Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

Virtualization technologies from hypervisors to containers: overview, security considerations, and performance comparisons

Tesi in
Sicurezza delle Reti

Relatore:
GABRIELE D'ANGELO

Presentata da:
ALESSANDRO FANTINI

Sessione III
Anno Accademico 2015/2016

To my parents, Davide and Letizia,
for always supporting me.

Abstract (Italian)

In un'epoca in cui quasi tutte le persone utilizzano quotidianamente applicazioni basate su cloud senza nemmeno farci caso e le organizzazioni del settore IT stanno investendo notevoli risorse in questo campo, non tutti sanno che il cloud computing non sarebbe stato possibile senza la virtualizzazione, una tecnica software che ha le sue radici nei primi anni sessanta.

Lo scopo di questa tesi è fornire una panoramica delle tecnologie di virtualizzazione, dalla virtualizzazione hardware e gli hypervisor fino alla virtualizzazione a livello di sistema operativo basata su container, analizzare le loro architetture e fare considerazioni relative alla sicurezza. Inoltre, dal momento che le tecnologie basate su container si fondano su funzioni specifiche di contenimento del kernel Linux, alcune sezioni sono utilizzate per introdurre ed analizzare quest'ultime singolarmente, al livello di dettaglio appropriato.

L'ultima parte di questo lavoro è dedicata al confronto quantitativo delle prestazioni delle tecnologie basate su container. In particolare, LXC e Docker sono raffrontati su una base di cinque test di vita reale e le loro prestazioni sono confrontate fianco a fianco, per evidenziare le differenze nella quantità di overhead che introducono.

Abstract

In an era when almost everyone is using cloud-based applications on a daily basis without ever thinking about it and IT organizations are investing substantial resources in this field, not everybody knows that cloud computing could not have been possible without virtualization, a software technique that has its roots in the early 1960s.

The purpose of this thesis is to give an overview of virtualization technologies, from hardware virtualization and hypervisors to container-based operating-system-level virtualization, analyze their architectures, and make security-related considerations. Moreover, since container technologies are underpinned by specific containment features of the Linux kernel, a few sections are used to introduce and analyze the latter individually, at the appropriate level of detail.

The last part of this work is devoted to a quantitative comparison of the performance of container-based technologies. In particular, LXC and Docker are benchmarked against five real-life tests and their performance are compared side-by-side, to highlight differences in the amount of overhead they introduce.

Contents

Introduction

Virtualization is a software technique that has its roots in the early 1960s. Over time, the concept evolved and multiple branches, like hardware virtualization and operating-system-level virtualization, were created.

Because different types of virtualization exist, it is difficult to find a definition that fits them all. According to Singh [26], virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.

Pierce et al. (2013) [19] define system virtualization as the use of an encapsulating software layer that surrounds or underlies an operating system and provides the same inputs, outputs, and behavior that would be expected from physical hardware. The software that performs this is called a *hypervisor*, or *Virtual Machine Monitor* (VMM). This abstraction means that an ideal VMM provides an environment to the software that appears equivalent to the host system, but is decoupled from the hardware state. For system virtualization, these virtual environments are called *Virtual Machines* (VMs), within which operating systems may be installed. Since a VM is not dependent on the state of the physical hardware, multiple VMs may be installed on a single set of hardware.

Other definitions identify virtualization as a technique that allows for the creation of, one or more, virtual machines that exist inside one computer or the software reproduction of an entire system architecture,

which provides the illusion of a real machine to all software running above it.

Virtualization is performed on a given hardware platform by *host* software, which creates a simulated computer environment (i.e., a virtual machine), for its *guest* software. The guest software can be user applications or even complete operating systems. Virtualization refers to the abstraction of computer resources. It separates user and applications from the specific hardware characteristics they use to perform their task and thus creates virtual environments [7]. The key point is that guest software should execute as if it were running directly on the physical hardware, with some limitations. As an example, access to physical system resources is generally managed at a more restrictive level than the host processor and system memory while guests are often restricted from accessing specific peripheral devices, or may be limited to a subset of the device's native capabilities, depending on the hardware access policy implemented by the virtualization host. The purpose of creating virtual environments is to improve resource utilization by aggregating heterogeneous and autonomous resources. This can be provided by adding a layer, called hypervisor, between the operating system and the underlying hardware [7].

As will be further explained in the first chapter, the hypervisor becomes the fundamental building block to provide hardware virtualization and its primary task is to monitor the virtual machines that are running on top of it.

Chapter 1

Hardware virtualization

Researches in the field of hardware virtualization dated back to the 1960s [6, 10] but only in the early 1970s [20] it started to become popular among computer scientists, that formalized some fundamental concepts that are still actively being used to teach the basis of virtualization.

1.1 Virtualization requirements

In the early years of virtualization, computer scientists Gerald J. Popek and Robert P. Goldberg defined a set of conditions sufficient for a computer architecture to support system virtualization efficiently. In their 1974 article *Formal Requirements for Virtualizable Third Generation Architectures* [20], they expressed formal requirements for a computer architecture to support efficient virtualization and provided major guidelines for the design of virtualized computer architectures.

Even if the original analysis by Popek and Goldberg was for old computer systems and the requirements were derived under simplifying assumptions, its content still holds true for current generation computers.

According to Popek and Goldberg, a virtual machine is an efficient, isolated duplicate of the real machine. The virtual machine abstraction is provided by a piece software called Virtual Machine Monitor (VMM), that has the following three essential characteristics [20]:

- **Equivalence / Fidelity**: The VMM provides an environment for programs which is essentially identical with the original machine, i.e. any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with two possible exceptions:
  - Differences caused by the availability of system resources. This consideration arises from the ability to have varying amounts of memory made available by the virtual machine monitor. The identical environment requirement excludes the behavior of the usual time-sharing operating system from being classed as a virtual machine monitor.
  - Differences caused by timing dependencies because of the intervening level of software and because of the effect of any other virtual machines concurrently existing on the same hardware.
- **Efficiency / Performance**: Programs that run in this environment show at worst only minor decreases in speed. To achieve this performance requirement, a statistically dominant subset of the virtual processor's instructions must be executed directly by the real processor, without VMM intervention. A virtual machine, in fact, is different from an emulator. An emulator intercepts and analyzes every instruction performed by the virtual processor, whereas a virtual machine executes most of the instructions on the real processor, relying on the virtual one only in some cases.
- **Resource control / Safety**: The VMM is in complete control of system resources (memory, peripherals, etc.) although not necessarily processor activity. The VMM is said to have complete control of system resources if both of the following two conditions hold true:
  - It is not possible for a program running under the VMM in the created environment to access any resource not explicitly allocated to it.

– It is possible under certain circumstances for the VMM to regain control of resources already allocated.

Popek and Goldberg [20] affirm that any control program that satisfies the three properties of efficiency, resource control, and equivalence is essentially a Virtual Machine Monitor (VMM). The environment which any program sees when running with a virtual machine monitor present is called a Virtual Machine (VM), instead. From another perspective, a virtual machine is the environment created by the virtual machine monitor.

An implication of the given definition is that a VMM is not necessarily a time-sharing system, although it may be. However, the identical-effect requirement applies regardless of any other activity on the real computer, so that isolation, in the sense of protection of the virtual machine environment, is meant to be implied. This requirement also distinguishes the virtual machine concept from virtual memory. Virtual memory is just one possible ingredient in a virtual machine; and techniques such as segmentation and paging are often used to provide virtual memory. The virtual machine effectively has a virtual processor, too, and possibly other virtual devices.

Smith and Nair, in their 2015 book [27] state that VMMs need only to satisfy the equivalence and resource control properties. A VMM that also satisfies the efficiency property is said to be efficient.

According to Popek and Goldberg [20], VMM developers main problem is to conceive a VMM that would satisfy all the three previous conditions while coping with the limitations of the Instruction Set Architecture (ISA)[1] on the host hardware platform.

[1]The ISA defines the machine code that a processor reads and acts upon as well as the word size, memory address modes, processor registers, and data type. It can be seen as an interface between a computer's software and its underlying processor.

1.2 Virtualization theorems

In their paper [20], Popek and Goldberg refer to a conventional processor with two modes of operation, *supervisor* and *user*, often called *privileged* and a *non-privileged* modes. This is quite common in today's architectures as well. The main difference between the two modes of operation is in the set of available instructions: in privileged mode all instructions are available to software, whereas in non-privileged mode they are not. The Operating System (OS) provides a small resident program called the *privileged software nucleus* (i.e., the *kernel*). User programs could execute the non-privileged hardware instructions or make *supervisory calls* (analogous to *system calls*) to the privileged software nucleus to make it perform privileged functions on their behalf. This is proven to work well for many purposes, however there are fundamental limits with the approach [26]. The following are the most remarkable:

- As only one *bare machine interface* is exposed, only one kernel can be run. Anything, whether it be another kernel – belonging to the same or a different operating system – or an arbitrary program that requires to talk to the bare machine – such as a low-level testing, debugging, or diagnostic program – cannot be run alongside the booted kernel.
- Activities that would disrupt the running system (for example, upgrade, migration, system debugging, etc.) cannot performed. Untrusted applications cannot run in a secure manner.
- One cannot easily provide the illusion of a hardware configuration that one does not have (multiple processors, arbitrary memory and storage configurations, etc.) to some software.

The Instruction Set Architecture (ISA) of a processor is made of three groups of instructions:

- **Privileged**: Instructions that trap[2] only if the processor is in user mode and do not trap if it is in supervisor mode. The trap that

[2]When a trap occurs, the current state of the machine is automatically saved

occurs under these conditions is often called a privileged instruction trap. This notion of a privileged instruction is close to the conventional one. Privileged instructions are independent of the virtualization process.

- **Control Sensitive**: Instructions that attempt to change the amount of (memory) resources available, or affect the processor mode without going through the memory trap sequence.

- **Behavior Sensitive**: An instruction is behavior sensitive if the effect of its execution depends on the value of the relocation-bounds register (i.e., upon its location in real memory), or on the mode. The other two cases, where the location-bounds register or the modes do not match after the instruction is executed, fall into the class of control sensitive instructions.

Assuming the programs one wants to run on the various virtual machines on a system are all native to the architecture (i.e., they do not need emulation of the instruction set) the virtual machines can be run in non-privileged mode. One would imagine that non-privileged instructions can be directly executed without involving the VMM, and since the privileged instructions would cause a trap (since they are being executed in non-privileged mode), they can be caught by the VMM, and appropriate action can be taken (for instance, they can be simulated by the VMM in software). Problems arise from the fact that there may be instructions that are non-privileged, but their behavior depends on the processor mode. These instructions are *sensitive*, but they do not cause traps [26].

The three virtualization theorems that use the previous instructions' classification are [20]:

- **Theorem 1 – Sufficient condition to guarantee virtualiz-**

and control is passed to a pre-specified routine by changing the processor mode, the relocation bounds register, and the program counter.

ability: *For any conventional third generation computer*[3]*, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

- **Theorem 2 – Requirements for recursive virtualizability**: *A conventional third generation computer is recursively virtualizable if (a) it is virtualizable, and (b) a VMM without any timing dependencies can be constructed for it.*

  Thus, if it is possible for a virtual machine system to run under itself a copy of the VMM that also exhibits all the properties of a VMM and this procedure can be repeated until the resources of the system are consumed, then the original machine is recursively virtualizable.

- **Theorem 3 – Requirements for Hybrid Virtual Machine**[4] **(HVM) monitors**: *A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.*

  The difference between a HVM monitor and a VMM is that, in the HVM monitor, all instructions in virtual supervisor mode are interpreted. Otherwise the HVM monitor is the same as the VMM. Equivalence and control can then be guaranteed as a result of two facts. Firstly, as in the VMM, the HVM monitor always either has control, or gains control via a trap, whenever there is

[3] Third generation computers (1965–1971) were the first to adopt integrated circuits in place of transistors. The invention of the integrated circuit by Jack Kilby in 1958 and the following mass adoption in electronics made computers smaller in size, more reliable, and efficient. Third generation computers were essentially scaled-down versions of mainframe computers. The *IBM 360*, *Honeywell 6000*, and *DEC PDP-10* are prominent examples of third generation computers.

[4] A Hybrid Virtual Machine (HVM) system has a structure that is almost identical to a virtual machine system, but more instructions are interpreted rather than being directly executed.

an attempt to execute a behavior sensitive or control sensitive instruction. Secondly, by the same argument as before, there exist interpretive routines for all the necessary instructions. Hence, all sensitive instructions are caught by the HVM and simulated. For these reason, HVMs are less efficient than VMMs.

1.3 Emulation and simulation

Virtualization software may make use of emulation or simulation for many reasons, for instance when the guest and host architectures are different.

A software emulator basically reproduces the behavior of one system on another. It is supposed to execute the same programs as the original system, producing the same results for the same input, without further intervention from the user. Software emulators are widely adopted for old and new hardware architectures, other than video game consoles and arcade games[5].

In the context of computing, a simulation is an imitation of some real system. A simulator can be informally thought of as an accurate emulator [26].

Today the Turing Machine has become the accepted formalization of an effective procedure [11]. Alonzo Church hypothesized that the Turing Machine model is equivalent to our intuitive notion of a computer. Alan Turing proved that a Universal Turing Machine can compute any function that any Turing Machine can compute. Often, the Church-Turing thesis is used to imply that the Universal Turing Machine can simulate the behavior of any machine. Nevertheless, given that an arbitrary computer is equivalent to some Turing Machine, it follows that all computers can simulate each other [26].

[5]The most popular arcade emulator is MAME, first released in 1997 and still being developed.

1.4 Virtualization benefits

Two primary benefits offered by any virtualization technology are **partitioning** and **isolation**.

Partitioning, also known as *resource sharing*, involves running multiple operating systems on one physical machine, by sharing the system resources between virtual machines. Unlike in non-virtualized environments, where all the resources are dedicated to the running programs, in virtualized environments the VMs share the physical resources such as memory, disk and network devices of the underlying host [23]. However, virtualization does not always implies partitioning (i.e., breaking something down into multiple entities). As an example of its different – intuitively opposite – connotation, one can take N disks, and make them appear as one logical disk through a virtualization layer [26].

One of the key issues in virtualization is to provide isolation between virtual machines that are running on the same physical hardware. Programs running in one virtual machine should not see programs running in another virtual machine [23]. Moreover, it is crucial to support isolation at the hardware level for fault tolerance and security purposes. The hard task is to preserve performance while assuring isolation. This is usually achieved with advanced resource controls.

1.5 Hypervisors classification

Hypervisors are commonly classified as one of these two types:

- **Type-1 hypervisor (*native* or *bare metal*)**: A native hypervisor (Figure 1.1) is a software system that runs directly on the top of the underlying host's hardware to control the hardware, and to monitor the guest operating systems. In this case the VMM is a small code whose responsibility is to schedule and allocate system resources to VMs as there is no operating system running below it. Furthermore, the VMM provides device drivers that guest OS use to directly access the underlying hardware [7]. Consequently,

the guest operating systems run on a separate level above the hypervisor [17]. Examples of this classic implementation of virtual machine architecture are *Xen* (*Oracle VM Server for x86*, *Citrix XenClient*), *VMware ESXi*, *Oracle VM Server for SPARC*, *Microsoft Hyper-V*, and *KVM*.

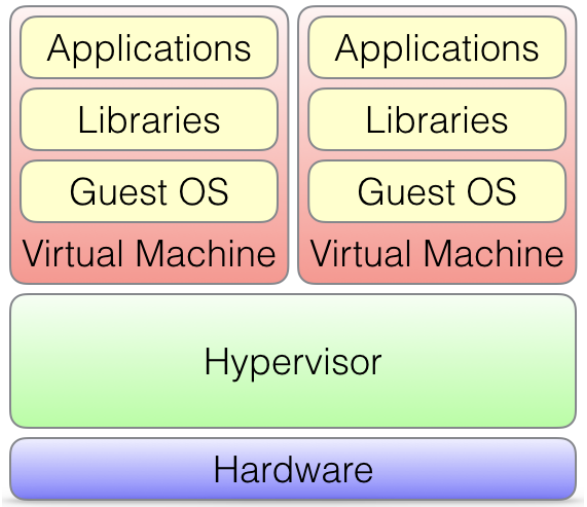


Figure 1.1: Type-1 hypervisor.

- **Type-2 hypervisor (*hosted*)**: A hosted hypervisor (Figure 1.2) is designed to run within a traditional operating system (the host). It runs as an application and the host OS does not have any knowledge about the hosted hypervisor and treats it as any other process. The host OS is responsible for resource allocation and scheduling of a guest OS. It typically performs I/O on behalf of guest OS. The guest OS issues the I/O request that is trap by host OS that in turn send to device driver that perform I/O. The completed I/O request is again route back to guest OS via host OS [7]. In other words, a hosted hypervisor adds a distinct software layer on top of the host operating system, and the guest operating system becomes a third software level above the hardware [17]. Examples of hosted hypervisors are *VMware Server/Player/Workstation/Fusion*, *QEMU*, *Parallels Desktop for Mac*, and *Oracle VM VirtualBox*.

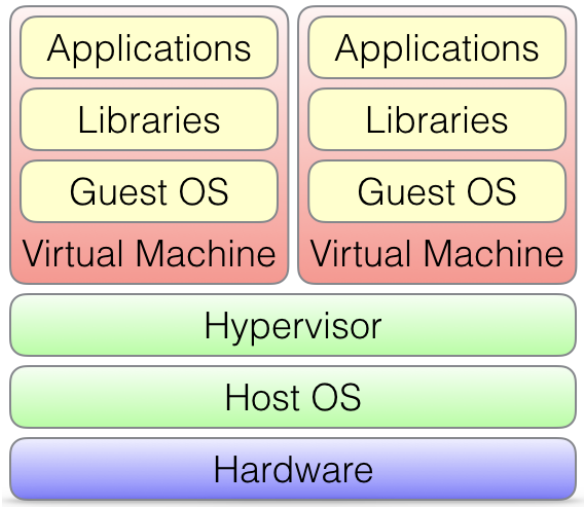


Figure 1.2: Type-2 hypervisor.

Native virtualization systems are typically seen in servers, with the goal of improving the execution efficiency of the hardware. They are arguably also more secure, as they have fewer additional layers than the alternative hosted approach. Hosted virtualization systems are more common in clients, where they run along side other applications on the host OS, and are used to support applications for alternate operating system versions or types. As this approach adds additional layers with the host OS under, and other host applications beside, the hypervisor, this may result in increased security concerns. [28]

1.6 Types of hardware virtualization

Hardware virtualization, often called platform virtualization, is the virtualization of complete hardware platforms. It allows abstraction and isolation of lower level functions and underlying hardware while enabling portability of higher level functions and sharing and/or aggregation of physical resources [23]. The piece of software that controls hardware virtualization is the hypervisor. Different hardware virtualization approaches exist and will be presented in the following sections.

1.6.1 Full virtualization

The full virtualization technique is used to create virtual machine environments that are a complete simulation of the underlying hardware so operating systems, and their respective kernels, can run unmodified inside VMs in isolation. In such environments, any software that can run on the raw hardware must be able to run in the virtual machine, including operating systems. An operating system that runs inside a virtual machine is called guest.

In this approach the VMM, often referred to as virtual machine manager, runs as an user space application on top of a host operating system and exposes a virtual hardware interface to a VM that is conceptually indistinguishable from physical hardware.

Hardware features that VMs need to access include the full instruction set, I/O operations, interrupts, memory access, and every other aspect of the hardware that is exposed to the VMs. The I/O devices are exposed to the guest machines by imitating the physical devices in the virtual machine monitor. Interactions with these devices in the virtual environment are then directed to the real physical devices either by the host operating system driver or by the VM driver. Moreover, the virtual hardware configurations provided by the VMM, on top of which guest operating systems and their respective applications can run, are usually highly customizable. The point is that the virtual machine environments provide enough representation of the underlying hardware to allow guest operating systems to run without modification [23].

Full virtualization implementations usually need a combination of hardware and software support. However not all architectures have the hardware needed to support virtualization. For example, it was not possible with most of IBM's *System/360* series with the exception being the IBM *System/360-67*; nor was it possible with IBM's early *System/370* system until IBM added virtual memory hardware to the *System/370* series in 1972. In the x86 architecture, full virtualization was not possible until 2005, when Intel added the necessary hardware virtualization extensions (*VT-x*) to its processors. AMD introduced the same feature

(*AMD-V*) a year later, in 2006. There are examples of hypervisors for the x86 platform that came very close to reaching full virtualization even prior to the AMD *AMD-V* and Intel *VT-x* additions. For example, VMware uses a technique called *binary translation* to automatically modify x86 software on the fly to replace some instructions and thus provide the appearance of full virtualization. In fact, a key challenge for full virtualization is the interception and simulation of privileged operations, such as I/O instructions. Conceptually, the VMs run in isolation[6] and thus some machine instructions can be executed directly by the underlying hardware, since their effects are entirely contained within the elements managed by the control program, such as memory locations and arithmetic registers. However, there are instructions that either access or affect state information that is outside the virtual machine and thus cannot be allowed to execute directly because they would break the VM boundaries. This kind of instruction must then be trapped and simulated instead.

A simple test to verify the correctness of a full virtualization implementation is whether a general purpose operating system can successfully run inside a virtual machine without modifications.

The main advantage of this approach is the ease of use. In fact, installing a software product like *Oracle VM VirtualBox* or *VMware Workstation* is as easy as with any other software product on the OS of choice. Inside *Oracle VM VirtualBox*'s (or *VMware Workstation*'s) virtual machines, a guest OS – together with all the applications that run on top of it – can be installed and used just like it would be running directly on real hardware, without modifications at all.

Successful use cases for full virtualization include the sharing of a computer hardware among multiple users, that also benefit from the isolation from each other, and the emulation of new hardware to achieve improved reliability, security and productivity.

Full virtualization has a cost in term of performance with respect

[6]The effects of every operation performed within a given virtual machine must be kept within that virtual machine. Virtual operations cannot be allowed to alter the state of any other virtual machine, the control program, or the hardware.

to other virtualization solutions. Guest OS that are running inside virtual machines, using virtual hardware, are obviously slower than when running directly on real hardware [23]. The degree of performance loss depends both on the optimization of the VMM and on the support to virtualization that is provided by the kernel of the host operating system.

Binary translation

The emulation of one instruction set by another through translation of binary code is called binary translation. That is, sequences of instructions are translated from the source to the target instruction set. Even if the concept is intuitive, its implementation is far from naive.

With this technique, guest OS' unprivileged instructions are run directly on the host hardware while privileged ones, that cause traps, are handled by the hypervisor and emulated so that the guest is able to run unmodified because it is unaware of being virtualized.

Two main types of binary translation exist: static and dynamic. Static binary translation converts all of the code of an executable file into code that runs on the target architecture without having to run the code first. This is very difficult to do correctly, since not all the code can be discovered by the translator. For example, some parts of the executable may be reachable only through indirect branches, whose value is known only at run time. Dynamic binary translation looks at a short sequence of code then translates it and caches the resulting sequence. This works by scanning the guests memory for instructions that would cause traps. When these instructions are found in the guest OS they are dynamically rewritten in the guest memory. This happens at run time and the privileged instructions are only translated when they are about to execute. To improve performance, branch instructions are made to point to already translated and saved code whenever possible. While being complex to implement, it allows the guests to yield higher performance as opposed to the performance yielded when being completely emulated. As well as letting guests OS run unmodified on the

hypervisor.

In some cases such as instruction set simulation, the target instruction set may be the same as the source instruction set, providing testing and debugging features such as instruction trace, conditional breakpoints and hot spot detection.

A VMM built around a suitable binary translator can virtualize the x86 architecture and it is a VMM according to Popek and Goldberg [1].

1.6.2 Hardware-assisted virtualization

Hardware-assisted virtualization is the type of virtualization that was used on the virtualization systems starting from the 1960s and is essentially the same as full virtualization detailed when trap-and-emulate virtualization technique is supported directly by the hardware. However, many architecture, such as the original design of the x86 architecture, lacked proper hardware support. With introduction of the hardware virtualization extensions by Intel in 2005, efficient full virtualization on the x86 platform is possible in the classic trap-and-emulate approach, without the need to use other techniques such as binary translation. Intel's implementation is known as *VT-x* and the similar technology from AMD is called *AMD-V*[7]. Basically, those hardware extensions introduced a new operating mode, host and guest.

Intel *VT-x*

Virtualization extensions of Intel's CPUs introduce a number of new primitives to support a classical VMM for the x86 architecture. An in-memory data structure called *Virtual Machine Control Block* (VMCB) combines control state with a subset of the state of a guest virtual CPU. A new, less privileged execution mode, called *guest mode*, supports direct execution of guest code, including privileged code. The previously adopted x86 execution environment is then called *host mode*. A new instruction, `vmrun`, is used to change from *host mode* to *guest mode* [1].

[7]The original name was Secure Virtual Machine (SVM).

Upon execution of `vmrun`, the hardware loads guest state from the VMCB and continues execution in guest mode. Guest execution proceeds until some condition, expressed by the VMM using control bits of the VMCB, is reached. At this point, the hardware performs an `exit` operation, which is the inverse of a `vmrun` operation. On *exit*, the hardware saves guest state to the VMCB, loads VMM-supplied state into the hardware, and resumes in host mode, now executing the VMM [1].

Hardware-assisted virtualization performance heavily depends on the number of VM exits. Because I/O operations require many VM exists, one way to optimize the performance is reducing them. In fact, a guest application that does not issue I/O operations runs effectively at native speed [2].

Hardware-assisted virtualization is also known as accelerated virtualization. Xen calls it hardware virtual machine (HVM).

1.6.3 Paravirtualization

Paravirtualization is a subset of server virtualization, which provides a thin software interface between the host hardware and the modified guest OS. Unlike full virtualization, in paravirtualization the running guest OS is aware of running in a virtualized environment and should be modified in order to be able to run on top of it. In this approach, the VMM is smaller and easier to implement because it only exposes a small software interface to virtual machines that is similar, but not identical to that of the underlying hardware. The intent of the modified interface is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment. Moreover, it is more trustworthy than one using binary translation and more similar to a virtual machine monitor that uses the trap-and-emulate technique. Furthermore, it shows performance closer to non-virtualized hardware. Device interaction in paravirtualized environment is very similar to the device interaction in full virtualized environment; the virtual devices in paravirtualized environment also rely on physical device drivers of the

underlying host [23].

Paravirtualization uses a different approach to overcome the x86 virtualization issues. With paravirtualization, the non-virtualizable instructions are replaced by virtualizable equivalent ones. This requires the guest OS to be changed. One difference with respect to the binary translation approach is that in paravirtualization, the guest OS knows that it is running in a virtual environment, while using binary translation the guest OS have the illusion that is running on a real machine.

Paravirtualization requires the guest operating system to be explicitly ported to the new API. As a consequence, a conventional OS that is not aware to be run in a paravirtualized environment cannot be run on top of a VMM that makes use of paravirtualization.

1.6.4 Nested virtualization

From a user perspective, nested virtualization (also called recursive virtualization [20]) is the ability of running a virtual machine within another. This can be extended recursively to arbitrary depths. From a technical point of view, it goes down to running one or more VMMs inside another VMM. If this procedure can be repeated until the resources of the system are consumed, then the original machine is recursively virtualizable [20].

There are many ways to implement nested virtualization on a particular computer architecture and they mostly depend on supported hardware-assisted virtualization capabilities. In case a particular architecture lacks proper hardware support required for nested virtualization, various software techniques are employed to enable it, at the cost of a performance loss.

1.7 Reasons to use virtualization

Virtualization and virtual machines offer several benefits. First of all, virtualization can provide *resource optimization* by virtualizing the hardware and allocating parts of it based on the real needs of users and

applications. This is particularly pointed towards enterprises [17], that usually have huge computational resources and struggle to manage the available computing power, storage space and network bandwidth in an efficient way. In the past, it was a common practice to dedicate individual computers to a single application. However, if several applications only use a small amount of processing power, the administrator can use virtualization to *consolidate* several computers into one server running multiple virtual environments [17] and thus reduce the amount of server resources that are under-utilized. Related benefits claimed by vendors include savings on hardware, environmental costs, management, and administration of the server infrastructure.

It happens from time to time to deal with legacy software. There are cases in which legacy applications or operating systems may not run on newer hardware because of compatibility issues [26]. VMs can solve the problem by recreating the environment these application expect. In fact, VMs can create virtual hardware configurations that differ significantly from the physical one of the underlying host.

Virtual machines are also increasingly being used used to provide secure, isolated sandboxes for running untrusted applications [26]. An interesting thing is that sandboxes can be created on the fly, as soon as an untrusted application must be run. In fact, virtualization is an important concept in building secure computing platforms; the only downside is a little decrease in performance. VMs can run in isolation with enhanced security, fault tolerance, and error containment. Moreover, VMs are used by software analysts to study the behavior of applications after they have been deliberately compromised.

In cloud scenarios, virtualization can enhance *security* thanks to the better separation of services that follows [23]. Using multiple virtual machines, in fact, it is possible to separate services by running one service on each virtual machine. If one service is compromised, the other services are unaffected. The server would contain a minimal install that could host several virtual machines. Each virtual machine consists of a minimal operating system and one service (e.g., a web server). In case the web server is compromised, the web pages hosted will be unreliable,

but the break in will not affect the remaining running services such as the database server, the mail server and the file server.

Virtualization is also popular in simulation, where multiple VMs can simulate big networks of independent computers, whose behavior can thus be observed and analyzed. VMs are, in fact, great tools for research and academic experiments. Since they provide isolation, they are safer to work with. They encapsulate the entire state of a running system: you can save the state, examine it, modify it, reload it, and so on [26]. The state also provides an abstraction of the workload being run. Moreover, VMs allow for powerful debugging and performance monitoring of operating systems. In fact, tools can be attached to the virtual machine monitor, for example, and the guest operating systems can be debugged without losing productivity, or setting up more complicated debugging scenarios. Virtual machines provide isolated, constrained, test environments to developers, that can reuse existing hardware instead of purchasing dedicated physical hardware.

In distributed network of computers, virtualization provides wide *flexibility* in several ways. It is possible to migrate a virtualized instance to another physical computer and the virtual instances can be handled gracefully from the host operating system with features like *pause*, *resume*, *shutdown* and *boot*. It is also possible to change the specifications of virtual computers while they are running, for example the amount of *Random Access Memory* (RAM), hard disk size and more.

Virtualization brings new opportunities to data center administration too. Guaranteed uptime of servers and applications, speedy disaster recovery if large scale failures do occur, instant deployment of new virtual machines or even aggregated pools of virtual machines via template images are common examples of the great advantages that virtualization provides. Moreover, it enables elasticity, i.e. resource provisioning when and where required instead of keeping the entire data center in an always-on state. Server virtualization provides a way to implement redundancy without purchasing additional hardware. If one virtual system fails, another virtual system takes over. By running the redundant virtual machines on separate physical hardware, virtualization can also

provide better protection against physical hardware failures, with tasks such as system migration[8], backup, and recovery being made easier and more manageable. Migration is typically used to improve reliability and availability: in case of hardware failure the guest system can be moved to a healthy server with limited downtime, if any. It is also useful if a virtual machine needs to scale beyond the physical capabilities of the current host and must be relocated to physical hardware with better performance. VMs make software easier to migrate, thus aiding application and system mobility.

Finally, VMs can be used to package an entire application. This simplifies the deployment of the application and its related dependencies while enabling multiple instances of it to be run in parallel over multiple VMs.

1.8 Brief history of virtualization

Virtualization is a technique that dates back to the 1960s, and continues to be an area of active development.

Virtual machines rely on virtual memory and time-sharing, concepts that were introduced between the late 1950s and the early 1960s.

1.8.1 Time-sharing

Time-sharing is the sharing of a computing resource among many users by means of multiprogramming and multi-tasking at the same time. Research activities started in the late 1950s at the Massachussets Institute of Technology (MIT). The first implementation of a time-sharing operating systems, known as *Compatible Time-Sharing System* (CTSS), was unveiled in November 1961 and the work was reported at the Spring Joint Computer Conference in May 1962 [6]. At the time, professor F. J. Corbató, the project leader of CTSS, was also to become one of the most

[8]Migration usually refers to the moving of a server environment from one place to another. It is very useful in cloud architectures.

prominent members of *Project MAC*[9], which then teamed with IBM to develop virtual machine solutions. In the early 1970s, time-sharing emerged as the prominent model of computing and is still considered one of the major technological shifts in the history of computing. Its popularity mainly resides in the ability to enable a large number of users to interact concurrently with a single computer.

1.8.2 Virtual memory

Virtual memory is a memory management technique that maps memory addresses used by a program, called virtual addresses, into physical addresses in computer main memory. Virtual memory makes processes see main memory as a contiguous address space. The mapping between virtual and real memory is manged by the operating system and the *Memory Management Unit* (MMU), an hardware component usually implemented as part of the central processing unit (CPU) that automatically translates virtual addresses to physical addresses.

Moreover, virtual memory enables the operating system to provide a virtual address space that can also be bigger than real memory so processes can reference more memory than the size that is physically present. This technique is called paging[10].

Virtual memory first appeared in the 1960s, when main memory was very expensive. Virtual memory enabled software systems with large memory demands to run on computers with less real memory. Soon, because of the savings introduced by this technique, many software systems started to turn on virtual memory by default.

[9]Project MAC (Project on Mathematics and Computation), financed by the Defense Advanced Research Projects Agency (DARPA), was launched on July 1, 1963. It would later become famous for its groundbreaking research in operating systems, artificial intelligence, and the theory of computation. Under Project MAC there was also an artificial intelligence group, directed by Marvin Minsky, which also included John McCarthy, the inventor of the Lisp programming language.

[10]Paging is a memory management scheme by which an operating system stores and retrieves data from secondary storage (e.g., hard disks) for use in main memory. The operating system retrieves data from secondary storage in same-size blocks called pages, hence the name paging.

The concept of virtual memory was first developed by German physicist Fritz-Rudolf Güntsch at the Technische Universität Berlin in 1956 [22]. Paging was first implemented at the University of Manchester as a way to extend the *Atlas Computer*'s working memory.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and increased memory availability for processes by exploiting the paging technique.

1.8.3 The early years of virtualization

In the early 1960s, IBM had a wide range of systems, each generation of which was substantially different from the previous. For customers, keeping up with the changes and requirements of each new system was painful. Moreover, at the time systems could only do one thing at a time. Batch processing was the only way to accomplish multiple tasks. However, since most of the users of those systems were in the scientific community, batch processing met the customers needs for a while, slowing down the research in the field. Later, because of the wide range of hardware requirements, IBM began working on the *System/360* (S/360) mainframe, designed as a broad replacement for many of their other systems, which also guaranteed backwards compatibility. When the system was first designed, it was meant to be a single user system to run batch jobs and did not support virtual memory.

Things began to change on July 1, 1963, when Massachusetts Institute of Technology (MIT) announced *Project MAC*. As part of the research program, MIT needed new computer hardware capable of more than one simultaneous user. IBM was refusing to develop a time-sharing computer because of the relatively small demand, while MIT did not want to adopt a specially modified system. General Electric (GE) on the other hand, was willing to make a commitment towards a time-sharing computer. For this reason MIT chose GE as their vendor of choice. Following the loss of this opportunity, IBM then started to take notice to the demand for such a system, especially when it heard of Bell Labs'

need for a similar system.

In late 1960s, in response to the need from MIT and Bell Labs, IBM designed the *CP-40* mainframe, the first version to run the CP/CMS (*Control Program/Cambridge Monitor System*) time-sharing operating system. CMS was a small single-user operating system designed to be interactive. CP was the program which created virtual machines. The idea was the CP ran on the mainframe and created virtual machines running the CMS, which the user would then interact with. The ability for the user to interact with the system was a key aspect of this generation of mainframes.

The CP-40 was never sold to customers, and was for laboratory use only. The CP-40 was an important milestone in the history of IBM's mainframes because it was intended to implement full virtualization by design. Doing so required hardware and microcode customization on a S/360-40[11], to provide the necessary address translation and other virtualization features.

Experience on the CP-40 project provided input to the development of the IBM *System/360-67*, announced in 1965. In 1966, CP-40 was reimplemented for the S/360-67 as CP-67, and by April 1967, both versions were in daily production use. The CP-67 was the first commercial mainframe to support virtualization. In 1968, CP/CMS was submitted to IBM Type-III Library by MIT's Lincoln Laboratory, making system available to all IBM S/360 customers at no charge in source code form.

On June 30, 1970, IBM announced the *System/370* (S/370) as the successor generation of mainframes to the *System/360* family. Work began on CP-370, a complete reimplementation of CP-67, for use on the S/370 series. As with the S/360 series, this new generation of mainframes did not support virtual memory. However, in 1972, IBM changed direction, announcing that the option would be made available on all S/370 models, and also announcing several virtual storage operating systems, including VM/370[12].

[11] S/360-40 was a model included in the initial S/360 announcement, in 1964.

[12] VM is a family of IBM virtual machine operating systems used on IBM mainframes *System/370*, *System/390*, *zSeries*, *System z* and others. The first version,

1.8.4 Modern virtualization on the x86 architecture

On February 8, 1999, VMware introduced *VMware Virtual Platform* for the Intel IA-32 architecture[13], the first x86 virtualization product, based on earlier research by its founders at Stanford University.

In the subsequent years, many of the virtualization products that are still being used were created:

- In 2001, VMware entered the server market with *VMware GSX Server* (hosted) and *VMware ESX Server* (hostless), the first x86 server virtualization products.

- In 2003, an initial release of first open-source x86 hypervisor, *Xen*, was released.

- On July 12, 2006, VMware releases *VMware Server*, a free machine-level virtualization product for the server market.

- On January 15, 2007, innoTek released *VirtualBox Open Source Edition* (OSE), the first professional PC virtualization solution released as open-source under the GNU General Public License (GPL).

- On February 12, 2008, Sun Microsystems announced that it had entered into a stock purchase agreement to acquire innoTek, makers of *VirtualBox*.

- In April 2008, VMware releases *VMware Workstation 6.5 beta*, the first program for Windows and Linux to enable *DirectX 9* accelerated graphics on *Windows XP* guests.

---

released in 1972, was VM/370, or officially *Virtual Machine Facility/370*. This was a *System/370* reimplementation of earlier CP/CMS operating system.

[13]IA-32 (Intel Architecture, 32-bit) is the 32-bit version of the x86 instruction set architecture (ISA), first implemented in the *Intel 80386* microprocessors, in 1985. IA-32 is the first incarnation of x86 that supports 32-bit computing.

1.9 Security vulnerabilities in virtualization

Most of security flaws identified in a virtual machine environment are very similar to the security flaws associated with any physical system. In fact, if a particular operating system or application configuration is vulnerable when running directly on real hardware, it will most likely also be vulnerable when running in a virtualized environment [28]. The use of virtualized systems adds some general security concerns, including:

- **Guest OS isolation**: Isolation is one of the primary benefits that virtualization brings. Guest OS isolation ensures that programs executing within a virtual machine may only access and use the resources allocated to it, and not covertly interact with programs or data either in other virtual machines or in the hypervisor [28]. If not carefully configured and maintained, isolation can also represent a threat for the virtual environment. The isolation level should be strong enough to contain potential break-ins into a compromised VM and to prevent it from gaining access either to the other virtual machines in the same environment or to the underlying host machine. As an example, while shared clipboard is a useful feature that allows data to be transferred between VMs and the host, it can also be treated as a gateway for transferring data between cooperating malicious program in VMs [23]. Furthermore, there exist virtualization technologies that do not implement isolation between the host and the VMs on purpose in order to support applications designed for one operating system to be operated on another operating system. The lack of isolation can be very harmful to the host operating system because it potentially gives unlimited access to the host's resources, such as file system and networking devices.

- **VM Escape**: The presence of the virtualized environment and the hypervisor may reduce security if vulnerabilities exist within it which attackers may exploit [28]. Such vulnerabilities could allow

programs executing in a guest to covertly access the hypervisor (together with the host OS, in case of a type-2 hypervisor), and hence other guest OS resources. This is known as VM escape and is perhaps the most serious threat to VM security. Moreover, since the host machine is running as root, the program which gain access to the host machine also gains the root privileges. This results in a complete break down in the security framework of the environment. [23]

- **Guest OS monitoring by the hypervisor**: The hypervisor has privileged access to the programs and data in each guest OS, and must be trusted as secure from compromised use of this access [28].

- **VM monitoring from the host**: The host machine is the control center of the virtual environment. There are implementations that enable the host to monitor and communicate with the applications running inside the VMs. Care should be taken when configuring the VM environment so that the isolation level is strong enough to prevent the host from being a gateway for attacking the virtual machines. [23]

- **Guest-to-Guest attack**: In case an attacker gains root privileges of the hypervisor/host from inside a virtual machine, then it can also access the other virtual machines and arbitrarily jump from one virtual machine to another [23].

- **VM monitoring from another VM**: It is considered as a threat when one VM without any difficult may be allowed to monitor resources of another VM. When comes to the network traffic, isolation completely depends on the network setup of the virtualized environment. If the host machine is connected to the guest machine by means of physical dedicated channel, then its unlikely that the guest machine can sniff packets to the host and vice-versa. However in reality the VMs are linked to the host machine by means of a virtual hub or by a virtual switch. In which case,

the guest machines may be able to sniff packets in the network or even worse redirect the packets going to and coming from another guest. [23]

- **Denial-of-Service**: Since guest machines and the underlying host share the physical resources such as CPU, memory disk, and network resource, it is possible for a guest to impose a Denial-of-Service (DoS) attack to other guests residing in the same system. DoS attack in virtual environment consists of a guest machine that consumes all the possible resources of the system, denying the service to other guests because there is no resource available for them to use.

- **Compromised VM snapshots**: Virtualization software often provides support for suspending an executing guest OS in a snapshot, saving that image, and then restarting execution at a later time, possibly even on another system. While this is useful in distributed contexts, it can also be very harmful. In fact, if an attacker succeeds in viewing or modifying the snapshot image, it can also compromise the security of the guest OS, together with the data and programs that run on top of it. [28]

1.10 Open-source implementations

The following are the most relevant open-source implementations in the field of virtualization.

1.10.1 Xen

Xen, first described in a SOSP 2003 paper called *Xen and the Art of Virtualization* [4], is an open-source type-1 hypervisor, which makes it possible to run many operating systems in parallel on a single machine. It is subject to the requirements of the GNU General Public License (GPL), version 2.

Architecture

Xen Project architecture is shown in Figure 1.3.

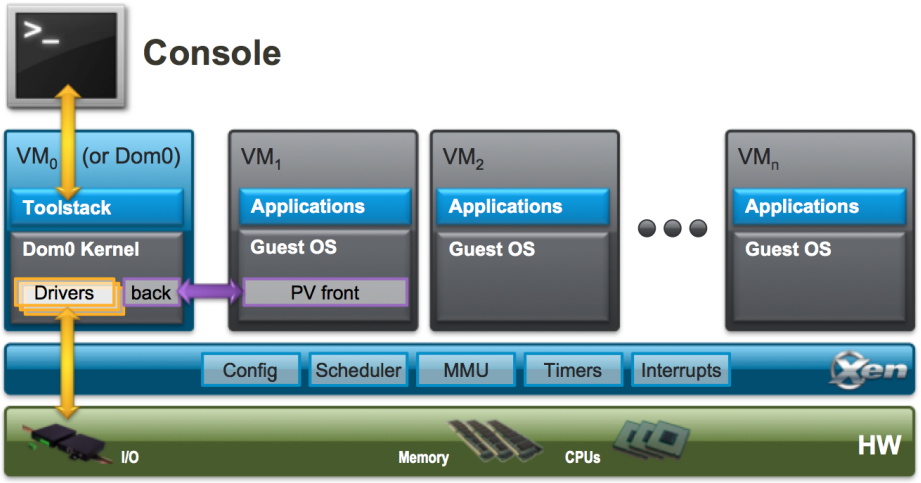


Figure 1.3: Xen Project architecture [29].

The following is a brief overview of each component [29]:

- **Xen** hypervisor is a thin software layer that runs directly on the hardware and is responsible for managing CPU, memory, and interrupts. It is the first program running after the bootloader exits. The hypervisor itself has no knowledge of I/O functions such as networking and storage.

- **Virtual Machines** are virtualized environments, each running their own operating system and applications. The Xen Project hypervisor supports two different virtualization modes: Paravirtualization (PV) and Hardware-assisted or Full Virtualization (HVM). Both guest types can be used at the same time on a single hypervisor. It is also possible to use techniques used for Paravirtualization in an HVM guest: essentially creating a continuum between PV and HVM. This approach is called PV on HVM. Guest VMs are totally isolated from the hardware: in other words, they have no privilege to access hardware or I/O functionality. Thus, they are also called unprivileged domain (DomU).

- **Domain 0 (Dom0)** is a specialized virtual machine that has special privileges like the capability to access the hardware directly, handles all access to the system's I/O functions and interacts with the other virtual machines. It also exposes a control interface to the outside world, through which the system is controlled. The Xen Project hypervisor is not usable without Dom0, which is the first virtual machine started by the system.

- **Toolstack**, resident in Dom0, allows a user to manage virtual machine creation, destruction and configuration. It exposes an interface that is either driven by a command line console, by a graphical interface or by a cloud orchestration stack such as *OpenStack* or *CloudStack*.

- **Dom0 Kernel** is a Xen Project-enabled kernel that powers Dom0.

- **Guest OS**: Paravirtualized guests require a PV-enabled kernel. Linux distributions that are based on recent Linux kernels are Xen Project-enabled and usually include packages that contain the hypervisor and the tools (the default toolstack and console). All but legacy Linux kernels are PV-enabled, capable of running PV guests.

The hypervisor supports virtualization of x86, x86-64, IA-64, ARM and other CPU architectures and has been used to virtualize a wide range of guest operating systems, including Windows, Linux, Solaris and various versions of the BSD operating system. Moreover, it is used as the basis for a number of different commercial and open-source applications, such as: server virtualization, Infrastructure-as-a-Service (IaaS), desktop virtualization, security applications, embedded and hardware appliances. The Xen Project hypervisor is powering the largest clouds in production today.

History

Xen 1.0 was officially released in 2004. At the time, Ian Pratt, senior lecturer at the University of Cambridge, and several other technology

leaders became involved with the project team. They founded a company known as XenSource, which was later acquired by Citrix in 2007 in order to convert the hypervisor from a research tool into a competitive product for enterprise computing. The hypervisor remained an open-source solution and has since become the basis of many commercial products.

In 2013, the project went under the umbrella of the Linux Foundation. Accompanying the move, a new trademark *Xen Project* was adopted to differentiate the open-source project from the many commercial efforts which used the older *Xen* trademark. [29]

As of 2016, Xen is the only type-1 hypervisor that is available as open-source. The current stable release, version 4.7, was released June 20, 2016.

1.10.2 Kernel-based Virtual Machine (KVM)

Kernel-based Virtual Machine (KVM) is a full virtualization software for Linux on x86 hardware. It is implemented as a loadable kernel module, `kvm.ko`, that provides the core virtualization infrastructure and a processor specific module, `kvm-intel.ko` or `kvm-amd.ko`. When the two kernel modules are loaded, the Linux kernel effectively acts as a type-1 hypervisor. [14]

Using KVM, one can run multiple virtual machines that use unmodified Linux or Windows images. Each virtual machine has private virtualized hardware (e.g., network cards, disks, and graphics card). Paravirtualization support for certain devices is available for Linux, Windows and other guests using the *VirtIO* API.

KVM is open-source software. The kernel component of KVM is included in mainline Linux, as of 2.6.20 (which was released February 5, 2007). The userspace component of KVM is included in mainline QEMU, as of version 1.3. KVM has also been ported to FreeBSD in the form of loadable kernel modules. KVM actually supports the following architectures: x86, x86-64, IA-64, ARM, PowerPC, S/390.

KVM and Xen serve the same purpose, however there are many

differences. Xen is a stand-alone hypervisor and as such it assumes control of the machine and divides resources among guests. On the other hand, KVM is part of Linux and uses the regular Linux scheduler and memory management. This means that KVM is much smaller. KVM only run on processors that supports x86 HVM[14] (e.g., Intel *VT-x* and AMD *AMD-V*) whereas Xen also allows running modified operating systems on non-HVM x86 processors using the paravirtualization technique. While KVM does not support paravirtualization for CPU, it may support paravirtualization for device drivers to improve I/O performance. [14]

KVM implements virtual machines as regular Linux processes. They inherit memory management features of the Linux kernel, including swapping and memory page sharing[15] [7]. Moreover, all the standard Linux process management tools can be used. For instance, it is possible to pause, resume or even kill a VM with the `kill` command or monitor its resource usage with `top`.

KVM uses QEMU to emulate motherboard hardware, like memory controller, network interface, ROM BIOS and other interfaces. QEMU is a machine emulator that runs an unmodified target operating system and all its application in a virtual machine. The primary usage of QEMU is to run one operating system on another. QEMU uses emulation, while KVM uses processor extensions for virtualization [7, 14]. KVM exposes the `/dev/kvm` interface, which a userspace host can then use to set up the guest VM's address space[16], feed the guest simulated I/O, and map the guest's video display back onto the host.

[14]Hardware Virtual Machine (HVM) is a vendor-neutral term often used to designate the x86 instruction set extensions that provide hardware assistance to virtual machine monitors. They enable running fully isolated virtual machines at native hardware speeds, for some workloads.

[15]Memory page sharing is supported by a kernel feature called kernel same-page merging (KSM). It scans for and merges identical memory pages occupied by virtual machines. Whenever a guest OS wants to modify that page it is provided its own copy.

[16]The host must also supply a firmware image (e.g., a BIOS) that the guest can use to bootstrap into its main OS.

1.10.3 QEMU

QEMU is a generic and open-source machine emulator and virtualizer. It has two operating modes:

- **Full system emulation**: In this mode, QEMU emulates a full system, including one or several processors and various peripherals (e.g., memory controller, network interface, ROM BIOS). It can be used to launch different operating systems without rebooting the computer or to debug system code [14]. In this mode, a target operating system and all its application can run unmodified in a virtual machine [7].
- **User mode emulation**: In this mode, QEMU can launch processes compiled for one CPU on another one to ease cross-compilation and cross-debugging [14].

When used as a machine emulator, QEMU can run operating systems and programs made for one architecture on a different one. It emulates CPUs through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems.

When used as a virtualizer, QEMU runs virtual machines at near-native speed by executing the guest code directly on the host CPU thanks to its hardware virtualization extensions. QEMU can also be used purely for CPU emulation for user-level processes, allowing applications compiled for one architecture to be run on another. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. Finally, QEMU can make use of KVM when running a target architecture that is the same as the host architecture. [21]

1.10.4 Oracle VM VirtualBox

Oracle VM VirtualBox (formerly *VirtualBox*) is a free and open-source type-2 hypervisor, and is considered one of the most popular virtualization solutions for x86 desktop computers.

VirtualBox was initially a proprietary solution offered by Innotek GmbH. On February 20, 2008, Sun Microsystems completed the acquisition of Innotek GmbH. In January 2010, following the acquisition of Sun Microsystems by Oracle Corporation, the product was finally re-branded as Oracle VM VirtualBox.

VirtualBox is a cross-platform virtualization application. It can be installed on Intel or AMD-based computers running Linux, OS X, Solaris and Windows host operating systems. It supports the creation and management of guest virtual machines running versions and derivations of Linux, FreeBSD, OpenBSD, Solaris, Windows, and others. Virtual machines running unmodified versions of OS X as the guest operating system are supported exclusively on Apple hardware running OS X as the host operating system. In theory, since VirtualBox is designed to provide a generic virtualization environment for x86 systems, it may run operating systems of any kind. [18]

VirtualBox can run in parallel as many virtual machines as the disk space and memory limits permit. The host can pause and resume each guest at will, and is able to take snapshots of each of these guests for backup purposes. Each of the virtual machines can be configured independently and can run in either software emulation mode or hardware assisted mode, using Intel *VT-x* or AMD *AMD-V* technology.

VirtualBox also supports hardware emulation. For example, hard disks can be stored as disk image files on the host. When a guest operating system reads from or writes to a hard disk, VirtualBox redirects the request to the image file. VirtualBox supports four variants of disk image files: Virtual Disk Image (VDI) is VirtualBox's container format for guest hard disks; VMDK, VHD, and HDD disk image formats are also supported. Other peripherals (e.g., CD/DVD/BD drives) can be emulated or attached to the real hardware of the host. In addition VirtualBox emulates ethernet network adapters, which enables each guest to connect to the internet through a NAT interface. Finally, for some guest operating systems, an optional package (called Guest Additions), to be installed inside a virtual machine after the guest operating system has been installed, is provided. It consists of device drivers and

system applications that optimize the guest operating system for better performance and usability. [18]

The Guest Additions extend the base features of VirtualBox with: mouse pointer integration, shared folders, better video support with 2D and 3D graphics acceleration, seamless windows, time synchronization and shared clipboard.

Chapter 2

OS-level virtualization

Operating-system-level virtualization (OS-level virtualization, for short) is a modern and lightweight virtualization technique that relies on particular features of the kernel of an operating system to provide multiple, conceptually isolated, user-space instances (i.e., containers). This kind of virtualization does not involve a hypervisor or a separate guest operating system. All containers that are present on a host machine run either on the same host kernel or on a copy of it.

The aim is to provide an environment similar to those provided by a virtual machine (VM), but with a very low overhead. Containers are usually faster, because they do not need to run a separate kernel on virtual hardware, but less secure than VMs because vulnerabilities in the kernel of the host are shared between the containers. Moreover, because of the shared kernel, the guest OS can only be one that uses the same kernel as the host OS. For instance, if the host OS is a Linux distribution, the guests can only be Linux distributions running the same kernel version.

In Linux based operating systems, containers are built upon crucial kernel features like control groups and namespaces. The kernel often provides resource management features and security mechanisms, like Mandatory Access Control (MAC), to provide enhanced security.

2.1 Linux kernel containment features

Linux containers are an operating-system-level virtualization method for running multiple isolated Linux systems on a host using a single Linux kernel. Today, many different implementations exist. Most of them rely on the Linux kernel *namespaces* and *control groups*, two subsystems that are at the core of lightweight process virtualization. Namespaces isolate the applications within different userspaces such as network, processes, users themselves and the filesystem. Control groups, on the other side, limit host hardware resources, such as CPUs, memory, and disks.

The following sections are used to give a quick overview of the main Linux kernel containment features that are adopted by the most prominent container implementations.

2.1.1 chroot

On Unix-like operating systems, a `chroot` is an operation that changes the root directory (denoted by the `/` sign) of the calling process and all of its child processes to a specified path, that is usually a subdirectory of the real root directory of the filesystem. This new restricted environment is called *chroot jail*. Despite the name, there are circumstances in which the calling process can escape it.

The term `chroot` is used to refer both to the `chroot(2)` system call and the `chroot(8)` user command. The `chroot(2)` system call, introduced during development of Version 7 Unix in 1979 and added to BSD in 1982, is often referred to as the first implementation of an operating-system-level container. However, as can be read in the designated reference manual page[1] [16], the system call only affects the pathname resolution process and thus it is not intended to be used for any kind of security purpose, neither to fully sandbox a process nor to restrict filesystem system calls.

As of Linux kernel 4.8.14, released December 10, 2016, the source

[1]The `chroot(2)` reference manual page is accessible on a Linux distribution by typing `man 2 chroot` in a terminal.

code of the system call can be found in the following files of the Linux kernel tree:

- Function prototype:
  `include/linux/syscalls.h`, line 473.
- Generic syscall table entry:
  `include/uapi/asm-generic/unistd.h`, lines 170–171.
- Architecture-specific sycall table entry (x86_64 architecture):
  `arch/x86/entry/syscalls/syscall_64.tbl`, line 170.
- Implementation:
  `fs/open.c`, lines 469–500.

The implementation of the `chroot` user command is provided by the GNU Core Utilities (also known as `coreutils`) project[2], instead.

A chroot jail can serve different purposes. As an example, it is a way to test a software in a minimal environment that only includes the library dependencies that are strictly required. Moreover, there are circumstances in which legacy software can't be run on the host operating system because of dependency conflicts and thus chroot jails are used as a fast and simple workaround to run it on those systems. Finally, chroot can be used as a tool to fix a system that for any reason has become corrupted or unbootable; after booting a live Linux distribution from an optical disc or a USB drive, it is as easy as mounting the partitions of the corrupted system and perform a chroot to regain control of the system and fix what caused it not to boot properly.

Minimal chroot example

A minimal `chroot` example consists of a chroot jail that only contains a shell executable – in this example we will use the GNU Bourne-Again SHell (i.e., `bash`) for its popularity but we could have chosen any other Unix shell – along with its library dependencies. As a consequence,

[2] https://www.gnu.org/software/coreutils/coreutils.html

the user inside the chroot environment can only enter the built-in `bash` commands. This example was created on a virtual machine running Ubuntu 16.10 64-bit. On other Unix-like operating systems many commands may have a different name or may need to be manually installed.

We know that the `bash` executable is located under `/bin/bash`, but we need to discover its library dependencies. To this end, there is the `ldd` command that outputs the shared libraries of the program specified on the command line:

```
fantox@ubuntu-vm:~$ ldd /bin/bash
        linux-vdso.so.1 =>  (0x00007ffd03edf000)
        libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f447404a000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f4473e46000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4473a7f000)
        /lib64/ld-linux-x86-64.so.2 (0x0000559bc9253000)
```

Now we need to create a new directory that will be used as the root directory of the chrooted environment (we will name it *newroot*) and then copy inside of it both the `bash` executable together with all its dependencies. While this can be done manually by using the `cp` command for each single library dependency, it is convenient to filter and manipulate (using the `grep` and `sed` commands, respectively) the output of the previous `ldd /bin/bash` command to obtain the library dependencies' absolute paths and then pass them, as arguments, to the `cp` command:

```
fantox@ubuntu-vm:~$ mkdir newroot
fantox@ubuntu-vm:~$ cp --parents /bin/bash newroot/
fantox@ubuntu-vm:~$ ldd /bin/bash \
> | grep '/' \
> | sed 's/\(.*> \)\|\t//;s/ .*//' \
> | xargs -I '{}' cp --parents '{}' newroot/
fantox@ubuntu-vm:~$ tree --charset ascii newroot/
newroot/
|-- bin
|   `-- bash
|-- lib
|   `-- x86_64-linux-gnu
|       |-- libc.so.6
|       |-- libdl.so.2
|       `-- libtinfo.so.5
`-- lib64
    `-- ld-linux-x86-64.so.2

4 directories, 5 files
fantox@ubuntu-vm:~$ sudo chroot newroot/
[sudo] password for fantox:
bash-4.3# pwd
/
bash-4.3# cd
```

```
bin/   lib/   lib64/
bash-4.3# cd bin/
bash-4.3# pwd
/bin
bash-4.3#
```

The `tree` command was used to list the content of the *newroot* directory in a graphical tree-like format and finally, the `chroot` command was used to enter the new environment. As can be seen by the console output, the working directory was automatically set to the new root and the root directory of the chroot jail matches the `newroot` directory that we created inside the home of the host system.

Security considerations

For security reasons, chroot jails should be run as a non-root user. This can be done by adding the option `--userspec=USER:GROUP` to the `chroot` command. Moreover, depending on the operating system and the implementation of the `chroot` command that is shipped with it, the working directory may or may not automatically be set to the new root. As a general rule, it is a good practice to manually change the working directory to the new root before issuing the `chroot` command to deny the jail access to resources outside of the new root. Other good practices include keeping the jail as small as possible and limiting the permissions of the files and directories inside the new root.

2.1.2 Namespaces

Generally speaking, a namespace is a collection of entity names that can be categorized in sets. Namespaces are usually organized as hierarchies.

A Linux namespace abstracts a global system resource, so that processes within the same namespace are given the illusion to have exclusive access to the global resource. In reality, they own an isolated instance of the global resource, so that changes to it only affect processes within the namespace. Linux namespaces partition processes, users, network stacks and other components into separate analogous pieces in order to provide processes a unique view. This is why Linux namespaces are

the fundamental building block of container implementations on Linux systems.

As of Linux kernel version 4.8.14, there are seven kinds of namespaces [16]:

Namespace	**Constant**	**Isolates**	**Symbolic link in `/proc/[pid]/ns/`**
Cgroup	`CLONE_NEWCGROUP`	Cgroup root directory	`cgroup`
IPC	`CLONE_NEWIPC`	System V IPC[3], POSIX[4] message queues	`ipc`
Mount	`CLONE_NEWNS`	Mount points	`mnt`
Network	`CLONE_NEWNET`	Network devices, stacks, ports, etc.	`net`
PID	`CLONE_NEWPID`	Process IDs	`pid`
User	`CLONE_NEWUSER`	User and group IDs	`user`
UTS	`CLONE_NEWUTS`	Hostname and NIS[5] domain name	`uts`

After a system is finished booting, additional namespaces can be created and processes can join one or more namespaces. Despite the different kinds, all namespaces work exactly the same way: each process is associated with a namespace, that in turn is associated with a view

[3]Interprocess communication (IPC) refers to the mechanisms provided by an operating system to let processes share data.

[4]The Portable Operating System Interface (POSIX) is a family of standards for maintaining compatibility between operating systems. *POSIX.1-2008* defines a standard operating system interface and environment, including a command interpreter (or *shell*), and common utility programs to support applications portability at the source code level. [12]

[5]Network Information Service (NIS) is a protocol originally invented by Sun Microsystems for distributing shared configuration files between computers on a computer network.

of a resource that is shared between the processes that are part of the same namespace and isolated with respect to the other namespaces.

The namespaces API includes three system calls, that are used to create and interact with the various kernel namespaces: `clone(2)`, `setns(2)`, and `unshare(2)`. The `clone(2)` system call is used to create a new process. It has a `flags` argument where it is possible to specify one ore more of the namespace constants shown in table above; as a consequence, new namespaces are created for each constant, and the child process automatically joins all of them. The `setns(2)` system call allows the calling process to join an existing namespace, while the `unshare(2)` system call moves the calling process to a new namespace and provides a `flags` argument that has the same meaning as in the previously described `clone(2)` system call.

NOTE: With the exception of user namespaces, the creation of a namespace through `clone(2)` or `unshare(2)` requires a capability named `CAP_SYS_ADMIN`, that is essentially equivalent to having root access.

Each process has a `/proc/[pid]/ns/` subdirectory containing one symbolic link[6] for each namespace that supports being manipulated by `setns(2)`. Each symbolic link is a handle for the corresponding namespace of the process. Opening one of the files in this directory returns a file descriptor for the corresponding namespace of the process specified by `pid`, that can be passed to to `setns(2)` as the first argument to make the calling process join the namespace.

Finally, it is worth noting that the user namespace enables a non-root user to create a process in which it will be root. This comes in handy while implementing containers.

As the Linux kernel was not designed with namespaces in mind, they should be considered a work in progress. In fact, they actually do not cover all the relevant areas of the kernel and, in future versions, will probably grow in number.

[6]Files in `/proc/[pid]/ns/` appear as symbolic links since Linux 3.8, in past releases they were hard links.

2.1.3 Control groups

Control groups (abbreviated, *cgroups*), are a Linux kernel feature for limiting and monitoring the usage of many kinds of resources by a process or a collection of processes. Processes, in fact, can be organized into tree-based hierarchical groups and control groups are essentially a mechanism to provide resource control to them.

The Linux kernel exposes an interface to control groups in the form of a pseudo-filesystem – similar to `/proc` and `/sys` – called `cgroupfs`. As a consequence, no new system calls were created to support them.

Starting from version 2.6.24 of the kernel, when a first implementation of control groups was added to the Linux mainline, processes can be grouped and partitioned, with child processes belonging to the same group as their parent process. Resource usage can be limited by means of kernel components called *subsystems* or *resource controllers*. Various subsystems have been implemented, one for each kind of resource that control groups deal with, such as CPU, memory and I/O in general.

For every subsystem, there is a hierarchy of cgroups. Attributes (e.g., limits) can be expressed at each level of the hierarchy and are considered as upper bounds throughout the subhierarchy underneath the cgroup where the attributes are defined, so lower levels can ovverride limits only with values that are smaller than those defined at higher levels. This hierarchy is managed by creating, removing, and renaming subdirectories within the cgroup pseudo-filesystem.

It is important to note that despite being a useful feature (as kernel namespaces are) that come in handy while implementing containers, control groups were not conceived with containers in mind and were originally only a feature to create groups of processes and limit their resource usage.

Cgroups version 1 and version 2

It all started in 2006 when Paul Menage, Rohit Seth, and other engineers at Google created a new project named *process containers*, finally renamed to *control groups* in 2007 to avoid confusion with other mean-

ings of the word *container*. Control groups initial release was in Linux 2.6.24 (January 24, 2008).

Soon Linux kernel developers started to take an interest in this new feature and in subsequent years various cgroup controllers have been created to extend management capabilities to many kinds of resources. However, the fast and uncoordinated development of controllers lead to inconsistencies in the first implementation of control groups (called version 1), and forced developers to start a new implementation (version 2) to address the problems with the previous version.

Development of cgroups version 2 started in 2013 as an alternative, orthogonal implementation to the first one. It was merged into kernel mainline in Linux 4.5, released March 14, 2016. The main difference between the two versions is that in version 1 each controller may be mounted against a separate cgroup filesystem that provides its own hierarchical organization of the processes on the system and cgroup membership is done on a per task (or thread, from a user-space perspective) basis. The ability to independently manipulate the cgroup memberships of the threads in a process caused problems. To address the issues, version 2 was designed to have only a single process hierarchy (often referred to as the *unified hierarchy*) and discriminates between processes, so that when the cgroup membership of a process is changed, it is also changed for all its threads.

Despite being conceived as a replacement for the initial version, it is high unlikely that the new version will take over it shortly to not break compatibility. Moreover, not every controller available in version 1 is currently implemented in version 2. This is not an issue since, with the only limitation of not being able to use the same controller in a version 1 hierarchy and in the version 2 hierarchy, a mix of version 1 and version 2 controllers can be used together on the same system with no issues.

Implementation

Control groups implementation consists of a core part, that is shared between cgroups v1 and v2, and many controllers.

The core part of cgroups – also known as the generic process-grouping system – is implemented in `kernel/cgroup.c`, which consists of more than 6.5K-LOCs[7].

As of Linux kernel version 4.8.14, there are twelve cgroups version 1 controllers, shown in the following list along with their respective kernel configuration options and a brief description [16]:

- `cpu`: With the `CONFIG_CGROUP_SCHED` kernel configuration option, this controller is used to guarantee a certain level of CPU usage *when the system is busy*. With the `CONFIG_CFS_BANDWIDTH`, the controller makes it possible to define an upper limit on the CPU time allocated to the processes in a cgroup within each scheduling period, instead.

- `cpuacct` (`CONFIG_CGROUP_CPUACCT`): This controller provides accounting for CPU usage by groups of processes.

- `cpuset` (`CONFIG_CPUSETS`): This controller enables binding between the processes in a cgroup and a set of CPUs and NUMA[8] nodes.

- `memory` (`CONFIG_MEMCG`): This controller can be used to enable reporting and limiting of process memory, kernel memory, and swap used by a cgroup.

- `devices` (`CONFIG_CGROUP_DEVICE`): This controller defines which processes may create – with the `mknod(2)` system call – and/or open devices for reading or writing. There is usually a whitelist of accessible devices (e.g., `/dev/null`).

- `freezer` (`CONFIG_CGROUP_FREEZER`): This controller can suspend and restore all processes in a cgroup, along with its children.

[7] Six thousand five hundred (6.5K) lines of code (LOC).

[8] Non-uniform memory access (NUMA) is a particular kind of memory design used primarily on servers to enhance memory access time. In fact, with NUMA, a processor accesses its own local memory way faster than memory shared between processors or memory that is local to another one.

- `net_cls` (`CONFIG_CGROUP_NET_CLASSID`): This controller is used to place a *classid*, specified for the cgroup, on network packets created by a cgroup and then use the classids in firewall rules for packets leaving the cgroup.

- `blkio` (`CONFIG_BLK_CGROUP`): This controller can limit access to specified block devices by applying I/O control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy.

- `perf_event` (`CONFIG_CGROUP_PERF`): This controller allows performance monitoring (with the `perf` user command) of the set of processes in a cgroup.

- `net_prio` (`CONFIG_CGROUP_NET_PRIO`): This controller allows priorities to be specified, per network interface, for cgroups.

- `hugetlb` (`CONFIG_CGROUP_HUGETLB`): This controller has the ability to limit the use of huge pages by cgroups.

- `pids` (`CONFIG_CGROUP_PIDS`): This controller can be used to limit the number of processes that may be created in a cgroup and its children.

Version 2 of control groups actually implements only three controllers: CPU, memory, and I/O. This is a work in progress and many other controllers are expected in the upcoming years.

2.1.4 Capabilities

On Unix-like operating systems (including Linux), the user with user identifier (UID) 0 – also known as *root* – has unrestricted control over the system. This, however, is not the only circumstance in which a user or a program have complete control over the system. In fact, following the principle of least privilege, Unix-like systems provide mechanisms to let unprivileged users temporarily assume higher privileges to perform tasks that need them. For instance, an unprivileged user running `sudo`

to temporarily act as the root user, or a binary file owned by root and with the `setuid` flag set is given the same unrestricted access to system resources as it is the case with the root user. Binary files owned by root and with the `setuid` flag set can be extremely harmful, indeed, because potential vulnerabilities in their code can be exploited by an unprivileged user to gain permanent root-level access to the system.

Traditional Unix-like implementations distinguish between privileged and unprivileged processes, the former being those whose effective UID is 0 and the latter all the remaining. Needless to say, privileged processes are potentially more harmful because they bypass all kernel permission checks.

Over the years, Linux and Unix-like systems in general suffered many attacks because of privilege escalation vulnerabilities, typically found in binaries run with effective UID 0.

To reduce the attack surface and extend the least privilege principle to privileged processes, *capabilities* were added to the Linux kernel starting with version 2.2, released in 1999, to provide fine-grained control over privileged permissions, allowing use of the root user to be avoided. In fact, from kernel version 2.2 onward the privileges traditionally associated with root are divided into distinct units, called capabilities, which are attributes that can be independently enabled and disabled on a per-thread basis.

Capabilities are implemented on Linux using extended attributes. As of Linux kernel version 4.8, almost forty capabilities have already been defined. For instance, `CAP_SYS_CHROOT` is the capability that a thread needs to have in order to use the `chroot(2)` system call. File capabilities can be examined with the `getcap(8)` and set with the `setcap(8)` system administration commands [16].

Because of their ability to extend the principle of least privilege to processes that usually need only to use a subset of the root privileges, capabilities are a fundamental kernel building block of all the most prominent container technology implementations.

2.1.5 AppArmor

AppArmor and SELinux (the latter will be explained in the next section) are Linux kernel security modules that provide mechanisms for supporting access control security policies, like Mandatory Access Control (MAC). They are part of the mainstream Linux kernel as Linux Security Modules (LSM) components. LSM is a framework for the Linux kernel that adds support to a variety of computer security models by loading specific modules. LSM provides hooks from within the Linux kernel to a loadable module (e.g., AppArmor and SELinux) at every point in the kernel where a user-level system call is about to result in access to an important internal kernel object, allowing the module to apply its mandatory access controls. AppArmor is not meant to replace the traditional Discretionary Access Control (DAC) of Unix systems but rather to extend it with means of Mandatory Access Control (MAC).

What AppArmor does is proactively protect the operating system and applications from external or internal threats by enforcing good behavior and preventing even unknown application flaws from being exploited [3]. In fact, AppArmor restricts programs' access to a limited set of resources by means of per-program sets of access control attributes, called AppArmor policies (or, simply, profiles), that are typically loaded into the kernel at boot time. Profiles are plain text files that completely define what system resources individual applications can access and with what privileges. They support comments and can be manually edited quite easily. Moreover, they may use variables, even those defined outside the profile thanks to the ability to include other files.

AppArmor profiles can be loaded either in *enforcement mode* or in *complain mode* (also known as *learning mode*). The difference is that profiles loaded in complain mode will only report policy violation attempts while those loaded in enforcement mode will also effectively result in enhanced security. Mixing of enforcement and complain mode profiles is allowed. In both modes, policy violation attempts are logged using either `syslog` or `auditd`.

AppArmor comes with a set of default policies that are suitable for a wide range of programs. New policies can be created either by hand or using the learning mode, that basically allows for the creation of a profile by running a program normally and dynamically learning its typical behavior. While the resulting profile may not be as strict as an hand-crafted one, it still adds a substantial layer of security. Furthermore, the ability to create, edit and apply policies without the need to reboot the system is very useful, especially while dealing with a program whose policy is under construction.

Systems that use AppArmor should promote confinement by applying a policy to every program installed on the system. This is a good practice, however it is not mandatory. In fact, AppArmor's confinement is selective in the sense that while some programs on the system may be confined, others may not. AppArmor augments traditional DAC in that confined programs are evaluated under traditional DAC first and if DAC allows the behavior then the AppArmor policy is consulted. [3]

AppArmor is easier to learn with respect to SELinux because it is path-based and supports include files to speed up development and simplify profiles. Common file permissions include: read, write, append, execute (and many variations of it), memory map executable, lock, and link. Other permissions – like create, delete, chown and chmod – are currently in development [3]. Access controls for capabilities and networking are supported too. Finally, access control rules can be defined at different levels of granularity so that when two or more rules can be used to decide the level of access to a resource, only the most specific rule matches. This is very similar to what most packet filters do.

From Linux kernel version 2.6.36 onward, the core of AppArmor is part of the mainline kernel. Ubuntu and other Linux distributions include AppArmor by default.

There are two main versions of AppArmor, the current 2.x series and the development 3.x series. The forthcoming version will allow for a much expanded policy and fine-grained control over the current 2.x version. [3]

2.1.6 SELinux

Security Enhanced Linux (SELinux) is an implementation of Mandatory Access Control (MAC) on Linux. Similar to AppArmor, SELinux is a Linux kernel security module that is aimed at extending the traditional Discretionary Access Control (DAC) of Unix systems with an additional layer of access control.

SELinux controls access between applications and resources. The resources whose access SELinux can constrain include files, but are not limited to them. In fact, with SELinux the administrator of a system is able to define how applications and users can access various kinds of resources including files, devices, networks and inter-process communication.

Unlike standard DAC that lets both the user and the applications that the user runs change the file modes (e.g., read, write and execute bits), SELinux access controls – as in AppArmor – are determined by a policy that, in theory, should prevent unauthorized users and applications from changing it. Moreover, the access controls have a finer granularity; for instance, file access controls also include who can unlink, move, and append data to a file.

Basic SELinux concepts are: users, roles, types, contexts, object classes, and rules. The first thing to take note of is that a SELinux user is not equivalent to a Linux user. First, a SELinux user do not change during a user session, whereas a Linux user might change it (e.g., by using `su` or `sudo`). Secondly, even if it is possible to have a one-to-one Linux user to SELinux user mapping – as with the root Linux user and the root SELinux user – the typical mapping between SELinux users and Linux users is one-to-may. By convention, SELinux users that are generic have a particular suffix (`_u`), such as `user_u`. Finally, a SELinux might assume on one or more roles, that are defined by the policy. Objects typically have the role `object_r`. By convention, roles have the suffix `_r`, such as `user_r`.

Every process is given a type (also known as domain), that is used to determine access to resources. By convention, a type has the suffix `_t`,

such as `user_t`. Every process and object in the system has a context (often referred to as label). This is an attribute used to determine if an access should be allowed between a process and an object. A SELinux context is made of four fields, the last one being optional:

```
user:role:type:range
```

Object classes, such as `dir` for directories and `file` for files, are instead used in the policy and in access decisions as a fine-grained way to specify what access is allowed. In fact, each object class has a special set of permissions which are the possible ways to access these objects. For example, `read`, `write`, `create` and `unlink` (i.e., delete) are the permissions of the `file` object class.

SELinux primary security mechanism is type enforcement, in which rules are specified using both the type of the process and the object. Type enforcement, in fact, works by adding a label to every single resource of the system. The label of a resource includes its type. For instance, the type `user_home_t` is commonly used to label the files that are inside the home directory. Running applications have labels too; as an example, the Firefox web browser may be running as `firefox_t`. The type enforcement allows the system administrator to easily specify what application label can access what resource label. If an application is given a custom label (e.g., `firefox_t` for the Firefox web browser), then SELinux basically lets you define what that application is allowed to do with resources that have a given label:

```
allow firefox_t user_home_t : file { read write };
```

In this simple example, the rule states that the `firefox_t` type (i.e., the Firefox web browser) is allowed to read and write the files with the `user_home_t` type, that are basically those inside the user's home directory [24].

SELinux can be used to achieve different security goals, like sandboxing applications or restricting users to access only the resources they need to get their work done. Many distributions, including Fedora and Red Hat Enterprise Linux, do not need the system administrator to manually write policies because they come with many predefined poli-

cies which allow most applications to do everything necessary in their default configurations with no changes at all. On the other side, custom configurations may need a policy update, that usually only involves resource relabelling instead of a completely new policy.

To encourage the creation of strong policies and avoid duplicate work, a side project named SELinux Reference Policy [25] has been created. The aim of the project is to provide users a complete SELinux policy that can be used as the system policy for a wide range of systems and as a reference example for creating other custom policies. Today, since all predefined policies included in major distributions are based off of the Reference Policy, system administrators in need of a custom policy can choose between creating a new policy from the Reference Policy or editing an existing one (e.g., to make it stricter).

Like AppArmor, SELinux also has two modes of operation: *enforcing mode* actively applies the policy and ensures the system is being protected by SELinux; *permissive mode*, on the other side, does not deny failed accesses but only logs them. The former mode is somehow similar to AppArmor's *enforcement mode*, while the latter resembles *complain mode*.

2.1.7 seccomp

seccomp (short for "Secure Computing") is a computer security facility that is part of the Linux kernel mainline since kernel version 2.6.12 (released March 8, 2005). It provides a useful mechanism for reducing to the minimum the exposed kernel attack surface. Without seccomp, a userland process is normally given a wide set of system calls that, however, is wider than the minimal one needed by the process to accomplish its tasks. In fact, many of the exposed system calls are usually never unused by a process[9].

Starting with Linux 3.17, seccomp implementation include a system call, named `seccomp(2)`, and many library functions. The system call is used to change the seccomp state of the calling process. Ac-

[9] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

tually, only two modes of operation, *strict* and *filter*, are supported. They can be enabled by setting the first argument of the system call to `SECCOMP_SET_MODE_STRICT` or `SECCOMP_SET_MODE_FILTER`, respectively.

In strict mode of operation, the only system calls that the calling is given access are `read(2)`, `write(2)`, `_exit(2)` (with the exclusion of `exit_group(2)`) and `sigreturn(2)`. Any other system call triggers a `SIGKILL` signal, that causes it to terminate immediately[10]. To use this mode, the kernel must configured with `CONFIG_SECCOMP` enabled. Moreover, second and third arguments of the system call must be set to `0` and `NULL`, respectively. Processes that need to execute untrusted byte code, perhaps obtained by reading from an input device, are encouraged to use this mode.

Filter mode of operation, on the other side, makes it possible to filter the system calls that are available to userland applications. To this end, a pointer to a Berkeley Packet Filter (BPF) is passed as the third argument to `seccomp(2)`. This argument, in fact, is a pointer to a struct `sock_fprog` where the system calls to filter, along with their arguments, can be specified. This mode of operation is available only if the kernel is configured with `CONFIG_SECCOMP_FILTER` enabled.

While seccomp alone does not provide sandboxes, it is an important tool that developers can use in conjunction with other Linux security modules, like AppArmor and SELinux, to create sandboxes. LXC and Docker are two prominent examples of container technology implementations that make use of seccomp.

2.2 Containers

Container technologies, as anticipated, make use of the many Linux kernel containment features to provide isolated environments. In the following sections, LXC and Docker container implementations will be introduced and analyzed.

[10]The `SIGKILL` signal cannot be caught, blocked, or ignored [16].

2.2.1 LXC

LXC is an open-source user-space implementation of container technology. As an operating-system-level virtualization technique, it makes it possible to run several isolated Linux containers on a single LXC host. It does not make use of a virtual machine, but rather it creates a virtual environment which has its own CPU, memory, blocking I/O, network and other types of resources [13]. LXC offers a stable and complete API, along with other tools that let users create, destroy and manage containers in an easy way.

LXC can be seen as an interface for the underlying Linux kernel containment features, like namespaces and control groups. Namespaces are used to give applications an isolated view of the operating environment and its resources, that include interprocess communication, mount points, process identifiers, networking and user ids. Control groups, on the other side, allow limitation and prioritization of resources, including CPU, memory, blocking I/O, and networking. More specifically, newest versions of LXC make extensive use of many Linux kernel features, including not only namespaces and control groups but also capabilities, AppArmor and SELinux profiles, Seccomp policies and `pivot_root` [15].

Similar in usage and behavior but not equal to the `chroot(2)` system call, `pivot_root(2)` changes the root filesystem of the calling process to the directory pointed to by the first argument while moving the old one to another directory (specified as the second argument). However, as of Linux kernel version 4.8, `pivot_root(2)` changes root and current working directory of each process or thread to the new root directory if they point to the old root directory. While this behavior is debatable, it is currently necessary in order to prevent kernel threads from keeping the old root directory busy with their root and current working directory [16]. The `pivot_root(8)` command simply calls the `pivot_root(2)` system call. All the other main containment features used by LXC were already introduced in the previous sections, so they are taken for granted.

An LXC container is something more than a simple *chroot* but, at the

same time, it is not a virtual machine and thus does not provide the same high level of isolation. LXC containers, in fact, are only meant to create an environment as close as possible to a standard Linux installation, but using the host's kernel instead of running its own [15].

Recent versions of LXC, like version 2.0.6 (released November 23, 2016), consist of various components: the *liblxc* library, a stable public *C* API (see `lxccontainer.h`) along with bindings for many languages, including *Python 3*, *Lua*, *Go*, *Ruby*, *Python 2*, *Haskell* [15]. Moreover, LXC provides a set of standard tools, to control the containers, and distribution container templates, to streamline their creation. In fact, LXC containers are primarily configured via templates and command line utilities. Templates are usually shell scripts, that either build or download root filesystems. LXC comes with a special *download* template, which downloads pre-built container images from a central, trusted, LXC server. Finally, thanks to the integration with init systems like *systemd*, many Linux distributions can automatically start LXC containers upon system boot.

Installation

As with previous examples, we will use Ubuntu 16.10 64-bit as the operating system of choice. On this OS, the `lxc` package – along with all the required and recommended dependencies – can be installed by simply typing `sudo apt-get update && sudo apt-get -y install lxc` in a terminal. As part of the installation process, a network bridge is automatically set up for containers to use. After the installation is done, a system reboot is recommended.

NOTE: By default, containers are located under `/var/lib/lxc` for the root user, and `$HOME/.local/share/lxc` for the other users.

Networking

LXC creates a private network namespace for each container. Containers may connect to the outside world by either having a physical network interface controller (NIC) or a virtual Ethernet (veth) tunnel endpoint

passed into the container. To this end, at host startup, LXC creates a bridge under NAT, called `lxcbr0`. Containers using the default configuration have, in fact, only one veth NIC with the remote end plugged into the `lxcbr0` bridge [15].

Privileged containers

Privilged containers are created, managed and destroyed, by running the LXC commands as the root user. Configuration files for this kind of container are found under `/etc/lxc`. One configuration file (`lxc.conf`) is used to customize several LXC settings, while another one (named `default.conf`) specifies the default configuration to be used by every newly created created container. The latter usually contains at least a network section, like the following:

```
fantox@ubuntu-vm:~$ cat /etc/lxc/default.conf
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
```

Creation of a privileged container

In this example we tell LXC to download (`-t download`) a pre-built image and create a new container named (`-n` option) `priv_cont_01`. The image we want is specified with three template options (everything after `--`). In this case the distribution is Ubuntu (`-d ubuntu`), the release is 16.10 codenamed *Yakkety Yak* (`-r yakkety`), and we choose the 64-bit architecture (`-a amd64`).

```
fantox@ubuntu-vm:~$ sudo lxc-create -t download -n priv_cont_01 \
> -- -d ubuntu -r yakkety -a amd64
Setting up the GPG keyring
Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Ubuntu container (release=yakkety, arch=amd64, variant=default)

To enable sshd, run: apt-get install openssh-server

For security reason, container images ship without user accounts
and without a root password.
```

```
Use lxc-attach or chroot directly into the rootfs to set a root password
or create user accounts.
```

Now that the container has been created, we can see it listed with the `lxc-ls` and use `lxc-info` to obtain detailed container information. The `lxc-start` command is used to start a container (the `-d` option makes the container run as a daemon). Once a container is started, we can create a running process inside it with `lxc-attach`. Finally, we can stop and destroy a container with `lxc-stop` and `lxc-destroy`, respectively:

```
fantox@ubuntu-vm:~$ sudo lxc-ls
priv_cont_01
fantox@ubuntu-vm:~$ sudo lxc-start -n priv_cont_01 -d
fantox@ubuntu-vm:~$ sudo lxc-info -n priv_cont_01
Name:           priv_cont_01
State:          RUNNING
PID:            11647
IP:             10.0.3.67
CPU use:        0.35 seconds
BlkIO use:      1.62 MiB
Memory use:     16.16 MiB
KMem use:       3.46 MiB
Link:           vethNMTB34
 TX bytes:      1.34 KiB
 RX bytes:      8.12 KiB
 Total bytes:   9.47 KiB
fantox@ubuntu-vm:~$ sudo lxc-attach -n priv_cont_01
root@priv_cont_01:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.10
Release:        16.10
Codename:       yakkety
root@priv_cont_01:/# exit
exit
fantox@ubuntu-vm:~$ sudo lxc-stop -n priv_cont_01
fantox@ubuntu-vm:~$ sudo lxc-destroy -n priv_cont_01
Destroyed container priv_cont_01
```

Unprivileged containers

Unprivileged containers are the safest container type LXC can provide. They are run under regular users on the host operating system and cannot access the hardware directly.

Unprivileged containers are underpinned by user namespaces that, because of their hierarchical structure, let privileged tasks in a parent namespace map its ids into child namespaces. By default every task on

the host runs in the initial user namespace, where the full range of ids is mapped onto the full range.

In recent versions of Ubuntu, everytime a new user is created, it is assigned by default a range of UIDs and GIDs, as specified in the `/etc/subuid` and `/etc/subgid` configuration files. Every UID (including UID 0, that is root) in the container is thus mapped to a non-zero UID on the host. This is done to prevent an attacker from gaining root privileges on the host OS in case it manages to escape the container.

As a side effect, many operations that require root privileges are not allowed. For instance, unprivileged containers are unable to create device nodes or mount block-backed filesystems, and to any operation against a UID/GID outside of the mapped set, in general.

NOTE: Unprivileged containers have the drawback of not working with most distribution templates out of the box. To overcome this limitation, pre-built images of the most common distributions that are known to work in such an environment are provided as separate downloads.

Creation of an unprivileged container as a user

In order to create an unprivileged container as a non-root user, we need to have a UID and a GID map. These are located under `/etc/subuid` and `/etc/subgid`, respectively.

The Ubuntu 16.10 installation we will use in the following examples allocates by default 65536 UIDs and 65536 GIDs to every user on the system and starts them at id 100000 to avoid conflicting with system users/groups, so the the user and group ids ranges are 100000–165536:

```
fantox@ubuntu-vm:~$ cat /etc/subuid
fantox:100000:65536
fantox@ubuntu-vm:~$ cat /etc/subgid
fantox:100000:65536
```

Then, we need to edit the `/etc/lxc/lxc-usernet` file, that specifies how unprivileged users may connect their containers to the host-owned network, because users aren't allowed to create any network device on the host by default:

```
fantox@ubuntu-vm:~$ sudo mkdir -p /etc/lxc
fantox@ubuntu-vm:~$ echo "$USER veth lxcbr0 10" | sudo tee -a /etc/lxc/lxc-usernet
fantox veth lxcbr0 10
```

In this way, the user *fantox* is allowed to create up to 10 virtual Ethernet (veth) devices connected to the *lxcbr0* bridge.

We are now ready to define the configuration file for the unprivileged container we will create in the next step:

```
fantox@ubuntu-vm:~$ mkdir -p ~/.config/lxc
fantox@ubuntu-vm:~$ echo "lxc.network.type = veth" >> ~/.config/lxc/default.conf
fantox@ubuntu-vm:~$ echo "lxc.network.link = lxcbr0" >> ~/.config/lxc/default.conf
fantox@ubuntu-vm:~$ echo "lxc.id_map = u 0 100000 65536" >> ~/.config/lxc/default.conf
fantox@ubuntu-vm:~$ echo "lxc.id_map = g 0 100000 65536" >> ~/.config/lxc/default.conf
```

NOTE: The two values of `lxc.id_map` for the user (`u`) and for the group (`g`) should match those found in `/etc/subuid` and `/etc/subgid` to avoid problems.

Finally, we are able to create and administer a new unprivileged container. To do so, we can use the exact same commands we used in the privileged container example, but this time we must not run the commands with `sudo`.

```
fantox@ubuntu-vm:~$ lxc-create -t download -n unpriv_cont_01 \
> -- -d ubuntu -r yakkety -a amd64
Setting up the GPG keyring
Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Ubuntu container (release=yakkety, arch=amd64, variant=default)

To enable sshd, run: apt-get install openssh-server

For security reason, container images ship without user accounts
and without a root password.

Use lxc-attach or chroot directly into the rootfs to set a root password
or create user accounts.
```

Security

The level of security of LXC containers mainly depends on their configurations; capability sets and bridge networking are, in fact, the main attack surfaces. Weak configurations in part are the consequence of the fact that LXC is commonly used to create system containers, that

provide an environment as close as possible to a minimal, isolated, operating system installation. Default configurations can thus be seen as too permissive when creating a container aimed at hosting a single application, like a web server.

Before Linux kernel version 3.8 was released, the root user of the guest operating system (i.e., inside the container) had access to the host operating system and could run arbitrary code on it with root privileges. This changed starting with Linux kernel 3.8, where unprivileged processes can create user and other types of namespaces with just the `CAP_SYS_ADMIN` capability in the caller's user namespace. Moreover, because when a non-user namespace is created it is owned by the user namespace in which the creating process was a member at the time of the creation of the namespace, actions on it require capabilities in the corresponding user namespace [16]. As a result, from LXC 1.0 onward, it is possible to run containers as regular users on the host operating system using unprivileged containers.

LXC ships with a default AppArmor profile that restricts access to the host filesystem, to prevent the host from being attacked from inside the container. For instance, the `usr.bin.lxc-start` profile is entered by running `lxc-start`. The aim of this profile is to prevent `lxc-start` from mounting new filesystems outside of the container's root filesystem. Moreover, before executing the container's initialization, LXC requests a switch to the container's profile. In absence of a custom one, LXC uses the default `lxc-container-default` policy, that can be found under `/etc/apparmor.d/lxc/lxc-default`. The default profile basically prevents the container from accessing many dangerous paths on the host, and from mounting most filesystems [15].

2.2.2 Docker

Docker is an open-source project to pack, ship and run any application as a lightweight container [9]. Originally developed by *dotCloud*, a Platform-as-a-Service (PaaS) company, Docker was later released as an open-source project in March 2013.

In 2013, Docker started as an operating-system-level virtualization technology. Like LXC, it creates containers, each having a virtual environment with its own CPU, memory, blocking I/O, network and other types of resources, instead of virtual machines. In subsequent years the Docker project has grown in size and today it also includes many services, like Docker Hub.

Despite having similar features, Docker differs from LXC in that it promotes the idea of having a single application per container. The philosophy behind Docker, in fact, is to ease the deployment of applications in cloud scenarios. To this end, Docker container images usually pack a filesystem that contains a certain piece of software along with every other thing it needs to run, like runtime, tools and libraries. The kind of containers that share the same philosophy as Docker's are often referred to as *application containers* while traditional containers à la LXC are named *system containers*.

Docker containers are both hardware-agnostic and platform-agnostic [9] and thus are meant to always run the same on every configuration, from laptops to cloud instances; this is the key feature that made Docker popular in PaaS. Docker images, in fact, can be seen as building blocks for deploying and scaling web apps, databases, and backend services without depending on a particular stack or provider [9]. Moreover, Docker images usually ship with common configurations that are meant to ease the work of developers. Despite being mostly used in servers, Docker can run on desktop operating systems as well.

The method Docker uses to sandbox applications is called *containerization*. To achieve isolation, Docker makes use of many Linux kernel containment features we already discussed in the previous sections, like namespaces and control groups. Containerization also solves the problem of dependency management because every container can pack its own dependencies without sharing them with any other application running inside another container: this avoids problems regarding both conflicting and custom dependencies. Even if containers are designed to be isolated and secure, they can be given access to resources belonging to other containers, whether they be local or remote.

Docker used to access Linux kernel's containment features through a LXC driver. This changed in March 2014 with the release of Docker 0.9, that introduced a new library made by the developers of Docker, called `libcontainer`. The new library, written in *Go* language, was specifically designed to access the kernel's container APIs directly [8], without requiring any other dependency. Even if *libcontainer* has been the default execution driver since its introduction, the LXC driver is still available along with other drivers like `systemd-nspawn`, `libvirt`, `qemu/kvm` and many others (see Figure 2.1).

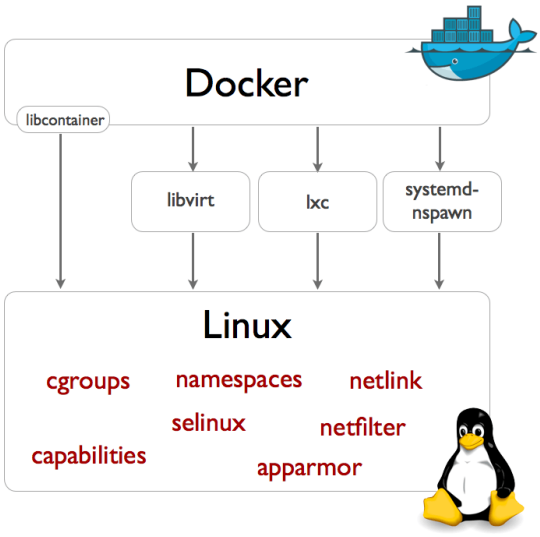


Figure 2.1: Docker Execution Drivers [8].

The `libcontainer` library makes system calls to directly create namespaces and control groups, manage capabilities, AppArmor profiles, network interfaces and firewalling rules [5] on behalf of the Docker client, without depending on user space packages like LXC.

Architecture

Docker uses a typical client-server architecture, made of a *Docker client* (i.e., the `docker` binary) that interacts with the *Docker daemon*, i.e. the server, that in turn creates, builds, manages and monitors images, containers, networks, and data volumes (see Figure 2.2).

NOTE: As the Docker daemon can only run as root, it is recommended to host it on a dedicated machine or an isolated environment, like a virtual machine.

Even if client and daemon can be installed on the same system, they are designed to run on different machines; in the latter configuration, the daemon is said to be remote. In fact, the Docker client uses a REST API – typically over a Unix socket – to interact with the Docker daemon through scripting or direct CLI commands [8]. Docker client, daemon, and REST API are the major components of what altogether is called *Docker Engine*.

NOTE: A Docker client is not coupled with a single local or remote daemon, it can interact with multiple unrelated daemons instead.

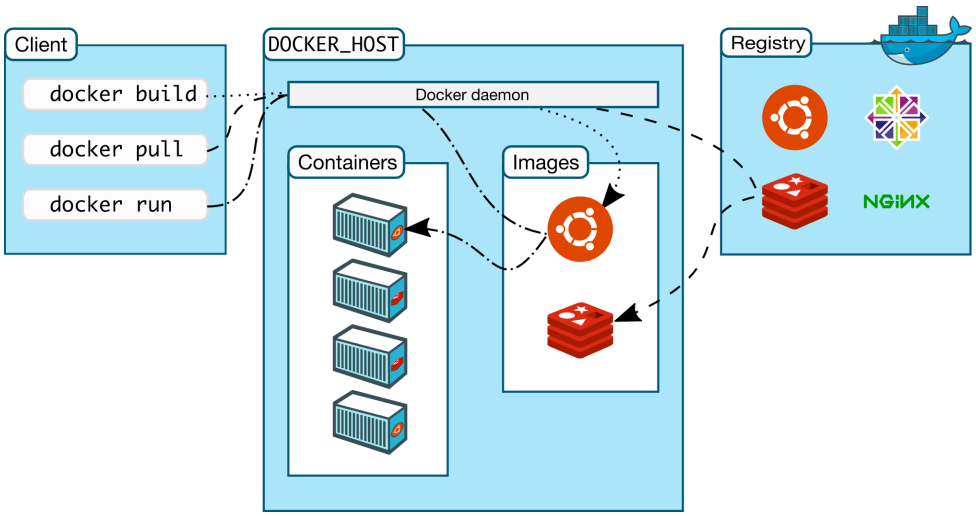


Figure 2.2: Docker Architecture [8].

As shown in Figure 2.2, Docker internals include not only containers but also images and a registries. An image is a template that states, by means of instructions, how a container is created, i.e. the Linux distribution to use and the applications to install. Containers are thus instances of images, that can be hand written from scratch or downloaded from the net, reusing the work of other developers. Moreover, images support multiple inheritance, that essentially allows for the creation of images starting from existing ones. A Docker registry is a (public or private)

collection of images, instead. It can be hosted on a dedicated server or share the system with the client or the daemon. Docker Hub is the most popular public registry where official, signed, Docker images are stored.

Docker images

As previously anticipated, a Docker image is template from which a container can be built. An image is usually made up of many layers, combined together to form a single image thanks to a union filesystem[11], that is capable of overlaying different filesystems – also known as branches – into a single coherent one, by merging their respective directory structures. The single branches are given a priority, to solve potential conflicts while performing the merge operation. Moreover, Docker handles changes to images in a smart way: it determines which layers need to be updated at runtime and therefore it replaces only the layers that are affected by the changes while preserving the others, in order to keep the size of images as small as possible, even after many edits.

A Docker image is defined in a *Dockerfile*, that is basically a text file with a special syntax. An image is created starting from a base image, that defines which Linux distribution to use, like Debian, Ubuntu or Fedora. However, since images support inheritance, one can choose any existing image as the base image for the creation of a new one.

Dockerfile

A Dockerfile is made of a sequence of instructions, that enable the developer to specify the base image (`FROM`), run a command (`RUN`), add a file or directory (`ADD`), create an environment variable (`ENV`), and what process to run when launching a container from the image (`CMD`). A complete list of all the available instructions and their respective usage can be found in the official documentation [8].

[11] Docker can use multiple union filesystem (UnionFS) variants, including AUFS, btrfs, vfs, and DeviceMapper [8].

Whenever one asks Docker to build an image, it executes all the instructions in the Dockerfile one after the other, creating a new layer after each instruction is executed and finally returning the new image. Because images are read-only, whenever a container is run Docker adds a read-write layer (thanks to the union filesystem) on top of the image to store the changes that originate from the execution of the application inside the container. When stopping a container, the new state can be saved as an updated image by simply merging the newly created read-write layer with the others used to build the container.

Example

On Ubuntu 64-bit installs (16.10, in our test machine), we need to enter the following commands to install Docker and other packages that are recommended by the official Docker documentation [8]:

```
sudo apt-get update
sudo apt-get -y install curl linux-image-extra-$(uname -r) linux-image-extra-virtual
sudo apt-get -y install apt-transport-https ca-certificates
curl -fsSL https://yum.dockerproject.org/gpg | sudo apt-key add -
sudo add-apt-repository "deb https://apt.dockerproject.org/repo/ ubuntu-$(lsb_release -cs) main"
sudo apt-get update
sudo apt-get -y install docker-engine
```

In the following example, we will write a Dockerfile that specifies how a custom nginx Docker image is built. First of all, we need to create a new directory to host the Dockerfile. Then, we can write the Dockerfile (its content can be found below as the output of the `cat` command):

```
fantox@ubuntu-vm:~$ mkdir docker_example_01
fantox@ubuntu-vm:~$ cd docker_example_01
fantox@ubuntu-vm:~/docker_example_01$ nano Dockerfile
fantox@ubuntu-vm:~/docker_example_01$ cat Dockerfile
FROM ubuntu:xenial

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 \
                --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62 \
  && echo "deb http://nginx.org/packages/mainline/ubuntu/ xenial nginx">>/etc/apt/sources.list \
  && apt-get update \
  && apt-get -y install nginx \
  && rm -rf /var/lib/apt/lists/*

EXPOSE 80 443

CMD ["nginx","-g","daemon off;"]
```

NOTE: Because Docker creates a union filesystem layer for each instruction inside the Dockerfile, it is recommended to concatenate commands under the same `RUN` instruction whenever they constitute intermediate steps to achieve a particular goal. For instance, all the steps

related to the installation of nginx – adding the PGP signing key and the repository, re-synchronizing the package index files from their sources and, finally, installing the package – can be seen as a single action: install nginx.

The `EXPOSE` instruction indicates the ports on which a container will listen for connections, while the last line of the Dockerfile (i.e., the `CMD` instruction) is the command that will be run as soon as the container is started. In this case, we simply start the nginx daemon to let it accept incoming connections.

We are ready to build the image and start a new container:

```
fantox@ubuntu-vm:~/docker_example_01$ sudo docker build -t docker_example_01 . > /dev/null
fantox@ubuntu-vm:~/docker_example_01$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker_example_01   latest              ac70a2f10863        13 seconds ago      137 MB
ubuntu              xenial              f49eec89601e        10 days ago         129 MB
fantox@ubuntu-vm:~/docker_example_01$ sudo docker run -d -p 80:80 docker_example_01
5a43c78d81fc5054001e0e2fd614213eda754ddf1ee6c718eca9af94b288cb61
```

The `docker build` command builds an image from a Dockerfile and a *context*, i.e. a directory on the local filesystem or a Git repository. In this case, the context is the current working directory (hence the `.` as the last argument) and the Dockerfile is automatically recognised because it is located inside it. To ease the identification of the newly created container, we tag it (`-t` option) with a custom label.

NOTE: Because the build process is run by the Docker daemon, that can also be remote, and because the entire context is always sent to the daemon prior to the build of an image, it is recommended to keep the context as small as possible. A good practice is to start with a directory that contains the Dockerfile and then add only the files needed for building the image, e.g. those that must be copied (with the `COPY` command) to the container's filesystem.

Finally, the `docker run` command starts the container in detached mode (`-d` option) with the container's exposed `80` port published (`-p` option) to the host's `80` port.

Now that the container is running, we can connect to the `80` port. In fact, opening a web browser and visiting `http://localhost:80`, would lead to the typical *Welcome to nginx!* page.

To see what containers are currently being run we can use the

`docker ps` command, while to stop a container whose id is known, there is the dedicated `docker stop` command:

```
fantox@ubuntu-vm:~/docker_example_01$ sudo docker ps --format "{{.ID}}\t{{.Image}}\t{{.Ports}}"
5a43c78d81fc    docker_example_01       0.0.0.0:80->80/tcp, 443/tcp
fantox@ubuntu-vm:~/docker_example_01$ sudo docker stop 5a43c78d81fc
5a43c78d81fc
```

Security

Similarly to LXC, the Docker daemon relies on particular kernel features to provide isolation of processes at the userspace level, like namespaces, control groups and capabilities [8]. Namespaces split the view that processes have of the system, while control groups restrict the resource usage of a process or group of processes [5]. Namespace isolation and capabilities are enabled by default. In particular, before starting a container, Docker creates a set of namespaces for it, that include the `pid`, `net`, `ipc`, `mnt`, `uts` namespaces, introduced in the previous sections. Resource limitations provided by control groups are not used by default, however, but can optionally be enabled on a per-container basis [5]. Custom control groups can be specified in the `docker run` command with the `--cgroup-parent` option [8].

By default, Docker containers are relatively secure because they run unprivileged are not allowed to access any devices. To run a privileged container – and thus give it full access to all devices – we need to execute `docker run` with the `--privileged` flag set [8]. Moreover, before running a privileged container, Docker automatically configures AppArmor or SELinux policies to give it almost the same level of access to the host as processes running directly on the host itself. Furthermore, when launching a privileged container, it is possible to add and drop individual capabilities with the `--cap-add` and `--cap-drop` options, respectively. Among the default capabilities that are given to a privileged container we find `MKNOD`, `CHOWN` and `SYS_CHROOT`, to name a few. A complete list of all the capabilities that can be added or dropped can be found on the official site.

Docker containers are not bullet proof. To further constrain their access to the host system or other containers and reduce the risk of

container escape, it is recommended to configure the host with strict AppArmor/SELinux profiles and Seccomp policies (i.e., host hardening). Default configurations, in fact, protect the host from containers but not containers from other containers [5]. Docker's philosophy of application containers makes them relatively secure because in many cases containers can be run unprivileged or with little to no capabilities. Security concerns arise when developers or system administrators try to use containers as a replacement to virtual machines, e.g. by packing inside a container far more than a single application and adding to the container capabilities that are dropped by default. As with LXC, improper use of containers and overly permissive configurations can have severe adverse effects on the host system. For instance, with `--cap-add=SYS_ADMIN`, a container can remount `/proc` and `/sys` subdirectories in read/write mode and change the host's kernel parameters [5].

In configurations where there is one or more remote daemons, an attacker can try to sniff the traffic between the client and the daemons and then carry out an attack. In case it manages to modify the communications, it is potentially given complete access to the daemons, that are run as root on their respective hosts. Once an attacker is given access to the daemons, it can potentially start a Denial-of-Service (DoS) attack.

Last but no least, there are security concerns regarding Docker images hosted in third party registries. When we download an existing image from a registry, Docker Hub included, we trust that the image is free from malicious code. Since image creation is mostly automated, we cannot be sure the images are safe to use. In fact, images usually include code coming not only from trusted sources, but also from third-party GitHub repositories of untrusted developers. This, however, is a common problem that affects package managers as well.

Docker containers vs VMs

Before the container technology emerged, the preferred method for distributing and isolating applications was to make use of virtual machines

(VMs). This made perfectly sense at the time because applications were effectively isolated and came with all they needed to run out of the box. However, there were limiting factors that often prevented VMs to be used in certain contexts. Firstly, VMs are usually quite big in size, so their transfer through the net is inconvenient, especially in case of large-scale deployments. Docker containers, on the other side, are typically small, from dozens to a few hundreds of megabytes. For instance, the latest official Ubuntu 16.04 (codenamed *xenial*) image is only 129 MB in size. Moreover, VMs' higher level of isolation comes at the cost of resources (i.e., CPU and memory) and performance [9], while Docker containers have almost no impact in terms of memory usage and CPU overhead. Finally, VMs are usually less developer-friendly because they pack an entire system, while Docker containers focus on the application and the tools needed by developers to get their work done, while hiding everything else. This is more a practical difference than a technological one: application containers are meant to be completely portable applications to be used by developers but the same philosophy could, in theory, be applied to VMs as well.

Chapter 3

Benchmarks

The purpose of this chapter is to carry out a quantitative analysis of the overhead introduced by the previously presented container technologies, i.e. LXC and Docker, while accomplishing ordinary tasks, like compressing/encrypting/transferring a file, compiling a piece of software from its source and serving a static website.

Similarly to the previously examined examples, all the following tests were run inside a custom virtual machine. In this way, it was possible to define a minimal environment inside which execute the benchmarks, with all the benefits of using a virtual machine, especially the ability to save its state and restore it later. The host machine on which the VMware Fusion type-2 hypervisor, that powered the virtual machine used in the following tests, was run is a MacBook Pro (Retina, 15-inch, Late 2013), with the following main technical specifications:

Processor	2.3GHz quad-core Intel Core i7 (Turbo Boost up to 3.5GHz) with 6MB shared L3 cache
Memory	16GB 1600MHz DDR3
Storage	512GB PCIe-based onboard SSD
Graphics	Intel Iris Pro 1536MB NVIDIA GeForce GT 750M 2GB GDDR5
Networking	10/100/1000BASE-T Thunderbolt Ethernet
OS	macOS Sierra version 10.12.3
Type-2 hypervisor	VMware Fusion Professional version 8.5.3

The following are the specifications of the custom virtual machine:

Processor	1 processor core
Memory	8GB
Storage	40GB virtual disk (vmdk)
Graphics	2GB shared memory
Networking	Bridged
OS	Ubuntu 16.10 64-bit Linux 4.8.0-37-generic (distribution kernel)
Packages	apt-transport-https (1.3.4) ca-certificates (20160104ubuntu1) curl (7.50.1-1ubuntu1.1) docker-engine (1.13.0-0~ubuntu-yakkety) linux-image-extra-4.8.0-37-generic (4.8.0-37.39) linux-image-extra-virtual (4.8.0.37.46) lxc (2.0.6-0ubuntu1~ubuntu16.10.2) open-vm-tools-desktop (2:10.0.7-3227872-5ubuntu1)

NOTE: The virtual machine was assigned only one processor core on purpose, as an easy way to keep variance of measurements small.

First, all the packages that ship with Ubuntu were updated to their latest versions available in the official repositories. Then, to provide enhanced functionality to VMware Fusion, the `open-vm-tools-desktop` package was added. After that, LXC and Docker together with their recommended packages (see the table above for the complete list) were installed. LXC and Docker versions are the 2.0.6, released November 23, 2016, and the 1.13.0, released January 18, 2017, respectively. They were the latest stable releases available at the time the virtual machine was set up. Finally, the virtual machine was compressed in a `ZIP` file.

Before each of the tests that will be explained in the following sections, a clean instance of the virtual machine was obtained by uncompressing the archive file while the old instance, used for the previous test, was trashed. This was done to ensure that tests are independent of one another, in the sense that previous tests do not affect the performance of subsequent ones.

During each test, LXC and Docker were used to create unprivileged containers running the latest Ubuntu 16.10 64-bit images available as base images, to reflect the operating system installed in the virtual machine. In this way, the version of a package installed on the three environments (the VM, the LXC container, and the Docker container) was exactly the same. Differences in total execution time of tests in the three distinct environments are thus not to be attributed to the versions of the installed packages.

For the sake of homogeneity, all the following tests were iterated ten times in each of the three environments (i.e., VM, LXC container, and Docker container). This allowed for a final side-by-side comparison of the average performance loss introduced by container technologies in the various tests. The performance loss was computed considering the average total execution time of the ten iterations in the virtual machine environment as the reference value. In each test, the `time` user command was employed to measure the elapsed time between invocation and termination of the command used to launch the specific benchmark. Those familiar with the `time` command should know that it returns three measures (*real*, *user*, and *sys*), however the only measure that was considered in the tests, and used in both tables and charts, was *real*.

3.1 Test 1: Compressing a file

The purpose of this test was to measure the overhead introduced by containers while performing the typical CPU and I/O intensive task of compressing a file. The size of the binary file used in this test was arbitrarily chosen to be exactly 256MiB[1]. The terminal command used to accomplish the task was the following:

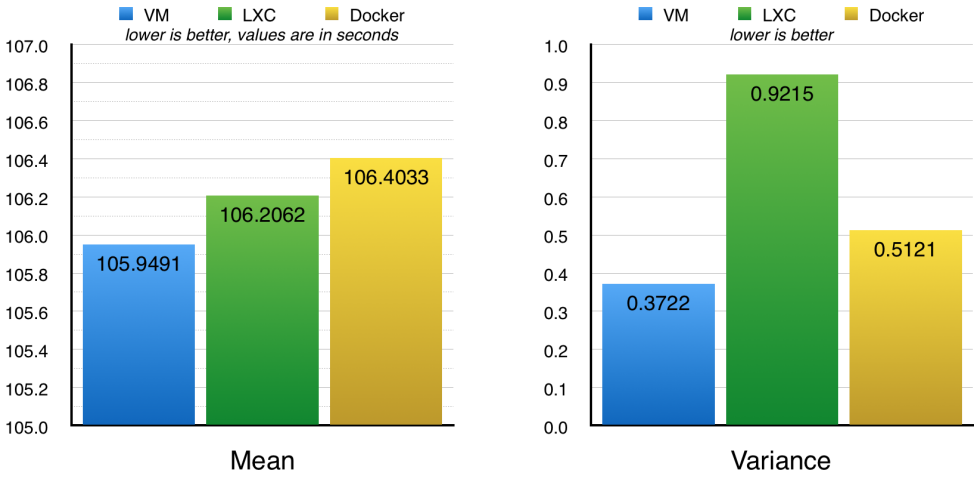
```
time tar -cJf ARCHIVE_NAME.tar.xz FILE_NAME
```

The following table contains the acquired data together with computed sample mean, unbiased sample variance, and performance loss indicator for each set of iterations:

[1] One mebibyte (MiB) is equal to 2^{20} bytes.

	VM	LXC	Docker
Iteration 1	1m45.686s	1m47.558s	1m47.409s
Iteration 2	1m46.416s	1m45.204s	1m45.831s
Iteration 3	1m46.769s	1m46.148s	1m46.054s
Iteration 4	1m45.215s	1m45.097s	1m46.072s
Iteration 5	1m45.464s	1m46.741s	1m45.502s
Iteration 6	1m44.874s	1m45.635s	1m47.260s
Iteration 7	1m46.177s	1m48.019s	1m46.588s
Iteration 8	1m46.366s	1m45.866s	1m45.743s
Iteration 9	1m46.401s	1m46.003s	1m46.184s
Iteration 10	1m46.123s	1m45.791s	1m47.390s
Mean	1m45.9491s	1m46.2062s	1m46.4033s
Variance	0.3722	0.9215	0.5121
Performance loss	//	0.243%	0.429%

The following charts show side-by-side comparisons of mean and variance, with minimum and maximum measured values for each column colored in green and red, respectively, to ease the reading of results:



NOTE: For convenience, all the time values in charts were converted to seconds. Moreover, to improve readability, the Y axis scale of charts was adjusted to highlight even the smaller differences in values.

The very low variance values prove that samples are stable and the acquisition method is quite accurate. In this test, LXC and Docker

performed really well, introducing a 0.243% and a 0.429% overhead, respectively, a practically negligible performance loss.

3.2 Test 2: Encrypting a file

Today, file encryption is more popular than ever. This test measured the performance impact of container technologies while encrypting a binary file of exactly 4GiB[2] using the Advanced Encryption Standard (AES) block cipher (128-bit block size) with 256 bit key length and using the Cipher Block Chaining (CBC) mode of encryption. The terminal command used to accomplish the task was the following:

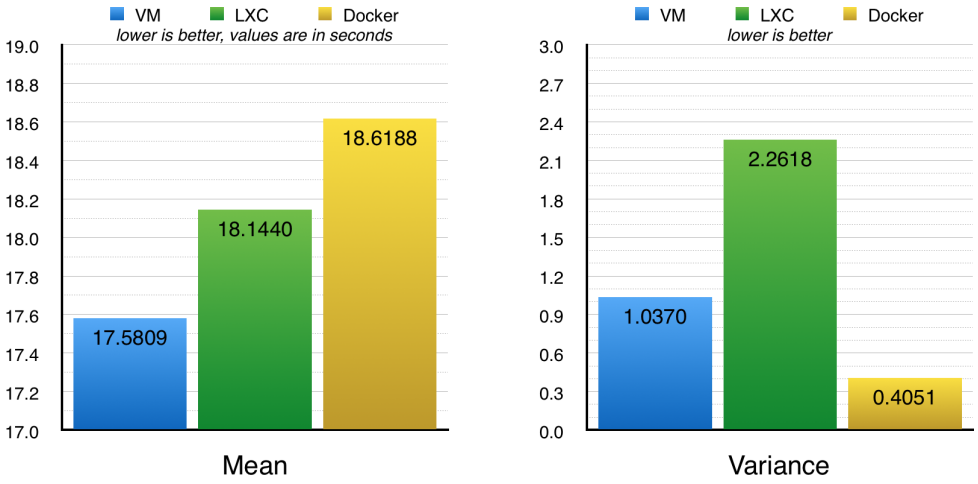
```
time openssl enc -e -aes-256-cbc -in FILE_NAME -out ENC_FILE_NAME.enc \
                -pass pass:STRONG_PASSWORD
```

As in the previous test, the acquired data together with computed sample mean, unbiased sample variance, and performance loss indicator for each set of iterations are shown in a table:

	VM	LXC	Docker
Iteration 1	17.517s	19.557s	19.364s
Iteration 2	18.242s	20.711s	19.315s
Iteration 3	17.252s	18.078s	18.997s
Iteration 4	17.015s	17.926s	17.508s
Iteration 5	16.608s	16.977s	17.960s
Iteration 6	17.968s	15.591s	18.712s
Iteration 7	16.944s	18.263s	18.624s
Iteration 8	16.235s	18.145s	17.895s
Iteration 9	18.292s	19.478s	19.127s
Iteration 10	19.736s	16.714s	18.686s
Mean	17.5809s	18.144s	18.6188s
Variance	1.037	2.2618	0.4051
Performance loss	//	3.203%	5.904%

Mean and variance, graphically:

[2] One gibibyte (GiB) is equal to 2^{30} bytes.



This test shows that containers suffer a slight performance loss while encrypting files with the OpenSSL library, with Docker performing worse than LXC. Docker, in fact, showed a nearly 6% performance overhead, almost twice LXC's.

3.3 Test 3: Compiling the Linux kernel from source

The aim of this test was to measure the overhead introduced by container technologies while doing the CPU intensive task of compiling a big piece of software, like the Linux kernel. The latest stable version available at the time of this test was the 4.9.8, released February 4, 2017. The base configuration file used to build the kernel was generated, in the virtual machine environment, by the `make localmodconfig` command and then manually edited to only include the relevant modules.

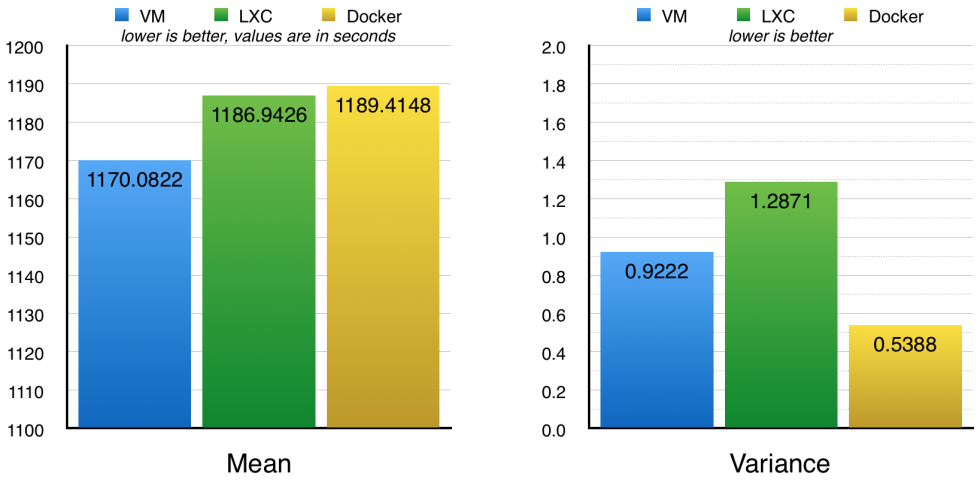
The terminal command used to launch the kernel build task was:

```
time make -s
```

The results can be found in the table below, that uses exactly the same conventions adopted in the previous tests:

	VM	LXC	Docker
Iteration 1	19m28.641s	19m48.405s	19m48.363s
Iteration 2	19m29.495s	19m46.648s	19m48.257s
Iteration 3	19m29.996s	19m45.837s	19m49.073s
Iteration 4	19m28.860s	19m45.670s	19m49.089s
Iteration 5	19m29.635s	19m49.278s	19m49.865s
Iteration 6	19m30.096s	19m46.518s	19m50.001s
Iteration 7	19m31.441s	19m46.115s	19m50.298s
Iteration 8	19m30.386s	19m47.305s	19m49.261s
Iteration 9	19m31.227s	19m46.686s	19m50.296s
Iteration 10	19m31.045s	19m46.964s	19m49.645s
Mean	19m30.0822s	19m46.9426s	19m49.4148s
Variance	0.9222	1.2871	0.5388
Performance loss	//	1.441%	1.652%

The following are the mean and variance charts, with time values converted in seconds:



Results show that, as expected, kernel build times are a few seconds longer in containers, however the performance loss is very low. LXC and Docker, in fact, performed quite well, with a decrease in performance below 1.7% in both cases.

3.4 Test 4: Serving a static website

One of the most popular use cases of containers is to power an isolated instance of a web server. In this test, the nginx web server was configured to serve a big static website consisting of 16,539 items, totalling nearly 348MiB in size.

The well-known `wget` command was used to retrieve all the items composing the static website in bulk, thanks to the `-m` option that translates to options suitable for mirroring being turned on. To prevent network congestion from affecting the measurements, the machine used to run the following command was the MacBook Pro host:

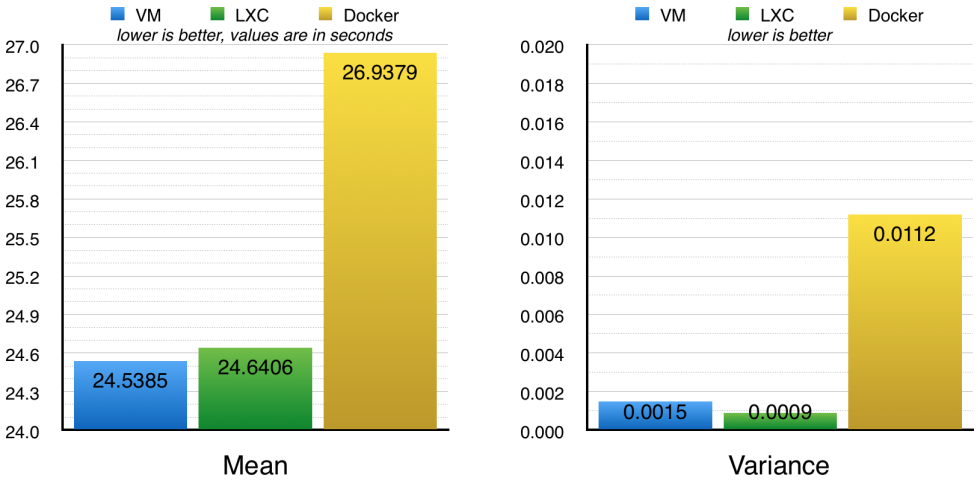
```
time wget -mq http://IP:PORT
```

The acquired measures together with the usual computed statistics, i.e. mean, unbiased sample variance, and performance loss indicator are shown in the table below:

	VM	LXC	Docker
Iteration 1	24.489s	24.654s	26.802s
Iteration 2	24.618s	24.624s	26.877s
Iteration 3	24.514s	24.634s	26.779s
Iteration 4	24.577s	24.700s	26.927s
Iteration 5	24.538s	24.624s	26.993s
Iteration 6	24.505s	24.686s	26.877s
Iteration 7	24.524s	24.617s	27.118s
Iteration 8	24.544s	24.620s	26.977s
Iteration 9	24.512s	24.616s	27.039s
Iteration 10	24.564s	24.631s	26.990s
Mean	24.5385s	24.6406s	26.9379s
Variance	0.0015	0.0009	0.0112
Performance loss	//	0.416%	9.778%

In this test, acquired data show impressively small values of variance, which is an indicator of very accurate measurements. Because values of variance are extremely small, the Y axis scale of the variance chart was

adjusted to only show values in the 0–0.02 range. Mean and variance values are shown in the following charts:



In this case, LXC performed best by introducing almost no overhead at all. Docker, on the other side, performed quite bad, with a performance loss of nearly 10%. This is the test were Docker performed worst. The causes of the high overhead introduced by Docker with respect to LXC are mostly unknown. In fact, because variance values are extremely small, it is unlikely that the causes of the increased total time spent by nginx running inside the Docker container to provide all the items composing the website were related to the method used to perform the measurements.

3.5 Test 5: Securely transferring a file

The purpose of this test was to see if there are differences in the time needed to securely transfer a file to a container with respect to accomplishing the same task directly on the virtual machine. This test is important because the task being run is particularly I/O intensive. To make the total execution time being higher than a few seconds, the size of the file being transferred was set to be 1GiB. Between the many ways to securely transfer a file, the `scp` command was chosen. As in the previous test, the command was run from the MacBook Pro host to

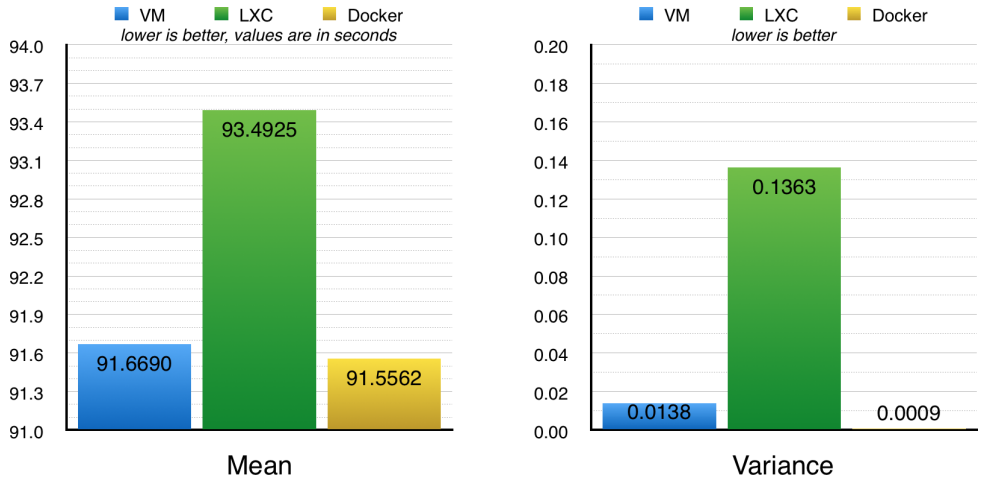
prevent network congestion from affecting the results:

```
time scp SOURCE_FILE_NAME USER@IP:DESTINATION_FILE_NAME
```

Both the acquired data and the usual computed statistics can be found in the table below:

	VM	LXC	Docker
Iteration 1	1m31.634s	1m33.231s	1m31.523s
Iteration 2	1m31.630s	1m33.865s	1m31.527s
Iteration 3	1m31.605s	1m33.411s	1m31.626s
Iteration 4	1m31.995s	1m33.908s	1m31.569s
Iteration 5	1m31.657s	1m33.590s	1m31.560s
Iteration 6	1m31.622s	1m32.965s	1m31.570s
Iteration 7	1m31.629s	1m33.232s	1m31.551s
Iteration 8	1m31.598s	1m34.045s	1m31.546s
Iteration 9	1m31.631s	1m33.596s	1m31.565s
Iteration 10	1m31.689s	1m33.082s	1m31.525s
Mean	1m31.669s	1m33.4925s	1m31.5562s
Variance	0.0138	0.1363	0.0009
Performance loss	//	1.989%	−0.123%

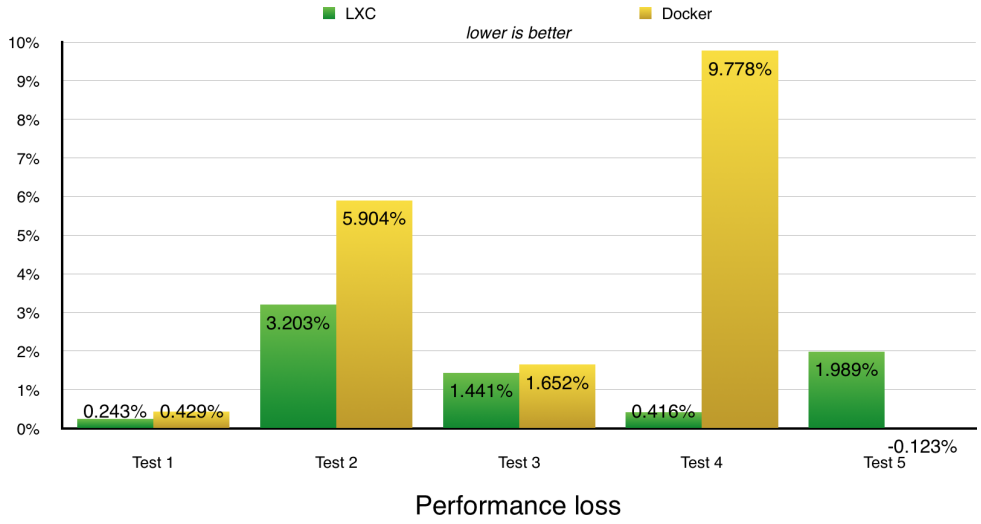
Mean and variance, graphically, with all the time values in charts converted to seconds for convenience:



In this test, LXC performed quite well with a performance loss lower than 2%. Surprisingly, Docker performed slightly better than the reference virtual machine, resulting in a performance *increase* of 0.123%. Theoretically, it should not be possible for a container to outperform the underlying virtual machine, so it is likely that instead of Docker performing better it is the underlying virtual machine that, for some reason, performed worse than it should maybe because of background kernel tasks being run during the test.

3.6 Performance comparisons

In the previous sections, five tests were presented and their result were analyzed individually. To have a better overview, the computed values of overhead introduced by containers with respect to the virtual machine are shown side-by-side in the following chart:



The first three tests show similarities, with containers introducing little to no overhead, especially in Test 1, and LXC always doing slightly better than Docker. Test 4 is an extreme case in which LXC completely outperformed Docker. Test 5 deserves a specific mention in that it is the only test in which a container technology (namely, Docker) did better than the underlying virtual machine. Test 5 it is also the only test in which Docker showed better performance than LXC.

As a final thought, it is worth noting that if we look at the tables representing the data of the five tests, we will notice that minimum and maximum execution times for each column (i.e., those colored in green and red, respectively) are totally independent of the iteration number. Iterations, in fact, were always run in a clean environment; before each iteration, file and folders created in the previous iteration were removed and, in general, the environment was restored as accurately as possible to the initial clean state. Most likely, if the environment had not been cleaned after each iteration, execution time would have progressively increased.

All things considered, container technologies did a good job in all the tests, resulting in a performance loss being always below 10%, and in most cases below 2%. LXC showed more consistent results than Docker, that holds both the best and the worst overall results in terms of performance loss.

Conclusion

The field of virtualization has been an active research area in computer science and engineering for a long time. Thanks to the work of brilliant computer scientist, like Gerald J. Popek and Robert P. Goldberg, hardware virtualization and hypervisors became popular in the 1970s for their ability to better optimize the hardware resources of mainframes. Today, type-2 hypervisors are still being used in computers for many reasons, e.g. to overcome incompatibility issues and to create isolated environments for security-critical applications. Type-1 hypervisors, on the other side, are more commonly employed in data centers, where the huge hardware resources need to be divided into smaller units and isolation is a requirement.

In the late 1980s, a new lightweight virtualization approach, named operating-system-level virtualization, marked the start of a new era. Today, the most popular implementations of this new kind of virtualization are container technologies. Despite being often considered the successors of hypervisors, container technologies are actually not meant to replace them. In fact, the virtual machine abstraction introduced by hypervisors is based on the concept of virtual hardware, on top of which an entire operating system is executed. Containers, on the other side, do not involve virtual hardware and run the same kernel of the underlying host. Moreover, some container technologies, especially Docker, shifted the focus from virtualizing an entire system to isolating a single piece of software. Application containers, in fact, are increasingly growing in popularity, especially among software developers, as a fast and easy way to pack and distribute software.

Virtual machine and containers also provide different levels of isola-

tion, with containers being intrinsically less secure than virtual machines because of their architecture. In the near future, containers will be able to guarantee even higher levels of isolation thanks to the advancements in underlying containment features of the Linux kernel that are being done at each release.

With regards to performance, containers have proven to be really fast. Benchmarks actually showed that both LXC and Docker introduce very low overhead and thus containers are to be considered a good solution when dealing with high performance requirements.

Because of all the aforementioned reasons, the choice of using virtual machines or containers must be made on a case-by-case basis, depending on the specific isolation and performance needs.

The benchmarks introduced in this work were run on a virtual machine equipped with only one processor core. In a future work, it would be interesting to execute the same tests in a virtual machine configured to use more cores and then compare the results.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 2–13, October 2006.

[2] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC '12, pages 373–385, 2012.

[3] *AppArmor wiki*. http://wiki.apparmor.net.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, October 2003.

[5] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, September 2016.

[6] F. J. Corbató, M. M. Daggett, R. C. Daley, R. J. Creasy, J. D. Hellwig, R. H. Orenstein, and L. K. Korn. *The Compatible Time-Sharing System: A Programmer's Guide*. The MIT Press, 1963.

[7] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International*

Journal of Innovative Technology and Exploring Engineering (IJITEE), 2(3):222–225, February 2013.

[8] *Docker*. https://www.docker.com.

[9] *Docker project on GitHub*. https://github.com/docker/docker.

[10] Kazuhiro Fuchi, Hozumi Tanaka, Yuriko Manago, and Toshitsugu Yuba. A program simulator by partial interpretation. In *Proceedings of the Second Symposium on Operating Systems Principles*, SOSP '69, pages 97–104, October 1969.

[11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[12] IEEE and The Open Group. *POSIX.1-2008*. http://pubs.opengroup.org/onlinepubs/9699919799.2016edition, 2016.

[13] Zhanibek Kozhirbayev and Richard O. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182, March 2017.

[14] *KVM*. http://www.linux-kvm.org.

[15] *Linux Containers*. https://linuxcontainers.org.

[16] The Linux man-pages project. *Linux man-pages release 4.09*. https://www.kernel.org/pub/linux/docs/man-pages/man-pages-4.09.tar.xz, December 2016.

[17] *Oracle VM Concepts Guide for Release 3.3*. http://docs.oracle.com/cd/E50245_01/E50249/html/index.html, July 2016.

[18] *Oracle VM VirtualBox*. https://www.virtualbox.org.

[19] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys*, 45(2):17:1–17:39, February 2013.

[20] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[21] *QEMU*. http://qemu.org.

[22] Laurie Robertson. Anecdotes. *IEEE Annals of the History of Computing*, 26(4):71–73, October 2004.

[23] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Proceedings of the 2010 Second International Conference on Computer and Network Technology*, ICCNT '10, pages 222–226, April 2010.

[24] *SELinux wiki*. https://selinuxproject.org.

[25] *SELinux Reference Policy project*. https://github.com/TresysTechnology/refpolicy.

[26] Amit Singh. *An Introduction to Virtualization*. http://www.kernelthread.com/publications/virtualization, January 2004.

[27] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[28] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson, 3rd edition, 2014.

[29] *Xen Project*. https://www.xenproject.org.

Acknowledgements

First and foremost, I would like to thank my supervisor, professor Gabriele D'Angelo, for his precious guidance, useful feedback and endless patience. I would also like to thank my family, especially my parents, for all the support, understanding and love. Last but not least, I thank my closest friends for always being there when I need them, and my fellow students, for the wonderful time we spent together.