for some use case and technologies still be significant. Previous works like [18], [15] and [53] focused their measurements on the difference between virtual machines and container.

2.2 Security Features

2.2.1 Capabilities

The traditional way to check permission on a UNIX system is to compare the user, or rather the user-ID, the process belongs to with the set permissions of the resource the process is trying to access (file, network, etc.). On a UNIX-system there are essentially two kinds of users, privileged (root) and unprivileged (all other) users. Privileged processes bypass permission checks and can do everything on the system, while unprivileged users are subject to all checks. An all-powerful user on a system is comfortable but also a huge risk. If this user gets compromised, an attacker can do everything on that system. Some operations, like listening on a network port, require some privileges. With the traditional permission model that means that the user either has to be root or escalate its privileged with programs like *sudo* [36].

With the kernel version 2.2, Linux began dividing the capabilities of the all-powerful root user into smaller ones. They can be individually assigned and revoked to both users and executable files. This way a normal user can gain some privileges, like listening on a network port or mounting file systems, without the risk of a complete system takeover in case of a malicious or erroneous user or executable file. Up until version 4.9 (used for this work) Linux separated the capabilities into 38 groups (like CAP_CHOWN, CAP_NET_BIND_SERVICE, etc.). Capabilities can be programatically set with the *capset(2)* system call. The new capability-set must be a subset of the previously owned capabilities if the user or executable doesn't have the *CAP_SETCAP* capability. Executing a program sets the capabilities of it to the combination of the users and executables capabilities. The capabilities of a process can be inspected via the */proc/[pid]/status* file. [27]

2.2.2 Application Armor

Application Armor, also called AppArmor, is a mandatory access mechanism that binds access capabilities to files. The allowed files and operations are written into a profile file that can be loaded by the kernel and applied to processes. By default AppArmor blocks everything not explicitly allowed in a profile. Not allowed accesses will generate an entry into the system logfile, and additionally terminate the process if it is configured that way (*enforcement* mode instead of *complain* mode). It's possible to restrict network access, files/directories, system calls, and more [1]. Profiles have to be applied just before the program that should be confined is executed because an already running process cannot be confined [50]. The idea is to specify every path, system call, etc. a program needs to function properly in a profile and to confine the process with it. If that program gets compromized and the attacker tries to execute system calls that were previously not allowed, the process will be terminated and damage to the system or other processes and files will be prevented.

2.2.3 Control Groups

In 2006 a group from Google Inc. announced that they were working on a project to create and manage processes as groups on a Linux system, called *Process Containers* [12, 47]. In 2008 it got merged into the upstream kernel and renamed to *Control cgroups*. [39] It allows the Linux kernel to control the system resources a group of processes can use. I allows to define a maximum amount of resources and, depending on the resource, also guarantee a minimal amount. The following resources can be limited: CPU shares, CPU accounting, CPU core binding, memory usage, devices, freezer, network classid, block devices, perf events, network priorities, huge memory pages, and PIDs. Cgroups can be nested, where all groups inside one parent group cannot exceed the resource usage of that parent, even if the the child cgroups' limits would exceed that limit.

The *CPU share* cgroup allows a minimum and maximum amount of CPU resources to be assigned. Groups can be limited to prevent denial-of-service attacks on the CPU, but also allow a group to get some CPU time even if the whole system is under heavy load.

cpuacct provides accounting for the CPU usage of cgroups.

cpuset binds processes inside a cgroup to a set of CPUs and NUMA

(non-uniform memory access) nodes.

With the *memory* cgroup it is possible to report and limit the usage of process and kernel memory, as well as the swap memory.

The creation and opening of devices for read and write can be restricted with the *devices* cgroup.

The *freezer* cgroup allows the suspension and resumption of processes inside a cgroup and all its direct descendants.

With the help of the *net_cls* cgroup outgoing network packets will be labeled. These labels can be used for firewall rules on the host system.

Access rates to specific block devices can be throttled through the *blkio* cgroup. The rates can be a percentage or an I/O-rate.

perf_event enables the system to perform the integrated perf monitoring of processes inside that cgroup.

net_prio permit cgroups to have priorities assigned to their network interfaces.

hugetlb limits the number of translation lookaside buffers (TLB) a cgroup is allowed to use. A TLB is a cache that stores the translation of a virtual address to the corresponding physical one.

And *pids* limits the number of processes that can be created inside a cgroup. [31]

2.3 Separation

2.3.1 Chroot

Resource isolation began in 1979 on the operating system Unix Version 7. In this release the system call *chroot* appeared for the first time and it allowed to change a running process' view of the filesystem. It changed the global root of the calling process on the whole filesystem structure to the directory called by chroot. The idea is to contain file accesses of processes to a specific part and to prevent one process from reading or manipulating files belonging to another process. A major expansion on that idea were *Jails* on the FreeBSD operating system. [19] [46]

2.3.2 FreeBSD Jails

Jails were introduced with the release of FreeBSD version 4.0 in the year 2000 [20, 43]. Jails expand the isolation idea of chroot by virtualizing file

system access, users, and networking. [44] They are in a way like virtual FreeBSD machines and are used to contain processes on a system in different ways. Every jail can contain a whole FreeBSD system and has a separate root user that can do almost everything inside it. Users and processes inside a jail can not determine if they are inside a jail. Todays jails transform the Namepsaces of the file system, process IDs, and network, and they can limit system resources for each jail. Processes inside a jail only have access to the files in their own file system, they can not see or communicate with processes on neither the host system nor inside other jails on that system, and their CPU, memory, hard disk, and network usage can be restricted (e.g. only one core, max 2GB RAM). Due to these isolations and restrictions it is not possible for a compromized service to disrupt the other services on that system. The namespace isolations (PID, file system, etc.) prohibit direct access, while the resource restrition helps to mitigate denial of service attempts by overusing the CPU, memory, or network bandwidth. Unlike virtual machines, all processes on a system, regardless if they are inside a jail or not, share the same kernel. [34, Chapter 5.9] [45]

2.3.3 Linux-VServer

The first wide-ranging isolation implemented on Linux was *Linux-VServer*, whose development started in the year 2001 [17]. Linux-VServer modifies the kernel to enable or extend resource isolation, like the networking stack, inter process communication, and process IDs; and resource accounting/limiting, like CPU time consumed and maximum memory usage. As in jails, processes inside an isolated environment, here called Virtual Private Server, can not see or interact with processes running on the host or inside other VPS. In the beginning, Linux-VServer implemented their own isolations by patching the lernel [42]. Over the years, the kernel developer implemented some isolation techniques into the mainline kernel called *Namespaces*.

2.3.4 Linux Namespaces

The first namespace, for mounts, was implemented in Version 2.4.19 released in 2002 [29, 38]. Since then other kinds namespaces were implemented to isolate process groups from each other. Currently (Linux version 4.9) there are seven different namespaces: Cgroups, IPC, Network, Mount, PID, User, and UTS. Namespaces change the view of system resources for a process or a

group of processes by restricting access to resources outside of its own scope, similar to Jails and Linux-VServer. Different container technologies leverage them to isolate running container from each other and the host system.

Cgroup namespaces isolate the view of cgroups of a group of processes. When a process creates a new namespace using clone(2) or unshare(2) with the *CLONE_NEWGROUP* flag, it enters a new namespace, where the root directories are set to its current directories. To use cgroup namespaces the kernel has to be configured with *CONFIG_CGROUPS*. Without it, processes inside a container could see the directories outside of it and therefore gain valuable information. Additionally, containers are easier to migrate and confined because they do not need to know the pathnames of the parent cgroups, which allows them to be mounted below any aprent directory. [28]

IPC stands for Inter Process Communication and consists of different techniques processes can use to communicate and exchange data with each other. *IPC namespaces* isolate IPC objects and POSIX message queues in such a way that processes inside of one namespace can neither see nor ineract with processes on the outside or inside other IPC namespaces. [30]

A *network namespace* isolates the network stack and everything associated with it, like network routes, firewall rules, and network devices. By default a process inherits the namespace from its parent process. [30, 4]

Mount namespaces isolate the mountpoints a group of processes can see. When creating a new mount namespace the child namespace has the same hierarchy as the parent namespace, which can then be modified as needed. Mounting or unmounting devices in one namespace doesn't affect the mount status inside of other namespaces. Additionally, parts of the filesystem tree can be shared across multiple namespaces. [29]

PID namespaces isolate the process IDs on the inside from the outside. PIDs in one namespace are unique, but the same PID can be assigned to a process inside another namespace on the same system. The first process inside a namespace has the PID 1 and is therefore the init process. If this process terminates, then all other processes in that namespace will be stopped. Only processes that are visible to a namespace are listed below the */proc* directory. [32]

User namespaces isolate security-related identifiers and attributes like the UID, GID, root directory, and capabilities. A process can have multiple UIDs and GIDs, one inside and one outside of a namespace. That allows a process to be restricted inside the parent namespace, but be unrestricted inside a child namespace. Upon creating a new child namespace, the process can gain capabilities inside that namespace that it doesn't have inside the parent namespace. System-wide operations, like setting the time, creating a device, or loading a kernel module, can only be performed by processes in the root namespace. [33]

UTS namespaces provide isolation for the hostname and the NIS domain name. [30]

2.4 Container

Container on Linux utilize both *namespaces* and *cgroups* for the separation and resource limitation. Docker was the first solution that made it easy to use them. They consist of an image and a program.

The image contains a description of the container and directories that make up a filesystem. Inside these directories are all files, libraries and dependencies the programs running inside a container need. No kernel is packed inside since the processes are running on the host kernel. The processes running inside a container are therefore executed directly by the kernel running on the host system.

The program manages the images and the running container. It reads the image, sets up the needed environment (create directories, apply namespaces and cgroups, start the first process of the image) and manages the running container.

2.5 Differences between Virtual Machines and Containers

Both virtual machines and containers separate groups of processes from each other and the host system, but in different ways. Virtual machines are controlled and separated by a hypervisor, a software layer that manages its resources and completely controls it. Its isolation is therefore stronger, but

comes with an increased resource usage. For containers this separation is happening inside the host kernel. There is no additional software layer between a container and the host sytem. This results in a less sophisticated separation but also in a smaller resource usage.

The missing software layer in the case of containers also poses a security risk. If one of the processes inside a container compromizes the kernel it gains access to all the other containers on that system. There is only one piece of software that needs to be exploitet. In the case of VMs, an attack needs to compromize the guest OS, the hypervisor and then the host system to gain unrestrained access. Furthermore, if an attack from inside a container can crash the kernel, all the other containers on this machine will also terminate. The kernel of a container host can be hardened to reduce the risk of an exploitation.

Figure 2 shows a container and a virtual machine setup. Please note that the container engine is not really sitting between the container and the kernel. It just sets the environment up and starts the containers, it is not executing them. The processes inside the containers are executed directly from the kernel.

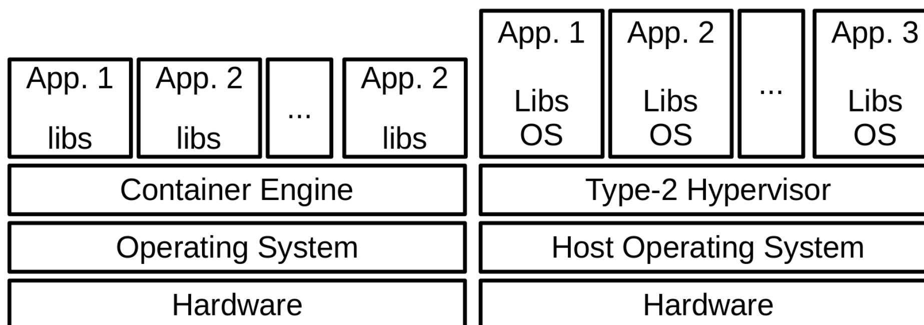

Figure 2: Setup of a container environment and a virtual machine environment.

2.6 Application hardening

Applications on a classical desktop and server operating system, like GNU/Linux and Windows, are mostly unrestricted. They can consume as much CPU and

memory as they want, they can access network resources, and communicate with other processes on the system. On other operating systems, like Android and iOS, access to a lot of resources is restricted by the operating system. Spplications have to request permissions for them, like reading the contact list or accessing the camera. There are a few methods to limit the impact of a malicious or compromized program running on a system: attack surface reduction, dropping unneeded privileges, using the right algorithms for a task (e.g. encryption, digital signing). All of them can be addressed by the program developers. Depending on the program and usage, some can also be tackled from others, such as attack surface reduction and dropping of unneeded privileges.

Let us first examine attack surface reduction. Every software has bugs. The larger a piece of software, the greater the risk of bugs. This risk can be reduced if the application has fewer entry points from the outside, like parsing arbitrary user input, but this has to be done by the developer. If a software is written in a loosely coupled multi-process way the user could deactivate not needed parts of it. Instead of running a bundle with a few different services, one could disable not needed services via a config file. If this is not possible a user could restrict access to that service, for example with a firewall.

The second method is dropping privileges and denying access to unneded resources. With the help of AppArmor, capabilities and others, a user can define files and directories, system calls, etc. that a program is allowed to access. Capabilities split all privileged access of a Linux system into small bits that can be applied to or revoked from processes. These tools empower a user to control the access a process has on the system without the need to modify the sourcecode.

2.7 Kernel hardening with grsecurity and PaX

The kernel of an operating system is the most important and most privileged process running on a machine. It manages the available hardware resources, provides inter process communication, schedules running processes, and does a lot more. Because of it's privilege it is a frequent target of attacks. If it gets successfully compromized, the attacker can execute arbitrary code with the highest privilege on the system, which means that it can do everything

a software is able to do. As previously explained, a VM contains a whole operating system that runs on top of a hypervisor, which in turn runs on the host system. Containers on the other hand do not have that many layers between the untrusted process (inside the container) and the kernel of the host machine, all untrusted processes interact directly with the host kernel. It is therefore the only piece of software that needs to get compromized in order to gain full access to the host machine. There are some techniques to make it harder to compromize the kernel and gain unauthorized access on the system. These can be patches, compile-time options or runtime settings.

Grsecurity, that also contains *PaX*, is the product of a company, Open Source Security, Inc., that provides hardening patches for the Linux kernel [37]. Up until April 2017 the development patches were freely accesible for everyone, with a paid option for support and more stable patches. The patches contained in PaX provide memory safety features. Executable memory is made not-writable to prevent buffer overflow bugs, the address space of processes is randomized to make it significantly harder for an attacker to determine the address of a function he or she wants to execute. Grsecurity introduces different defensive and hardening protections into the kernel: preventing the kernel to execute code that resides in a memory region owned by a user process, greater separation of memory stacks belonging to different processes, bounds checking when copying data from a userland process to the kernel, and a lot of other features. A complete list can be found on their page [16].

They also significantly expand the logging of events. Memoy acces violations, trying to execute blocked system calls and a whole array of other events are logged. These entries could be used to feed an Intrusion Detection System or Intrusion Prevention System and to detect anomalies in running processes.

2.8 High-Energy Physics

High-energy physics, also called particle physics, is a field that studies the fundamental constituents of matter and how particles interact. Particles are accelerated to almost light speed before they collide heads on with other particles also travelling at almost light-speed in the opposite direction. The enegies created by these collisions are measured and stored for later analysis.

This data includes hints at how these particles interact, how correct the standard model of physics is, and what happened just after the big bang. CERN in Geneva, Switzerland is such an institution and also the biggest and most sophisticated accelerator and detector in the world. The analyzation is done on a high-throughput computing (HTC) grid distributed all over the world. HTC aims to provide a reliable infrastructure for a large amount of computing resources that can be used to execute long running jobs, typically over weeks and months of runtime. Jobs running inside these environments are usually not that tightly coupled like the ones executed in high-performance computing environments. [9, 26, 3]

3 Methodology

This chapter describes the used hard- and software, the testcases, and how the measurement was conducted.

3.1 Used hardware

Alle the tests were conducted on a Supermicro X8DAH Mainboard (Version 2.1) with two Intel Xeon E5520 processors for a total of 8 cores and 16 threads, dynamically clocked to up to 2.27GHz. The machine had 12GB of RAM and a Western Digital WD5002ABYS-0 HDD with a capacity of 500GB, 7200 RPM and 16MB Cache. The swap area was set up as a file on the root file system, not as a separate partition. The network interface was an Intel 82576 Gigabit Network card, which is capable of transfering 1Gbps over ethernet.

3.2 Used software

The host machine ran Ubuntu 14.04 LTS with a different kernel (version 4.9 instead of 4.4) but with the default settings and the tools needed to create and run the tests. These were the containers, namely Docker [21], rkt [13], Singularity [48], the *cvmfs* [7] and *PbPbbench* as well as their dependencies. Depending on the test the kernel was patched with AUFS or grsecurity.

3.2.1 Kernel patching

The linux kernel version 4.9 was chosen because it was the latest version with available grsecurity and PaX patches at the time of writing. AUFS, which Docker depends on, was was available for all Linux versions. For half of the testruns a kernel patched with grsecurity and PaX was used. Applying them was straight forward. The Linux sources were obtained from [40] and the grsecurity and PaX patches from [37]. Both were extracted and placed inside the same directory. grsecurity and PaX were applied with the commands show in listing 1. None of the upstream kernel options were changed. grsecurity and PaX can be configured to set a lot of options based on the preferred usage: security or performance. Because the performance impact should be low the performance option was chosen. During the tests occured some errors that could only be resolved by disabling PaX' recomputation of size parameters of function arguments (*CONFIG_PAX_SIZE_OVERFLOW*). As shown in listing 2 AUFS was similarily easy to apply after downloading it from [52].

```
1  cd linux-4.9.23/
2  patch -p1 < ../grsecurity-3.1-4.9.23-201704181901.patch
3  make menuconfig
4  fakeroot make deb-pkg
```

Listing 1: Applying the grsecurity and PaX patches to the kernel and compiling it.

```
1  cd linux-4.9.23/
2  patch -p1 < ../aufs-standalone/aufs4-kbuild.patch
3  patch -p1 < ../aufs-standalone/aufs4-base.patch
4  patch -p1 < ../aufs-standalone/aufs4-mmap.patch
5  cp -r ../aufs-standalone/Documentation .
6  cp -r ../aufs-standalone/fs .
7  cp -r ../aufs-standalone/include/uapi/linux/aufs_type.h
       include/uapi/linux/
8  make menuconfig
9  fakeroot make deb-pkg
10 sudo dkpg -i ../*.deb
```

Listing 2: Patching AUFS and compiling the kernel.

The command *make menuconfig* opens a menu where the user can browse through all the kernel features and enable, disable, or modify them as needed.

AUFS had to be enabled this way as it is not enabled by default. The grsecurity and PaX parameters were also enabled and modified this way. After the kernel compilation is finished, the Debian-packages will be placed inside the parent directory. They can be installed with through the packet manager *dpkg*.

3.2.2 CERN software

CERN provides its software through the CernVM File System, also called *cvmfs*. It is a POSIX-compliant file system in userspace, implemented as a FUSE module on Linux. It is read-only and, unlike other network file systems like NFS, accessed over HTTP. The communication over HTTP allows it to be easily accessed through firewalls. It's main usage is to distribute the software thats needed to reconstruct and simulate the collisions to the other locations of the worldwide-distributed computing infrastructure. The software and environment needed to run the benchmark-software, PbPbbench, is also located on there. Mounting is managed by *autofs*, but it can also be done via *cvmfs_config*. After the repository has been added to the system the cvmfs packages were installed through apt. A config file *default.local* was created inside */etc/cvmfs/* with the content shown in listing 3. For a more detailed explanation on how to install, configure, and debug it, please see [8].

```
1 CVMFS_REPOSITORIES=alice.cern.ch
2 CVMFS_CACHE_BASE=/tmp/cvmfs
3 CVMFS_QUOTA_LIMIT=50000
```

Listing 3: Content of the cvmfs config file.

An important program for these tests is *AliRoot*. It is based on the ROOT framework, also developed at CERN, and used by the ALICE experiment to simulate, reconstruct, and analyze data obtained from collisions. These capabilities are needed by *PbPbbench*, which was used in this work as a benchmark to measure the impact of different container technologies. PbPbbench was chosen as the benchmark because it is a physics simulation and therefore similar to the other jobs usually running in this environment. For more information about ROOT or AliRoot, please see [10] or [11].

3.2.3 Un-containered Simulation

PbPbbench in version v5-05-Rev-16, which was obtained from the CVMFS drive, was used as the benchmarking-software. The exact path was /cvmfs/alice.cern.ch/x86_64-2.6-gnu-4.1.2/Packages/AliRoot/v5-05-Rev-16/test/PbPbbench/. Inside this directory is a script called *runtest.sh* which executes aliroot with *sim.C* and also other files, but only this sim.C was important for the tests. This starts a monte carlo simulation of a collision based on the data saved in the OCDB directory which is also located in the PbPbbench directory. The simulation calculates amongst other things, the generation of particles that form during a collision and their energy deposition and path through the detector [51, Chapter 3.5]. The command to start it is shown in listing 4. To separate the individual testruns from each other every instance copied the PbPbbench directory, including the OCDB directory, to an own separate directory based on the ID received from the calling process. The aforementioned OCDB directory is a database of recorded data from previous events, taken from [24], commit 8a334a30 from July 31st 2015.

```
1 cp -r ~/.alien/tmp/PbPbbench ~/workdir/PbPbbench_${2}
2 cd ~/workdir/PbPbbench_${2}
3 echo "starttime:this=${2}:`/bin/date +%F_%H:%M:%S`" >>
     $LOGDIR/native_resulttime.log
4 aliroot -l -b -q sim.C
5 echo "endtime:this=${2}:`/bin/date +%F_%H:%M:%S`" >>
     $LOGDIR/native_resulttime.log
```

Listing 4: Command to start a simulation inside the PbPbbench-directory.

For aliroot and the simulation to work some environment variables had to be set. The easiest way to do this was to print the environment of the aliroot user, evaluate it inside of the user' shell and to overwrite the variables not containing the correct value, like the path to the OCDB. Listing 5 shows the setting of the environment variables. All the dependencies, except *libgfortran.so*, were installed from the reposiroty. This library was copied from one of the CentOS container to the host system as it could not have been found inside the repository.

```
1 AliRootVersion="v5-05-Rev-16"
```

```
2 eval `/cvmfs/alice.cern.ch/bin/alienv printenv
    VO_ALICE@AliRoot::$AliRootVersion`
3 export ALICE_ROOT_OCDB=~/workdir/PbPbbench_${2}/OCDB
4 ...
```

Listing 5: Set all environment variables as in the alien environment.

The simulation inside of the three containers started after changing into the PbPbbench directory, while the simulation outside of the container had to make an additional step. Before the simulation, and also before the start time was saved, the PbPbbench directory was copied into a new directory before being started from inside that new directory. Every instance of the un-containered execution received its ID as a commandline parameter not only to create this directory but also to log its ID next to its time into the logfile. Just before the simulation started and just after it finished, every instance saved the time and their ID into that file.

3.2.4 Docker

Docker was the first container technology on Linux that gained mainstream popularity. It provides their own tools to ease the creation, modification, execution, and management of container. It uses namespaces and cgroups to isolate the container that are built with a *Dockerfile*. This file contains a base image from which the container inherits its environment (package manager, libraries, etc.) and optionally additional instructions. These instructions are additional steps to extend and customize the base image to the needs of the user. They can be anything like installing new packages, copying files from the host machine inside the container, manipulating files, or enabling and disabling processes on startup.

Besides the namespace isolation and cgroup resource limitation, Docker by default sets capabilities on a container before it is being executed. Processes inside of containers can for example listen on a network port and change ownership of files and directories, but they can't load kernel modules and enable or disable kernel auditing, as well as some other operations. The complete list can be found in [22]. Docker container can additionally be secured with SELinux, AppArmor, or other hardening techniques [23]. Containers are managed by a daemon, *dockerd*, which can receive commands over the network and the *docker* binary on the local machine. The daemon is always running as the root user on the host machine.

The container was created with the *Dockerfile* shown in listing 7. The first entry of a Dockerfile has to be the base image upon which the rest of the Dockerfile builds; in this case centos6. After that the CERN repositories are added, the package database is updated and the needed packages are installed. Next the needed directories are created, the user and group *aliprod* are added and environment variables are set. After that the PbPbbench directory is copied into the container and the default directory and user when starting the container are set. Before copying the PbPbbench directory into the container the runtest.sh script was modyfied to save the right parameter, the container ID, into the logfile.

```
1 echo "starttime:this=1:`/bin/date +%F %H:%M:%S`" >> $LOGDIR
    /docker_resulttime.log
2 aliroot -l -b -q sim.C
3 echo "endtime:this=1:`/bin/date +%F %H:%M:%S`" >> $LOGDIR/
    docker_resulttime.log
```

Listing 6: Command to save starttime with variable container ID.

```
1 FROM centos:centos6
2
3 # Add cern repos
4 RUN curl http://linuxsoft.cern.ch/wlcg/wlcg-sl6.repo -o /
    etc/yum.repos.d/wlcg-sl6.repo
5 RUN curl http://linuxsoft.cern.ch/wlcg/RPM-GPG-KEY-wlcg -o
    /tmp/RPM-GPG-KEY-wlcg
6 RUN rpm --import /tmp/RPM-GPG-KEY-wlcg
7 RUN /usr/bin/yum --enablerepo=*-testing clean all
8 RUN rm /tmp/RPM-GPG-KEY-wlcg
9 RUN rm -rf /var/cache/yum
10
11 # Installing prerequisites.
12 RUN yum update -y
13 RUN groupadd -g 355 aliprod
14 RUN useradd -g 355 -d /var/lib/aliprod -m aliprod
15 RUN yum install -y HEP_OSlibs_SL6
16 RUN yum install -y which
17 RUN yum install -y gcc-gfortran
18 RUN yum install -y redhat-lsb-core-4.0-7.el6.centos.x86_64
```

```
19 RUN sed -i '$ a\export PATH="/cvmfs/alice.cern.ch/bin:$PATH
       "' /var/lib/aliprod/.bashrc
20 RUN sed -i '$ a\export LANG=C' /var/lib/aliprod/.bashrc
21
22 # Create the read only configuration files for
23 # AliEn services and Jobs.
24 COPY opt/.alien /var/lib/aliprod/.alien
25
26 # Create writable directories for the AliEn jobs
27 RUN mkdir /var/lib/aliprod/.alien/cache
28 RUN mkdir /var/lib/aliprod/.alien/logs
29 RUN mkdir /var/lib/aliprod/.alien/tmp
30
31 # Copy simulation files
32 COPY ./PbPbbench/ /var/lib/aliprod/.alien/tmp/PbPbbench
33
34 # Make aliprod the owner of these directories and files
35 RUN chown -R aliprod:aliprod /var/lib/aliprod/.alien
36
37 ENV HOME="/var/lib/aliprod"
38 ENV PATH="/cvmfs/alice.cern.ch/bin:$PATH"
39 ENV LANG=C
40
41 USER aliprod
42 WORKDIR /var/lib/aliprod/.alien/tmp
```

Listing 7: Dockerfile to create the testimage.

An image can be created from a Dockerfile with *docker build* as shown in listing 8. This image is placed inside the Docker directory on the system and it can be managed or started. Before the tests were conducted 10 separate containers were created from the same image. During the tests the pre-created container just had to be started as the on-demand creation of them would have taken additional time. Because Docker by defaults starts the containers with an AppArmor profile, it had to disabled during container creation with the commandline argument *--privileged*.

```
1 docker build --tag tests:pbpbbench .
```

Listing 8: Command to create Docker image from a Dockerfile.

When starting a container, directories from the host system can be mapped into the container and accesses by processes from within. This was used so that the simulations would save their times and IDs into the same file on the host system. Listing 9 shows the command to start a container with the directory /home/daniel/my_times/ on the host mapped to the same directory in the container and without an AppArmor profile and the simulation ID 1.

```
1 docker run --privileged --mount type=bind,source=/home/
    daniel/my_logs,target=/home/daniel/my_logs --name docker
    -1 tests:pbpbbench /var/lib/aliprod/.alien/tmp/PbPbbench
    /runtest.sh 1
```

Listing 9: Command to create Docker container with ID 1 from an image.

3.2.5 rkt

The main focus of rkt is to be used inside cloud-based environments. The tools provided allow *pods* to be started and managed, but not created or modified. One or more images can be packed together to work as one unit. This unit is called a pod and all processes running inside of one pod can communicate with each other and share the same environment, even if they were in different images before being placed into a pod [2]. For the creation and manipulation of images the user has to use other tools like acbuild, or just specify a Docker container which rkt will convert to its native image format before executing. The supported image formats are ACI and OCI. For the tests the ACI image format was used.

The rkt-image was created from the previously created Docker image with *docker2aci*. This program takes a Docker image and converts it to an ACI image, which then was extracted, modified and repacked. The script *runscript.sh* from PbPbbench had to be modified because a rkt container is not able to receive parameters from the calling process like Docker. The only lines that were changed were the output of the times. Instead of the variable output shown in listing 6, the container ID was hard-coded into the command as shown in listing 10. *this=1* was changed to the respective ID for every container image.

```
1 echo "starttime:this=1:`/bin/date +%F_%H:%M:%S`" >> $LOGDIR
      /rkt_resulttime.log
2 aliroot -l -b -q sim.C
3 echo "endtime:this=1:`/bin/date +%F_%H:%M:%S`" >> $LOGDIR/
      rkt_resulttime.log
```

Listing 10: Command to save starttime with fixed container ID.

The repacking was done manually. At first the Docker image was extracted with *docker export* and converted to an ACI image with *docker2aci*, as shown in listing 11. Then it was unpacked with *tar*, wich created a directory *rootfs* and a *manifest* file, and both files were moved into the working directory *workdir* that was previously created. The manifest file describes the container and its settings that are needed for both the startup phase and during execution, and rootfs contains all files of that container, exactly as they are when it runs. workdir/rootfs/var/lib/aliprod/.alien/tmp/PbPbbench/runtest.sh was modified as shown in listing 10. After every modification of runtest.sh a container image was created with tar.

```
1 docker export tests:pbpbbench > dockerimg.tar
2 docker2aci ./dockerimg.tar
3 mkdir workdir
4 tar zxf dockerimg.aci
5 mv rootfs manifest workdir
6 vim workdir/rootfs/var/lib/aliprod/.alien/tmp/PbPbbench/
      runtest.sh
7 tar -pczf rkt_1.aci workdit/manifest workdir/rootfs/
```

Listing 11: Commands to create and modify an ACI container from a Docker image.

rkt also applies some capabilties to a pod before it is started, like denying applications to listen to a port number below 1000, change its capabilities, execute chroot system calls, and others. Currently there are 14 denied capabilities which can be seen at [14]. Additional capabilities can be set or revoked with the *--caps-retain* and *--caps-remove* commandline arguments, or by modifying the *capabilities-retain-set* or the *capabilities-remove-set* files (exact location depends on the distribution).

Just as with Docker, rkt also allows directories from the host system to be

mapped into pods. Listing 12 shows how to start one rkt container with the cvmfs directory mapped into the container and the /home/daniel/my_logs directory on the host mapped to /var/lib/aliprod/my_logs inside the container.

```
1 rkt run --insecure-options=image --volume=log-dir,kind=host
    ,source=/home/daniel/my_logs,readOnly=false --volume=cvm
    -vol,kind=host,source=/cvmfs,readOnly=true --mount
    volume=cfm-vol,target=/cvmfs --mount volume=log-dir,
    target=/var/lib/aliprod/my_logs --exec PbPbbench/runtest
    .sh
```

Listing 12: Command to start rkt container.

3.2.6 Singularity

Singularity was developed to be primarily used inside of HPC-environments. It was designed with HPC-provider to bring the portability, reproducibility, and security of containers into their datacenters. Like Docker, it delivers tools to create and modify container images, as well as to run and manage them. Container can be created interactively, with a script like the others, or a Docker image can be imported and converted to the Singularity image format. Processes running inside a container cannot gain privileges, for example become root, if they don't have that capability on the host machine as well. It can easily be integrated into existing HPC software like a job scheduler (e.g. SLURM, SGE) [25].

It is a known problem that Singularity can't just create CentOS images on an Ubuntu host. But Singularity is able to bootstrap from a Docker image and continue the remaining adjustments. Similar to Docker, Singularity also has a recipe file containing every command needed to setup and create a container. This container was based on the Docker image of CentOS 6 and executed the same commands as Docker. Listing 13 shows this recipe.

```
1 # Base system
2 BootStrap:docker
3 From:centos:6
4
5 # Setup needed environment variables for the simulation
6 %environment
```

```
7       HOME="/var/lib/aliprod"
8       PATH="/cvmfs/alice.cern.ch/bin:$PATH"
9       LANG=C
10
11  # Copy files from host system into the container
12  %files
13      opt/.alien /var/lib/aliprod/.alien
14      ./PbPbbench/ /var/lib/aliprod/.alien/tmp/PbPbbench
15      starttest_aliprod.sh /var/lib/aliprod/testrun.sh
16
17  # Install packages and create directories/user
18  %post
19      # Add cern repos
20      curl http://linuxsoft.cern.ch/wlcg/wlcg-sl6.repo -o /
            etc/yum.repos.d/wlcg-sl6.repo
21      curl http://linuxsoft.cern.ch/wlcg/RPM-GPG-KEY-wlcg -o
            /tmp/RPM-GPG-KEY-wlcg
22      rpm --import /tmp/RPM-GPG-KEY-wlcg
23      /usr/bin/yum --enablerepo=*-testing clean all
24      rm /tmp/RPM-GPG-KEY-wlcg
25      rm --recursive --force /var/cache/yum
26
27      # Installing prerequisites.
28      yum update -y
29      groupadd --gid 355 aliprod
30      useradd --gid 355 --home /var/lib/aliprod --create-home
            daniel --password xx/xxxxxxxxxx
31      yum install -y HEP_OSlibs_SL6-1.0.18-0.el6
32      yum install -y which
33      yum install -y vim
34      yum install -y gcc-gfortran
35      yum install -y redhat-lsb-core-4.0-7.el6.centos.x86_64
36
37      # Create needed directories
38      mkdir --parents /runtimes
39      mkdir --parents /var/lib/aliprod
40      mkdir --parents /var/lib/aliprod/.alien
41      mkdir --parents /var/lib/aliprod/.alien/cache
42      mkdir --parents /var/lib/aliprod/.alien/logs
43      mkdir --parents /var/lib/aliprod/.alien/tmp
```

```
44      mkdir --parents /cvmfs
45      mkdir --parents /cvmfs/alice.cern.ch
46      mkdir --parents /cvmfs/alice-ocdb.cern.ch
47
48      # Set ownership of aliprod home directory
49      chown --recursive daniel:aliprod /var/lib/aliprod/.
            alien
50      chmod --recursive 777 /var/lib/aliprod
51      chown --recursive daniel:aliprod /runtimes
52      chmod --recursive 777 /runtimes
53
54  %runscript
55      cd
56      ./testrun.sh $*
```

Listing 13: Singularity recipe to create the testimage.

The file is divided into sections that begin with a % followed by the name of the section [49]. Entries inside *environment* are saved inside the container and are globally applied to all processes running inside a container. The *file* section copies all files and directories from the host system (left-hand side) inside the container (right-hand side). After these steps are finished, the build processes executes the commands inside the *section* from inside the container. The final section *runscript* contains the commands that will be executed when the container is normally started. An image is created with the command shown in 14 while inside the same directory as the recipe file. The resulting image was then copied 10 times. Images created with Singularity have a storage limit, unlike the ones created with Docker and rkt. The default size is around 768 megabytes but this is not enough for the files and dependencies needed for the simulation. During the creation of the container the size was therefore increased to 2 gigabytes.

```
1  sudo singularity create --size 2048 singularity_test.img
```

Listing 14: Singularity recipe to create the testimage.

When starting a container, directories from the host system can be mapped into the container and accesses from processes from within (just as Docker and rkt). This was used so that the simulations would save their times and IDs into a file on the host system. Listing 15 shows the command to start the container with the ID 2 for the testcase with 5 concurrent container at once.

The directory /home/daniel/times_singularity on the host system is mapped into the container at /runtimes, and the host directory /cvmfs/alice.cern.ch is mapped to the same position.

```
1 singularity run -w -B /home/daniel/times_singularity:/
    runtimes -B /cvmfs/alice.cern.ch singularity_1.img 5 2
```

Listing 15: Command to start Singularity image.

3.3 Test procedure

The wall clock time the aliroot simulation needed to finish was measured and taken as the reference. The number of concurrent instances ranged from 1 through 10 for both the containers (Docker, rkt, Singularity) and the uncontainered simulation. All of these test cases were executed five times. All these tests were conducted on two different kernels: one patched with AUFS and one patched with grsecurity and PaX.

This yields the following tests cases that were all executed five times:

1. Stock Kernel
    - Without Container: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside Docker: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside rkt: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside Singularity: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
2. Hardened Kernel (grsecurity and PaX)
    - Without Container: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside Docker: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside rkt: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.
    - Inside Singularity: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 jobs at once.

3.4 Test setup

The wall clock time it took the simulation to finish without any container on a stock kernel was measured first. Five runs of each testrun were taken, first

executing one job alone, then two simultaneously, then three, and so on up to and including ten at once. After the five runs of the un-containered tests the same process was repeated with Docker, rkt and Singularity.

After the first round the machine rebooted with the hardened kernel for the second round of tests. Only the testruns without a container, inside rkt, and inside Singularity were repeated. As explained above, Docker needs the AUFS kernel module whose patches had to be applied to the kernel. Both AUFS and grsecurity with PaX modified the same files and just applying the patches one after the other, as explained above, reversed some of the changes of the previous one, which resulted in compilation errors. It should be possible to apply the two patches manually but it would have taken too much time and it was not the main scope of this work.

During both of the kernels (stock and hardened) the started programs sometimes failed instantly with a message that there are too many levels of symbolic links inside the directory /cvmfs/alice.cern.ch. Clearing the cache and probing the configured directories with the provided *cvmfs_config* program solved the error. Listing 16 shows how.

```
1 cvmfs_config wipecache
2 cvmfs_config probe
```

Listing 16: Commands to fix error of too many levels of symbolic links.

Another issue during the tests of the un-containered simulation was grsecurity and PaX. The simulation got killed with the message shown in listing 17. PaX' overflow detection plugin spottet a memory overflow and killed the process aliroot. This error was not necessarily a bug in aliroot it could also be a bug in the PaX plugin. Other people had the same problem with different kinds of applications. Thoroughly inverstigating the source of the issue would have consumed too much time. The *PAX_SIZE_OVERFLOW* in the kernel configuration was therefore disabled and the kernel was recompiled.

```
1 PAX: size overflow detected in function
     invalidate_inode_pages2_range mm/truncate.c:626 cicus
     .140_280 max, count: 5, decl: unmap_mapping_range; num:
     3; context: fndecl;
```

```
2 CPU: 6 PID: 1234 Comm: aliroot Tainted: G I 4.9.23-grsec #4
3 ...
```

Listing 17: PaX error message of un-containered simulation.

The next issue came from rkt trying to start a pod. The filesystem protections of grsecurity were preventing rkt from setting up the container environment and starting it. Disabling it at compile time would remove a lot of mitigations that could be useful in a production environment. Some of them can be enabled and disabled during the runtime of the kernel by writing a 1 or a 0 into the proper subdirectory of the */proc* virtual filesystem. Only the protections that prevented rkt from starting the pod were disabled, one after the other. A total of four protections had to be disabled this way at runtime (without recompiling the kernel). The four protections were: deny calling fchdir while inside a chroot, deny calling chroot while inside a chroot, deny changing capabilities while inside a chroot, deny mounting directories while inside a chroot. All of these system calls are potentially dangerous if called from within a chroot'ed environment, as they could be used to escape the already effective chroot. The commands with the complete paths are shown in listing 18.

```
1 echo 0 > /proc/sys/kernel/grsecurity/chroot_deny_fchdir.
2 echo 0 > /proc/sys/kernel/grsecurity/chroot_deny_chroot.
3 echo 0 > /proc/sys/kernel/grsecurity/chroot_caps.
4 echo 0 > /proc/sys/kernel/grsecurity/chroot_deny_mount.
```

Listing 18: Disabling PaX filesystem protection features preventing rkt from starting a pod.

4 Results

This chapter presents the results, discusses them and shows possible reasons for the time differences between the un-conainered simulation and the different container solutions.

4.1 Results

Tables 1 and 2 show the average runtime, in seconds, of the un-containered simulation and every container for the stock kernel and hardened kernel respectively. Every testcase (one concurrent job, two concurrent jobs, etc.) was executed and measured five times. The times are for easier comparison also shown in figure 3 (stock kernel) and 4 (hardened kernel).

Figures 5, 7 and 8 show the time difference of the un-containered simulation, the simulation inside a rkt container, and the simulation inside a singularity container, between a stock and a hardened kernel respectiely. Figure 6 show the average time of Docker only on a stock kernel. As previously explained Docker could not be tested on a hardened kernel.

Concurrent Jobs	un-containered [s]	Docker [s]	rkt [s]	Singularity [s]
1	4406	4257	4316	4287
2	4387	4273	4324	4400
3	4380	4327	4395	4395
4	4288	4369	4478	4413
5	4450	4488	4607	4522
6	4511	4564	4697	4617
7	4574	4583	4730	4656
8	4846	4630	4794	4882
9	5287	5023	5226	5148
10	5699	5497	5686	5702

Table 1: Average runtimes of all simulations on a stock kernel, in seconds.

4.2 Discussion

The execution time of the un-containered simulation was measured and all container runtimes were compared against it. The speedup of containers on a stock kernel was at most 2.84%, while the biggest slowdown was almost 5%. All speedups and slowdowns are shown in table 3. A positive value reprents a slowdown of this testcase compared to the un-containered case while a negative value shows that this container was faster.

Except for four un-containered jobs at once the execution time of all four testcases increased with the number of simultaneous jobs. All four test cases

Concurrent Jobs	un-containered [s]	rkt [s]	Singularity [s]
1	4886	4333	4755
2	4872	4366	4769
3	4925	4430	4794
4	4928	4496	4850
5	4935	4630	4866
6	4935	4709	4881
7	4960	4754	4938
8	5071	5013	5129
9	5317	5342	5565
10	5781	5811	6033

Table 2: Average runtimes of all simulations on a hardened kernel, in seconds.

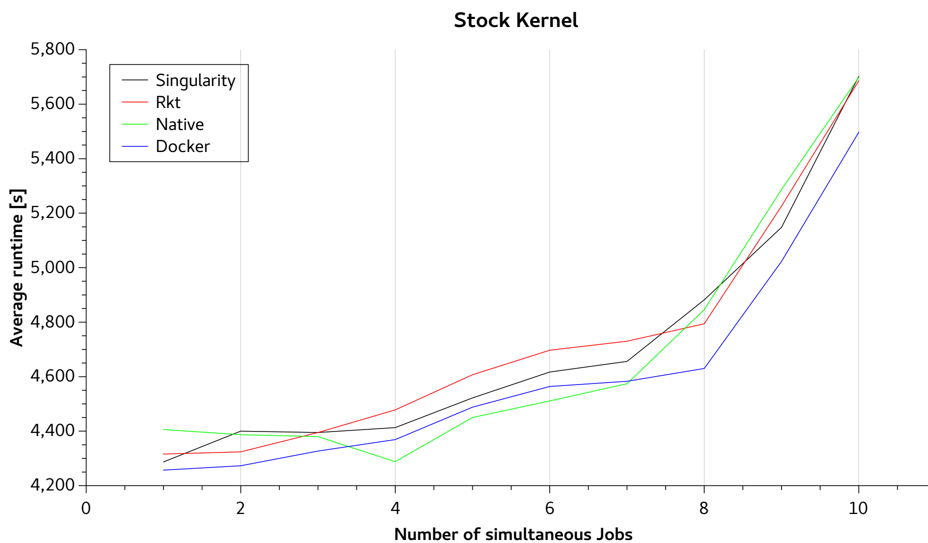

Figure 3: Comparison of runtimes without a container, Docker, rkt and Singularity on a stock kernel, in seconds.

them show a sharp increase at or after eight concurrent jobs. The memory consumption of the simulation is not the reason for this sharp increase. One un-containered job comsumed at most approximately 1020 megabytes of main memory while the containers consumed not more than 980 megabytes. The

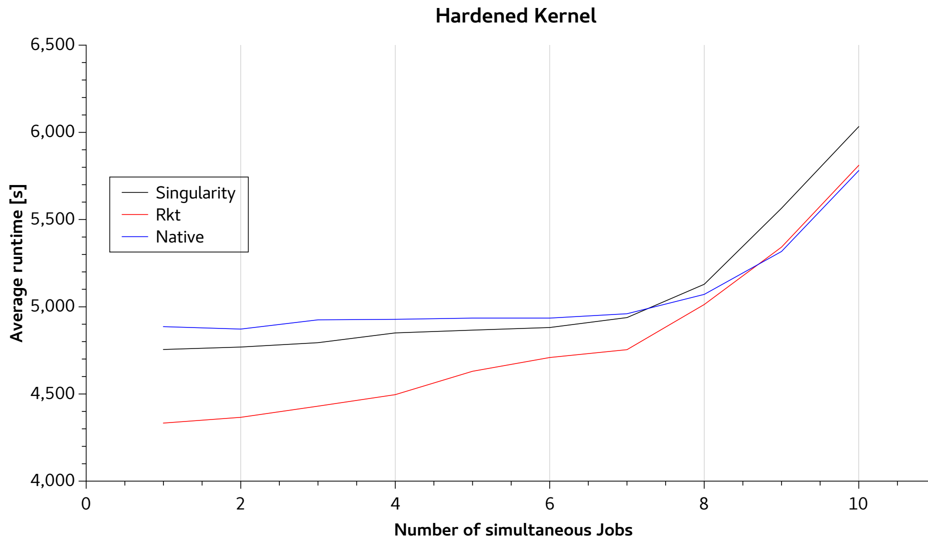

Figure 4: Comparison of runtimes without a container, rkt and Singularity on a hardened kernel, in seconds.

testmachine had 12 gigabytes of main memory and the operating system used around 250 megabytes. Even with ten jobs running at the same time the operating system did not had to swap memory pages out onto the hard disk.

Possible reasons for the sharp increase could be kernel, disk or processor caching effects, or an increased management overhead from the kernel. The kernel caches some recently accessed data, both main memory and disk data, to speed up the access time if it is requested again in the near future. It could have been that these caches got purged while there were many jobs running, which means that the needed data had to be requested from the main memory (in the case of the CPU cache) or from the disk (in the remaining cases). The possible increased management overhead of the kernel can be explained with the almost completely used main memory and the corresponding housekeeping (e.g. memory pages for virtual memory). The increased number of process switches and associated context-switches could also have flushed the processor caches which resulted in longer waiting times for all running processes.

The difference between the containers on a hardened kernel is bigger than on a stock kernel. The speedup was as big as 12.75%, while the slowdown was

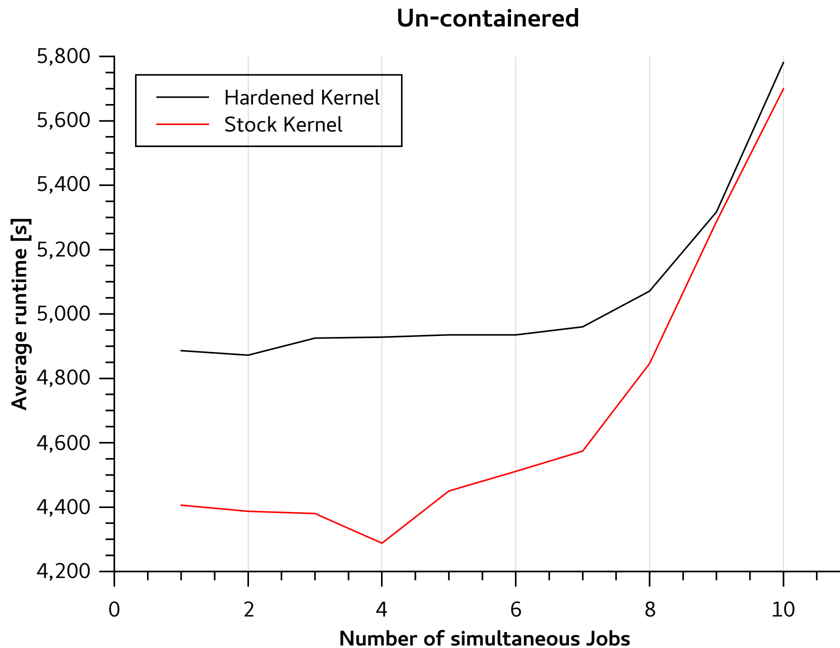

Figure 5: Average runtimes of the un-containered simulation on a stock and a hardened kernel, in seconds.

only 4.44%. The only difference between these tests and the previous ones is the kernel which means the time difference arises solely from the hardening process. The same container images, settings and restrictions were used on both kernels. The memory consumption of the running container (and un-containered simulation) was also the same as on the stock kernel. Additional checks, randomizations, etc. inside the hardened kernel increased time of system calls which thereby also increased the overall execution time of the simulation. It could also be that the size of the data structures for managing user processes are bigger on a hardened kernel than on a stock kernel.

Every testcase on both kernels had the same resource limitation, not one container had different limits (neither minimal nor maximum). The nice-value, used to influence the scheduling priority of a process, was also the same in every test. The simulation itself could be in some way affected when running inside a container. This could be analyzed with one of the recommended profiler for aliroot listed in the offline bible [51, Chapter 3.4.10]. Another difference that could have impacted the performance between the

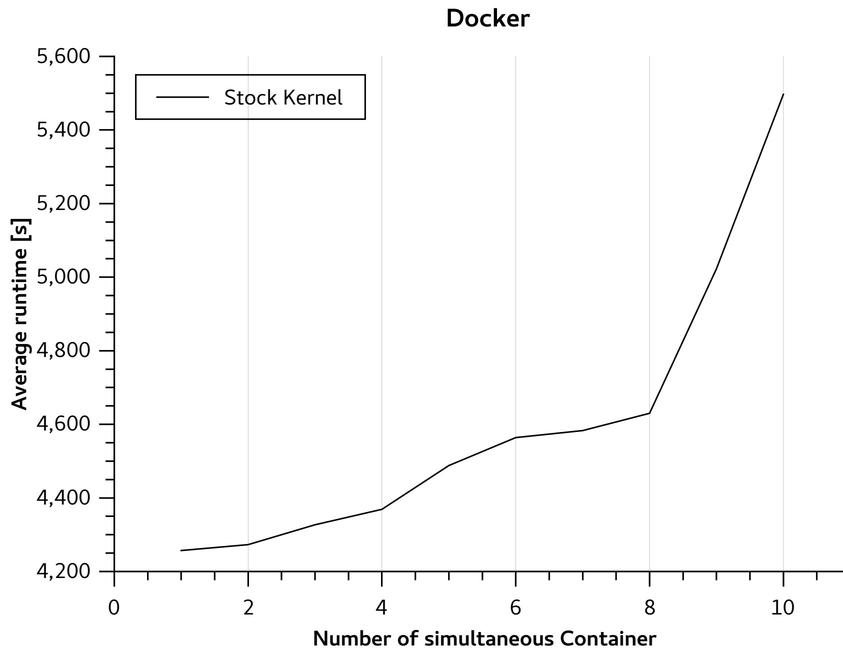

Figure 6: Average runtimes of the simulation inside Docker on a stock kernel, in seconds.

un-containered simulation and the containers were some of the used libraries. A lot of distributions modify, compile and package the available software on their own. The libraries installed from the respective repositories therefore differed from distribution to distribution.

The physics simulation used in these tests almost never accessed the hard disk. The I/O performance of Docker, rkt and Singularity was therefore not tested. These measurements should therefore not be applied to I/O-heavy applications such as a database or webserver. The performance difference of them could differ substantially from the simulation used in this work.

Specialized tests could be used to determine the performance impact of container technologies on the I/O-throughput and also other parts of the system. Benchmarking-softwares are capable to test only one subsystem, like I/O or network. These tests could be conducted on container technologies to determine if a subsystem performs significantly worse and if yes, if they could be optimized.

Another interesting container technology to test would be LXD **??** which

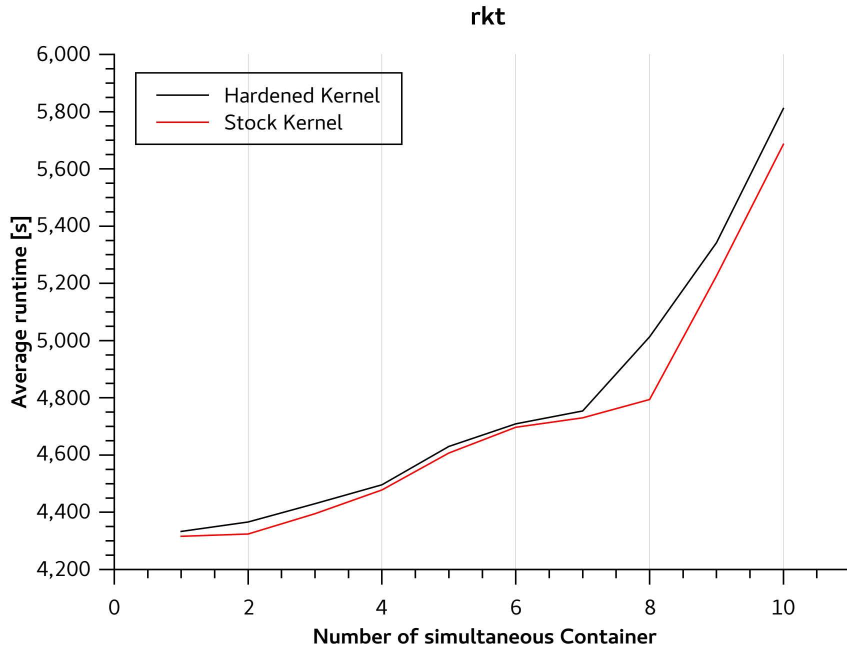

Figure 7: Average runtimes of the simulation inside rkt on a stock and a hardened kernel, in seconds.

Concurrent Jobs	Docker to un-cont.	rkt to un-cont.	Singularity to un-cont.
1	3.39%	-2.09%	-2.78%
2	2.61%	-1.46%	0.30%
3	1.21%	0.35%	0.35%
4	-1.88%	4.25%	2.84%
5	-0.86%	3.40%	1.59%
6	-1.18%	3.96%	2.29%
7	-0.22%	3.30%	1.77%
8	4.47%	-1.08%	0.74%
9	4.99%	-1.15%	-2.69%
10	3.54%	-0.22%	0.05%

Table 3: Runtime difference of the container compared to un-containered simulation on a stock kernel.

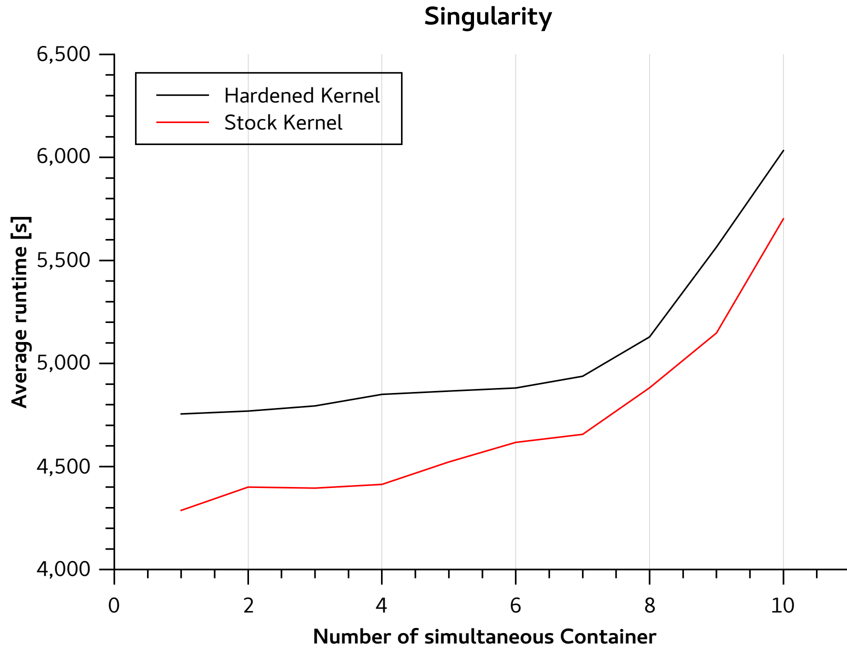

Figure 8: Average runtimes of the simulation inside Singularity on a stock and a hardened kernel, in seconds.

Concurrent Jobs	rkt to un-cont.	Singularity to un-cont.
1	-12.75%	-2.76%
2	-11.58%	-2.16%
3	-11.17%	-2.73%
4	-9.62%	-1.60%
5	-6.59%	-1.43%
6	-4.81%	-1.11%
7	-4.35%	-0.44%
8	-1.15%	1.13%
9	0.45%	4.44%
10	0.51%	4.18%

Table 4: Runtime difference of the container compared to un-containered simulation on a hardened kernel.

is developed by Canonical Ltd. Unlike the container solution tested in this work, LXD container run on top of a hypervisor. As explained in chapter 2.1, hypervisors are a software layer that completely manages a running virtual machine. The authors claim that there is no performance penalty for the software running inside a LXD container compared to an un-containered execution.

Manually patching a kernel with grsecurity and PaX is not always possible or desired. Security features provided by the stock Linux kernel could still be used to secure running applications on a system. AppArmor, SELinux and others allow a user to restrict access to resources of running processes. The performance impact of these features could be investigated.

These security features are most effective if the profile is customized for the application it is applied to. This can be a tedious and time-consuming task that could be automated. AppArmor has two modes: complain and enforcement. In the complain-mode AppArmor logs all profile violations but does not terminate a process. This approach could be applied to SELinux to automatically create custom profiles for an application.

5 Conclusion and Future Work

5.1 Conclusion

This work analyzed the impact different container technologies have on the performance of jobs on a high-energy physics grid. The physics simulation PbPbbench, from the ALICE experiment of CERN, was used as a benchmark to compare the differences. Physics simulations inside of CERN run inside a high-energy physics/high-throughput computing environment. Providers of these environments strive to provide as much computing resources as possible with their available hardware, while at the same time securing their own and their customers data. This protection can be achieved with separation of jobs as provided by virtual machines or containers. Unlike containers, virtual machines incur a rather big resource overhead that make them unattractive in these environments.

The time the simulation needs to finish outside of a container was measured and taken as a baseline. Three container solutions, namely Docker, rkt and Singularity, were tested in this work. The same simulation was packed into these containers, measured and compared with the un-containered sim-

ulation. The difference between the un-containered simulation and the containers was at most 5% on a stock Linux kernel, and almost 13% on a Linux kernel that was hardened with grsecurity and PaX. Packaging the simulation into containers even reduced the execution time in some cases.

Even though containered processes are stronger separated by the kernel from each than when not running inside a container, the performance did not nessecarily degrade. Not only the runtime but also the resource usage is affected by the container technology used. All three of the tested containers used slightly less main memory than the un-containered simulation.

5.2 Future Work

This work showed how the performance of an average physics simulation is affected by packaging it into different containers technologies. Physics simulation do not extensively use all subsystems (e.g. hard disk access). These subsystem could be tested with microbenchmarks to determined if any of them is significantly slower.

Some container technologies allow an easy confinement of container with AppArmor, SELinux, and others. It is also possible to restrict a process or group of processes in its capabilities without using any container. It could be investigated how these confinements impact the performance of a process or container.

Thinking this idea further a program could be designed that analyzes a program and creates custom SELinux profiles.

Another container technology that could be tested is LXD from Canonical. These container run on top of a hypervisor but allegedly without any performance loss compared to an un-containered execution.

References

[1] Ubuntu manpage: apparmor.d - syntax of security profiles for apparmor. http://manpages.ubuntu.com/manpages/xenial/man5/apparmor.d.5.html. [Online, accessed November 25th 2017].

[2] Kubernetes Authors. Pods - kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod/. [Online, accessed November 5th 2017].

[3] Jim Basney and Miron Livny. Deploying a high throughput computing cluster.

[4] Eric W. Biederman. ip-netns(8). http://man7.org/linux/man-pages/man8/ip-netns.8.html. [Online, accessed October 21th 2017].

[5] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. *National Computer Conference*, 1973.

[6] Massimo Cafaro and Giovanni Aloisio. Grids, clouds and virtualization, 2011.

[7] CERN. Cernvm file system. https://cernvm.cern.ch/portal/filesystem. [Online, accessed November 4th 2017].

[8] CERN. Cernvm-fs client quick start. https://cernvm.cern.ch/portal/filesystem/quickstart. [Online, accessed November 4th 2017].

[9] CERN. Physics — cern. https://home.cern/about/physics. [Online, accessed December 3rd 2017].

[10] CERN. Root a data analysis framework. https://root.cern.ch/. [Online, accessed November 5th 2017].

[11] CERN. Welcome to the home page of the alice off-line project — alice offline. https://alice-offline.web.cern.ch/. [Online, accessed November 4th 2017].

[12] corbet. Process containers [lwn.net]. https://lwn.net/Articles/236038/. [Online, accessed October 22th 2017].

[13] CoreOS. rkt, a security-minded, standards-based container engine. https://coreos.com/rkt/. [Online, accessed November 12th 2017].

[14] CoreOS. spec/ace.md at master appc/spec github. https://github.com/appc/spec/blob/master/spec/ace.md#oslinuxcapabilities-remove-set. [Online, accessed November 12th 2017].

[15] Wes Felter, Alexandre Ferreira, Ramakrishnan Rajamony, and Juan C. Rubio. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.

[16] grsecurity Team. Grsecurity/appendix/grsecurity and pax configuration options - wikibooks, open books for an open world. https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Opt [Online, accessed November 1st 2017].

[17] Jaques Glinas. vserver 0.4 change log. http://archives.linux-vserver.org/200110/0001.html. [Online, accessed October 18th 2017].

[18] Joshua Higgins, Violeta Holmes, and Colin C. Venters. Securing user defined containers for scientific computing. *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 449–453, 2016.

[19] Murray Hill. Unix time-sharing system: Unix programmer's manual. https://s3.amazonaws.com/plan9-bell-labs/7thEdMan/v7vol1.pdf. [Seventh Edition, Online, accessed October 21th 2017].

[20] Jordan K. Hubbard. Freebsd 4.0 announcement. https://www.freebsd.org/releases/4.0R/announce.html. [Online, accessed October 17th 2017].

[21] Docker Inc. Docker - build, ship, and run any app, anywhere. https://www.docker.com/. [Online, accessed October 17th 2017].

[22] Docker Inc. Docker run reference — docker documentation. https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities. [Online, accessed December 7th 2017].

[23] Docker Inc. Docker security. https://docs.docker.com/engine/security/security/#other-kernel-security-features. [Online, accessed November 5th 2017].

[24] Mikolaj Krzewicki. Ocdb master mikolaj krzewicki / aliroot. https://gitlab.cern.ch/mkrzewic/AliRoot/tree/master/OCDB. [Online, accessed December 5rd 2017].

[25] Gregory M Kurtzer, Vanessa V. Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. In *PloS one*, 2017.

[26] Miron Livny, Jim Basney, and Ramakrishnan Raman. Mechanisms for high throughput computing. 1997.

[27] Linux man-pages project. capabilities(7). http://man7.org/linux/man-pages/man7/capabilities.7.html. [Online, accessed November 15th 2017].

[28] Linux man-pages project. cgroup_namespaces(7). http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html. [Online, accessed October 19th 2017].

[29] Linux man-pages project. mount_namespaces(7). http://man7.org/linux/man-pages/man7/mount_namespaces.7.html. [Online, accessed October 21th 2017].

[30] Linux man-pages project. namespaces(7). http://man7.org/linux/man-pages/man7/namespaces.7.html. [Online, accessed October 19th 2017].

[31] Linux man-pages project. pid_namespaces(7). http://man7.org/linux/man-pages/man7/cgroups.7.html. [Online, accessed October 22th 2017].

[32] Linux man-pages project. pid_namespaces(7). http://man7.org/linux/man-pages/man7/pid_namespaces.7.html. [Online, accessed October 22th 2017].

[33] Linux man-pages project. user_namespaces(7). http://man7.org/linux/man-pages/man7/user_namespaces.7.html. [Online, accessed October 19th 2017].

[34] Watson McKusick, Neville-Neil. *The Design and Implementation of the FreeBSD Opearting System.* Addison-Wesley Professional, 2 edition, 2014.

[35] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Syst. J.*, 9:199–218, 1970.

[36] Todd C. Miller. Sudo manual. https://www.sudo.ws/man/1.8.18/sudo.man.html. [Online, accessed November 25th 2017].

[37] Inc Open Source Security. grsecurity. https://grsecurity.net/. [Online, accessed November 1st 2017].

[38] Linux Kernel Organization. Index of /pub/linux/kernel/v2.4/. https://www.kernel.org/pub/linux/kernel/v2.4/. [Online, accessed October 21th 2017].

[39] Linux Kernel Organization. Index of /pub/linux/kernel/v2.6/. https://www.kernel.org/pub/linux/kernel/v2.6/. [Online, accessed October 21th 2017].

[40] Linux Kernel Organization. The linux kernel archives. https://www.kernel.org/. [Online, accessed December 3rd 2017].

[41] Matthew Portnoy. Virtualization essentials, 2012.

[42] Linux VServer Project. Paper. http://linux-vserver.org/Paper. [Online, accessed October 18th 2017].

[43] The FreeBSD Project. Freebsd 4.0 release notes. https://www.freebsd.org/releases/4.0R/notes.html. [Online, accessed October 17th 2017].

[44] The FreeBSD Project. Freebsd 4.0 release notes. https://www.freebsd.org/doc/handbook/jails.html. [Online, accessed October 17th 2017].

[45] The FreeBSD Project. Jail man page. https://www.freebsd.org/cgi/man.cgi?query=jail&sektion=8. [Online, accessed October 17th 2017].

[46] The OpenBSD Project. chroot(2). https://man.openbsd.org/chroot.2. [Online, accessed October 21th 2017].

[47] Rohit Seth. Containers: Introduction [lwn.net]. https://lwn.net/Articles/199643/. [Online, accessed October 22th 2017].

[48] Singularity. Singularity. http://singularity.lbl.gov/. [Online, accessed November 5th 2017].

[49] Singularity. Singularity. http://singularity.lbl.gov/docs-recipes. [Online, accessed December 7th 2017].

[50] AppArmor Team. Ubuntu manpage: Apparmor - kernel enhancement to confine programs to a limited set of resources. http://manpages.ubuntu.com/manpages/xenial/man7/apparmor.7.html. [Online, accessed November 25th 2017].

[51] The ALICE Team. The alice offline bible, version 0.00 (rev. 23). http://svn.cern.ch/guest/AliRoot/trunk/doc/OfflineBible.doc. [Online, accessed December 5rd 2017].

[52] The AUFS Team. http://aufs.sourceforge.net/. [Online, accessed December 3rd 2017].

[53] Miguel G. Xavier, Marcelo Veiga Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and César A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, 2013.