USING VM CLONING FOR EFFICIENT, BACKWARD-COMPATIBLE, SECURE CONTAINERIZATION

A Thesis

by

MANVITHA KANMANTHA REDDY

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2020

Major Subject: Computer Science

ABSTRACT

Despite the lack of hardware-enforced isolation, containers have been widely adopted in cloud due to deployability and lightweightness. Secure containers such as gVisor, Firecracker, and Kata-container have addressed the security issue, yet its still missing a solution that is both backward compatible and resource efficient. This thesis proposes a new approach to build secure, lightweight, backward-compatible containers using the Xen hypervisor. Using Copy-on-Write (COW) cloning, a container can be quickly spun up in a virtual machine (VM) that is identical to the container-hosting VM. The containers built this way don't need any underlying kernel modifications or external agents or proxies and are more efficient than spinning up a whole new VM for a container. We successfully demonstrate a prototype which shows the feasibility of such a virtualization-based containerization solution.

DEDICATION

To **M**y **P**arents **D**hana Laxmi, **P**rabhakar Reddy,

Cousin **B**rother **A**shok.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Chia-Che Tsai and committee member Dr. Dilma Da Silva for their continuous guidance and support throughout the course of this research project.

I would also like to to express my deepest gratitude for fellow colleagues, Ranjith Tamil Selvan and Gowtham Srinivasan for their certain aspects of experimentation and code insights.

This section remains incomplete without expressing my sincere thanks to my parents, family and friends for all their love and moral support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Chia-Che Tsai and Dr. Dilma Da Silva of the Department of Computer Science and Dr.I-Hong Hou of the Department of Electrical and Computer Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

The work was funded by the Dean Start-Up Grant 02-241570 from Texas A&M University.

NOMENCLATURE

VM	Virtual Machine
OS	Operating System
VMM	Virtual Machine Monitor
COW	Copy on Write
IT	Information Technology
L0	Level 0
L1	Level 1
KVM	Kernel-based Virtual Machine
Cgroups	Control groups
LXC	Linux Containers
PID	Process Identifier
initRd	initial ramdisk
Dom 0	Domain 0
Dom U	Domain U
HVM	Hardware assisted Virtual Machine
PV	Paravirtualized
EPT	Extended Page Tables
HAP	Hardware Assisted Paging
OCI	Open Container Initiative
Ptrace	Process Trace

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Background

Cloud computing [1] is the delivery of on-demand computing services ranging from applications, storage and processing power on a pay-as-you-go model over the internet. As cloud computing is based on sharing of resources, it makes compute power available at a cheap price and thus achieves economies of scale. Before the introduction of cloud computing, server/client [2] is the most common paradigm for application delivery. Over the years, cloud computing has evolved through various phases from mainframe computing [3], grid computing [4], utility computing, and Software-as-a-Service, etc. Currently, the various types of cloud computing offerings are Infrastructure-as-a-Service (IaaS) [5], Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS). In IaaS, the client is provided with all the hardware and it is up to the client how to use it, whereas in PaaS, the vendor inherently provides platforms used for application development. SaaS lets developers to host and serve their applications on cloud to users.

Virtualization [6], as illustrated in Figure 1.1, is one of the technologies that drives cloud computing and makes it possible to provide isolated environments while sharing resources. There are various kinds of virtualization like Hardware virtualization [7], Network virtualization [8] and Memory virtualization [9] etc. Hardware virtualization or Server virtualization is one of the most widely used virtualization types in cloud computing. This technology makes it possible to run various virtual machines (VM) with the same hardware resources. These VMs are stand-alone servers capable of all kinds of computation. All the VMs running on the same server are isolated and any user-level and kernel-level compromises are restricted in the scope that they can affect the system.

Virtualization software called hypervisor [10] or Virtual Machine Monitor (VMM) does the heavy lifting of abstracting and multiplexing underlying hardware from applications. The physical server hosting the VMs is called the host and the VMs are the guests. Depending on the type of hypervisor, it either sits in between the hardware and the guest Operating System (OS) or in be-

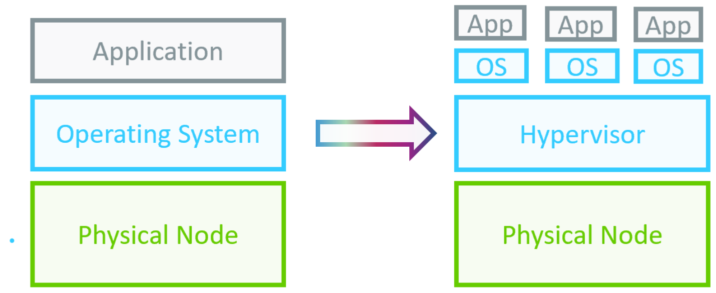


Figure 1.1: Difference between Traditional System and Virtualized System Architectures.

tween host OS and guest OS. VMM handles resource allocation, scheduling, access to the VMs etc. Hyper-V [11], VMware vSphere [12], Xen and KVM [13] are some of the available hypervisors.

Server virtualization includes Full virtualization [14] and Paravirtualization [15]. Full virtualization emulates the entire hardware and thus allows running of unmodified guests. Paravirtualization needs the guest OS to be modified as it does not emulate entire underlying hardware.

The main idea behind virtualization is efficient hardware usage. Most of the times, physical servers only use a small portion of their compute or storage capacity. With virtualization, we mostly overcome this problem by creating multiple VMs out of hardware and every VM is equivalent to a physical server. Each VM has it's own OS and thus all OS overhead and large spin-up times.

Containerization [16], as illustrated in Figure 1.2, is a lightweight alternative to virtualization based on OS-level virtualization. It is the process of encapsulating an application along with its dependencies i.e., libraries, frameworks, and configuration files etc. Linux Containers [17] (LXC) available in linux is OS-level virtualization method that provides the features for building isolated processes i.e., container.

Using LXC features, multiple containers can be spun up on a single system. Containers are lightweight, portable, have fast spin-up times and highly scalable with efficient resource utilization. The cost of spinning up a container is mostly lower than spinning up a VM. Docker, Apache Mesos [18], and CoreOS rkt [19] are some of the currently available container runtimes.

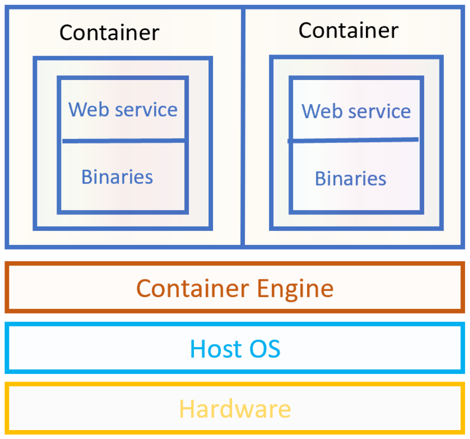


Figure 1.2: An Overview of Containerization.

1.2 Motivation

Containers don't have their independent OS. They share the host OS and thus require all the userspace dependencies to be compatible with host OS. As all the containers running on a server share the host kernel, the host kernel can be compromised if one of the containers exploit any kernel vulnerabilities, raising some grave concerns about security and isolation. They are not as isolated as VMs. Any security vulnerabilities in host kernel can lead to comprising all the co-hosted containers security. Even the security vulnerability in hypervisor can also lead to VM security breaks. However, the privileges vested with hypervisor are way less compared to host OS privileges.

As a result, the demand for sandboxed containers has gained traction and led to the development of various industry solutions. Google gVisor [20] creates a specialized guest kernel for running containers, IBM Nabla builds containers on top of Unikernels [21], Amazon Firecracker [22] is an extremely lightweight hypervisor for sandboxing applications, and Katacontainer [23] places containers in a specialized VM. These listed container runtimes are secure and some of them are as performance and resource efficient as native containers.

Although these solutions satisfy the security constraints, each of these container solutions have their additional hardware requirements, or need external proxies and agents. Moreover, most of them even need modifications to the underlying kernels which leads to loss of backward compatibility. Thus, we currently lack secure, backward compatible and efficient containers which is the main focus of this thesis work.

Running containers in distinct virtual machines can provide great isolation and compatibility but it often requires or may require a larger resource footprint making them resource inefficient. Containers spin up this way also have high start up times and also make container scaling difficult. In this work, we will be exploring various methods of building compatible containers over Xen platform.

This thesis proposes a new approach to build secure, backward-compatible containers using VM cloning techniques over Xen platform. Xen is widely embraced open source hypervisor. VM cloning techniques are mostly used for creating backup snapshots and there are many VM checkpointing techniques that focus on achieving low VM down times. In our work, we mainly focus on Suspend-save-restore and Copy-on-write (COW) checkpointing. Xen currently has support for Suspend-save-restore checkpointing and has features that let one to implement COW checkpointing given the host system has hardware assisted virtualization support enabled.

Using the current existing Xen features like libvchan (for interdomain communication), Suspend-save-restore checkpointing and minimal hypervisor code changes, we build a prototype that achieves backward-compatibility objective. Our prototype shows the feasibility of such a virtualization-based containerization solution and later we proceed to compare the performance against existing secure containers solutions.

In our future work, we plan to implement COW cloning and integrate into our prototype to achieve better performance. Using COW cloning, a container can be quickly spun up in a virtual machine that is identical to the container-hosting VM. The main benefit of this technique over Suspend-save-restore is no new physical frames are allocated for copied VM unless and until both the original and copy domainU (domUs) deviate similar to COW in process forking. This COW cloning results in less memory overhead and also low spins-up time as no physical frame allocation happens and only Extended Page Tables (EPT) tables are copied.

1.3 Contributions

1. A comprehensive study of the existing lightweight secure containerization solutions, including gVisor, AWS Firecracker, and Katacontainer, with performance evaluation of their startup time and system latencies.

2. Based on the case study, we propose a new problem of backward-compatibility among secure containerization solutions, and suggest a new approach of using VM cloning for providing backward-compatible secure containers.

3. Based on our approach, we propose the architecture and the evaluation plan for demonstrating the feasibility of the idea.

1.4 Structure

In Section 2, we will go through detailed discussion on virtualization and containerization. Section 3 covers details of our comprehensive case study. Then in Section 4, we move on to our architecture and implementation details. Section 5 focuses on evaluation and conclusions.

2. BACKGROUND

2.1 Virtualization

Virtualization, in general, is a method of creating a virtual version of physical resources. Virtualization in computing implies logically segregating the physical hardware resources like storage, network or computing power. It is the technology that drives cloud computing by abstracting the physical compute resources from underlying hardware and creates an illusion of existence of the multiple independent physical resources by creating respective multiple virtual resources. One of the main advantages of virtualization is efficient physical hardware resource utilization. We will briefly discuss various virtualization types in below sections.

2.1.1 Network Virtualization

Network virtualization e.g., VL2 [24], VICTOR, SPAIN, etc. abstracts network hardware elements like connections, switches etc. into software running on a hypervisor. This helps in easier network management without actually touching underlying hardware resources. Various types of network virtualization include Software Defined Networking, Network Function Virtualization etc.

2.1.2 Desktop Virtualization

Desktop virtualization lets you run multiple desktop operating systems inside a single system without actually modifying the primary operating system. Virtual Desktop Infrastructure, Local Desktop Virtualization are two different forms of desktop virtualization.

2.1.3 Cloud Virtualization

Cloud virtualization e.g., AWS, OpenStack [25], etc. offers a range of services to customers by virtualizing hardware resources, development tools, cloud based services and software, etc. Various types include IaaS, PaaS and SaaS.

2.1.4 Server Virtualization

Server virtualization is one of the most sought after virtualization forms. Some of the examples of server virtualization are VMware workstation [12], Virtualbox [26], etc. This technology lets you create multiple virtual servers out of a single physical server. Server virtualization uses hypervisor or VMM to abstract hardware elements like processors, memory and storage etc. and creates multiple isolated virtual environments VMs.

A VM is an equivalent of a single stand alone server capable of performing all computation tasks carried out on a traditional physical server. A VM runs its own operating system. Although a VM has been allocated only a portion of underlying physical hardware resources, it still behaves like a complete computer. The virtual servers are called guests and the physical server providing hardware is called host as it hosts these virtual servers. Server virtualization can be broadly categorized into two more categories: Software based virtualization and hardware assisted virtualization.

2.1.4.1 Software Based Virtualization

Software based virtualization applies privileged/sensitive instruction trap and emulation in software. It can be further classified into 3 types.

Full Virtualization with Binary Translation

In this form of virtualization, virtual machines are unaware of neither the hypervisor nor the co-existing virtual machines running on the same host. As VMs are unaware of virtualization, they have the illusion of having all the physical resources dedicated to them and thus resulting the need for entire hardware emulation. VMs run unmodified OS and each VM can run OS of its choice irrespective of host OS or other co-existing VM OSes.

Binary translation is the technique used by hypervisor to trap and emulate the execution of non-virtualizable privileged instructions. Along with instruction trapping, the hypervisor also has to handle VM physical resource allocation. Thus, hypervisor is heavily burdened and also this technique incurs large performance overhead compared to natively virtualized architectures. VMware [12] and VirtualBox [26] are some of the hypervisors based on binary translation.

Paravirtualization with Hypercalls

In Paravirtualization, virtual machines are aware of the virtualization environment. This avoids the need for instruction trapping thus overcoming performance latency. Thus VMs have to run tweaked OS that suits Paravirtualization. Paravirtualization uses hypercall for communication with hypervisor as it does not use trapping anymore. A hypercall to a hypervisor is similar to a system call to kernel.

Paravirtualization has reduced complexity and better I/O performance compared to full virtualization techniques. Xen [27] is the best known hypervisor of this type.

2.1.4.2 Hardware Assisted Virtualization

Hardware assisted virtualization enables efficient full virtualization utilizing underlying host hardware capabilities. HAV techniques minimize the involvement of the host system in managing guest's privilege and address space translation.

In 2005 and 2006, Intel and AMD introduced virtualization support known as Intel VT-x [28] and AMD-V [29] respectively, in the processor hardware by providing a new privilege level, ring 1 which can be used to run the hypervisor, letting the guest OSes continue accessing CPU in ring 0 similar to running on a physical host. This helps in achieving running unmodified virtual machines without binary translation overhead.

Using hardware assisted virtualization, the guest OS has direct access to hardware resources without any emulation or modification dissimilar to software virtualization where VMM emulates entire hardware to guest OS.

2.1.4.3 Nested Virtualization

Nested virtualization, as depicted in Figure 2.1, is one of the complex and interesting forms of virtualization where we run virtual machines inside another virtual machine. It is virtualization inside a virtualized environment. It has two separate independent hypervisors operating at different levels. Level 1 or L1 hypervisor runs inside the VM and the Level 0 or L0 hypervisor is responsible for creating the VM hosting L0 hypervisor. Nested virtualization provides enhanced flexibility and significant cost savings. However, not all hypervisors and operating systems provide support for nested virtualization. KVM and Xen [30] hypervisors are some supporting it.

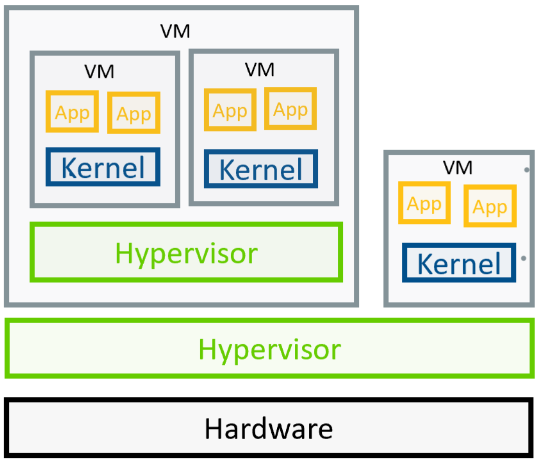


Figure 2.1: Nested Virtualization Architecture.

2.1.5 Hypervisor

Hypervisor or Virtual Machine Monitor is a software that handles creation, resource allocation and management of virtual machines. Hypervisors are mainly of two types:

2.1.5.1 Type I or Bare Metal or Native Hypervisor

The hypervisor runs directly on top of the hardware and has direct access to CPU, memory and I/O drivers. As shown in Figure 2.2, there is no middleware sitting between the hypervisor and hardware and hence it leads to enhanced security and better performance. However, expensive content management software is required to control host hardware and regulate multiple VMs.

2.1.5.2 Type II or Hosted Hypervisor

The hypervisor runs on top of the host OS, similar to an application program and does not have direct access to host hardware. Figure 2.3 presents the details of Type II hypervisor. This kind of hypervisors have degraded performance compared to Type I hypervisors because of extra

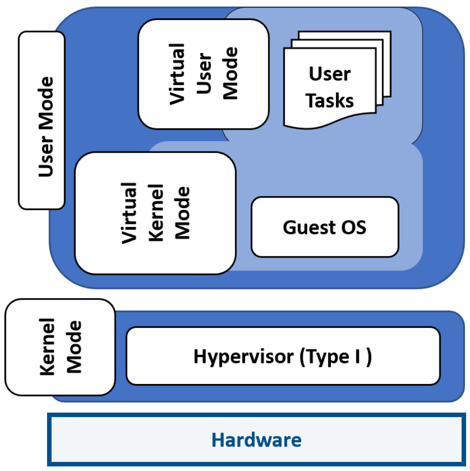


Figure 2.2: Bare Metal Hypervisor.

middleware. It does not need any external content management software making it cheap and also easy to manage.

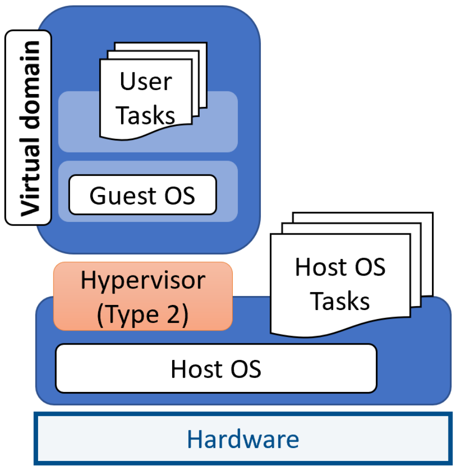


Figure 2.3: Hosted Hypervisor.

Some other types of virtualization forms are Data virtualization, Storage virtualization, Application virtualization, Data Center virtualization, CPU virtualization, GPU virtualization and Linux virtualization etc.

2.2 Containerization

In the above sections, we have gone through virtualization and the benefits gained using virtual machines. However, some of the biggest concerns with virtual machines is they are resource heavy and have larger start up times i.e., in the magnitude of seconds as their start up involves entire OS boot up [31]. Therefore, the need for something light weight and fast enters the scene. Thus containerization, a light weight alternative to full hardware virtualization, based on operating system level virtualization comes into picture.

Containerization is the process of encapsulating an application along with its dependencies i.e., libraries, binaries, frameworks, and configuration files etc. so that the applications can be run consistently on any platform. This technology lets one to create isolated user spaces called containers which are portable, light weight, efficient and easy to deploy. A container is a set of isolated processes which share the host kernel with limited access to underlying hardware resources. Differences between VMs and containers is illustrated in Figure 2.4.

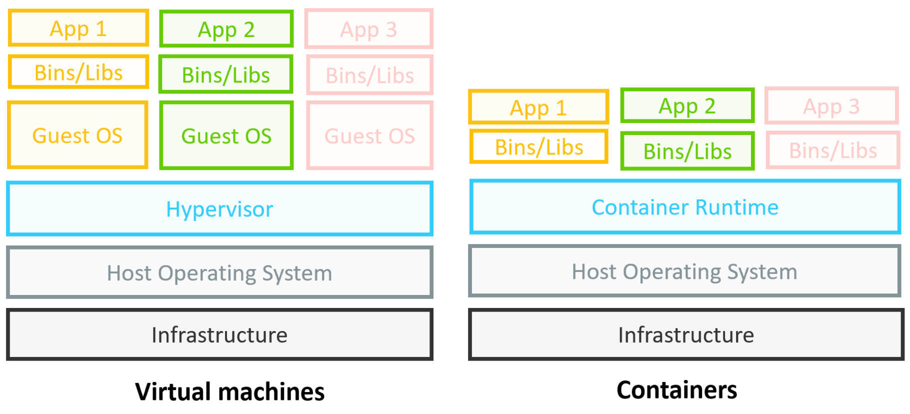


Figure 2.4: Difference between Container and Virtual Machine.

Container runtime engine is the software that sits on the top of host OS and handles container resource allocation and management. It conduits the sharing/allocation of OS resources among various containers. Docker [16], RKT [32], CRI-O, and LXD [17] are some of the widely used container engines. Container runtime is the lowest level component of container engines that actually runs containers from Image files. runC is the Open Container Initiative (OCI) Runtime Standard reference implementation. Docker, CRI-O, and many other container engines rely on runC. Although runC is the most widely adopted container runtime, there exists other OCI compliant runtimes, such as crun, Firecracker [22], gVisor [20], Katacontainers [23] and Nabla Containers [33]. Container runtime is responsible for:

1. Consuming the container mount point provided by the container engine
2. Consuming the container metadata provided by the container engine
3. Kernel communication to start containerized processes
4. Cgroups set up
5. SELinux policy set up
6. AppArmor rules set up

At the lowest level, container runtime provides the ability to start a container consistently, no matter the container engine. runC expects a mount point (directory) and meta-data (config.json) to be provided to it in order to start a container.

Apart from the host OS, containers also share common container layers, like common bins and libraries, among multiple containers. The shared OS components remain read only and the modifiable parts are mounted specifically to every container. As a container is abstracted from the host OS, it resolves the OS dependency problem thus letting to achieve the motto, "Written once and run anywhere". Containers are stand-alone independent executables. They can be run on any compatible hardware without worrying about dependencies.

The small resource footprint and portability provides room for supporting exponentially large number of containers to run on the same hardware compared to VMs, thus dramatically cutting down the infrastructure costs. Docker is one of the most famous container runtimes.

2.2.1 Benefits

Some of the significant benefits offered by containers are

1. Portability: Container is a standalone independent executable with all dependencies, libraries, and bins etc. Hence, without any modifications they can be ported from one platform to another and run successfully irrespective of the new platform.

2. Fast: Containers use the underlying host OS and thus able to avert own OS overhead making them light weight. Containers spin up faster as their start up does not involve entire OS boot up.

3. Efficiency: Virtual machines are based on hardware abstraction whereas containers are based on host OS abstraction. As containers don't need to have their own OS, network, stack, memory etc., their size typically varies in MB's, whereas VMs are in sizes of GB's. This permits us to run multiple containers on a single server compared to number of virtual machines.

4. Fault isolation: As containers are a set of isolated processes, any failure in one container does not impact the working of any of its co-hosted containers. The malfunctioning container can be destroyed or repaired without causing any downtime for the other containers.

2.2.2 Container Components

All the processes running within a system can be classified into two groups: User space processes, Kernel space processes. Kernel space processes will be running in ring 0 and can execute privileged instructions whereas User space processes running in ring 3 have to invoke system calls to run privileged instructions.

The following sections describes two main components that make up the skeleton of the LXC, namely Cgroups and Namespaces.

2.2.2.1 Cgroups

Linux kernel feature control groups, a.k.a. Cgroups, isolate and control resource allocation and usage by user processes. They limit the amount of CPU processors, memory, and network etc.

resources allocated to a set of processes. Thus, they help in prioritizing applications which ones to get resources as there is always physical limit on available resources. They help in monitoring and measuring resource usage.

2.2.2.2 Namespaces

Namespaces, another linux kernel feature, is a collection of processes which share same resource constraints. There are multiple namespaces within a system. The resource allocation for these namespaces can be handled through Cgroups. It eventually led to the new feature i.e., namespace isolation, which gives illusion to the set of processes within a namespace of being in a isolated environment away from the main host system. Some of the notable namespace isolation features that paved path for the creation of containers are:

2.2.2.3 PID (Process Identifier) Namespaces

The process within one PID namespace can't see the processes running in other namespaces.

2.2.2.4 Network Namespaces

Isolates low level networking tools like iptables, network interface controllers etc.

2.2.2.5 User Namespaces

Users within a namespace are restricted to accessing other namespaces thus avoiding User ID conflicts.

2.2.2.6 Mount Namespaces

Provides isolation of the list of mount points seen by the processes within each namespace instance.

After going through container architecture, it is evident that containers are not as secure as VMs because of the host OS sharing. All the users related to a container are granted access to the kernel root account. Consequently, the user is able to access all the containers shared under the linux kernel and can break into those containers. Also, the security scanners running inside containerized environments protect the OS but not application containers.

3. CASE STUDY

In this section we will thoroughly go through various container runtimes and evaluate their performance.

3.1 Container Runtimes

Container runtime is the lowest level component of container engine that actually runs containers from image files. High-level container runtime handles image pull/push from repositories while low-level container runtime manages OS features. Some of the widely used secure and unsecure container runtimes are listed below:

3.1.1 runC

runC is an OCI compliant CLI tool for spawning and running containers based on runtime-spec repository [34]. It relies on Seccomp, SeLinux, or AppArmor for security policies. As it is based on system call filtering, it is not very secure. runC architecture is clearly listed down in Figure 3.1, reused from [35] under Creative Commons Attribution 4.0 International (CC BY 4.0) License. runC does not specifically provide security related features as it assumes defense by default paradigm. It assumes host system to be safe and secure and expects host system to have defense mechanisms in place. CVE-2019-5736 [36] is one of the runC's security vulnerability in recent times which allows attacker to overwrite the host runC binary and consequently obtain host root access.

3.1.2 gVisor

gVisor [20], an OCI-compatible sandbox runtime, that sits in between the host OS and applications, provides a virtualized container environment. Container created through gVisor runtime provide a similar level of isolation as provided by VM. But these containers are more resource lightweight than a VM.

gVisor performs the mixed job of VMM and guest kernel. As illustrated in Figure 3.2, adapted

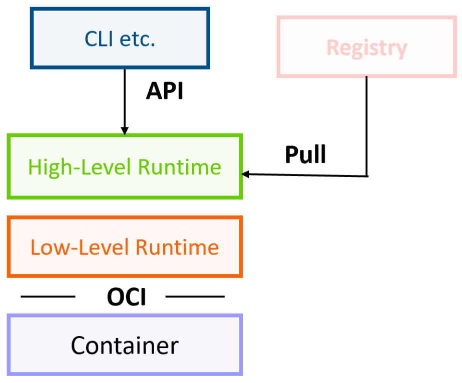


Figure 3.1: runC Architecture.

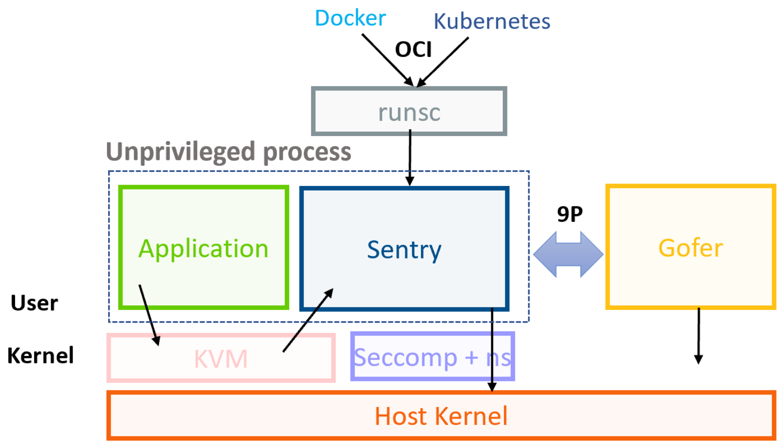


Figure 3.2: gVisor Architecture.

from [37] under Apache License 2.0, the core of gVisor is a lightweight userspace kernel Sentry which runs as a normal, unprivileged process and implements most of the linux system calls and essential kernel functions such as signal delivery, memory management, network stack, and threading model. gVisor sandboxes applications by intercepting all application's system calls and handling them with Sentry, all while in user space before forwarding to the host kernel.

gVisor has its own limitations. One of the major limitations of gVisor is backward-compatibility as it uses a custom-made kernel. gVisor implemented 211 of the 319 x86-64 system calls and uses only 64 system calls in the host system. Even if all the system calls are implemented still it can't achieve complete backward-compatibility status as these system calls are re-implementations of the original linux system calls.

Moreover, it is also not suitable for system call heavy applications because of the overhead incurred for intercepting and handling sandboxed application's system call. Thus, the applications that use unimplemented system calls can't be run in gVisor. gVisor lacks direct hardware access, so applications that require hardware access such as Graphics Processing Unit (GPU) cannot run in these virtualized environments.

3.1.3 Firecracker

Firecracker [22] is, an open source VMM based on linux KVM virtualization infrastructure, capable of creating and managing secure, multi-tenant containers. It launches lightweight virtual machines (MicroVMs) for every firecracker process i.e., container. Firecracker enhances both the security and performance by providing each guest VM with minimal OS features and devices. Each Firecracker instance runs as a user space process.

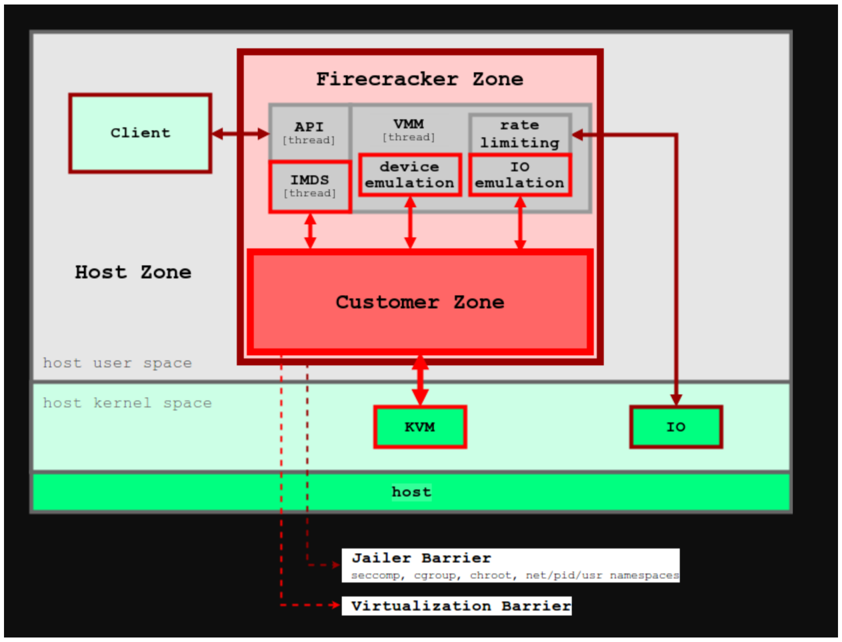


Figure 3.3: Firecracker Architecture.

Figure 3.3, reused from [38] under Apache License 2.0, illustrates Firecracker architecture. Firecracker provides a REST-based configuration API and device emulation for disk, networking and serial console. It is compatible with all arbitrary linux binaries and libraries and does not need any code changes or recompilation. It allows resource soft allocation overcommiting CPU, memory etc.

Each Firecracker process is locked down by the Seccomp, Cgroup, and namespace policies so that the system calls, hardware resource, file system, and network activities are strictly limited. It additionally implements Jailer, a wrapper around Firecracker, to safeguard against unwanted VMM behaviour. The Jailer puts Firecracker in a restrictive sandbox before guest boots up thus trying to mitigate code injection bug into VMM by guest.

Firecracker is not backward-compatible. It boots relatively recent linux kernels, and that too compiled with a specific set of configuration options. Also, it boots a minimal kernel config without relying on an emulated bios and without a complete device model. Moreover, Firecracker has not yet fully integrated with Docker and Kubernetes. It does not provide support for hardware passthrough, so applications that need GPU or any device accelerator access are not compatible. It has limited VM-to-host file sharing and networking models as well.

3.1.4 Kata-container

Kata-container [23] is an open source community working on building secure container runtime for creating lightweight virtual machines that feel and perform like containers, but provide stronger workload isolation using hardware virtualization technology as a second layer of defense. Kata will launch a lightweight virtual machine, which includes a minimal guest kernel and a guest image and uses the guest's linux kernel to create a container workload.

The kata containers runtime i.e., Kata-runtime is OCI-compatible runtime and is mainly based on Virtcontainers project [39]. It launches Kata-shim to monitor, control or reap container processes. A Kata-shim instance will both forward signals and stdin streams to the container process running on the guest and pass the container stdout and stderr streams back up the stack to the CRI shim via the container process reaper. Every container has its own Kata-shim daemon.

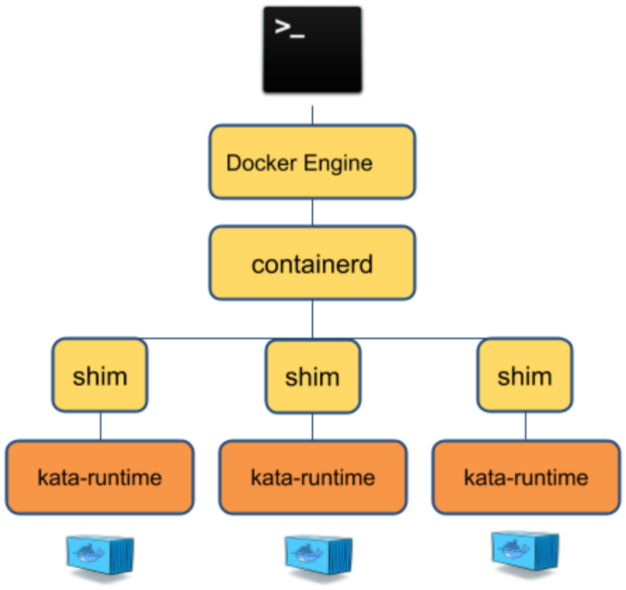


Figure 3.4: Kata-container Architecture.

Figure 3.4, reused from [40] under Apache License 2.0, presents entire Kata-container architecture depicting the roles of Kata-proxy, Kata-agent and Kata-shim etc. As Kata-runtime can run several containers per VM, we need a supervisor for managing containers and processes running within those containers and thus kata-agent comes into foreground. The Kata-agent execution unit is the sandbox. A Kata-agent sandbox is a container sandbox defined by a set of namespaces (NS, UTS, IPC and PID).

As Kata-container launch a whole new VM, all the unwanted OS overhead comes back negating the performance promises of containerization. Kata-containers are able to address security drawbacks of traditional containers but they suffer from heavy resource footprint and large spin-up times.

3.2 Performance Evaluation

In this section, we will evaluate the start up times and system latencies of container runtimes runC, gVisor, Firecracker, Kata-runtime, to get a sense of the performance of these products.

3.3 Evaluation Setup

The evaluation was performed on Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz. The CPU frequency is 900.052 MHz. It has 16GB RAM and RAM speed 2666MHz (0.4 ns). The disk size is 1863GiB (2TB), with 180MB/s read speed and 204MiB/s write speed. The system has Ubuntu 18.04 OS running and linux kernel version 4.15.0-76-generic

The version of these container runtimes is listed down:

- gVisor: release-20191213.0-37-g822d847ccaa1-dirty
- runC: 1.0.0-rc10+dev
- Firecracker: v0.15.2
- Kata-runtime: 1.10.0-rc0

We used lmbench-2.5 to benchmark system latency metrics. We performed evaluation for both KVM and Ptrace platforms of gVisor.

3.3.1 System Latency Metrics

Tables 3.1 & 3.2 provides insights into system latencies of containerization solutions. The metrics provided cover some of the important basic operations like system call, file open/close, read and write operations etc.

It can be noted that gVisor performs poorly in both the platform modes in almost all of the metrics. This might be resulting from the interceptions of all system call operations by Sentry before forwarding to host OS. On the other hand, Firecracker based on hypervisor technology is observed to have read/write, file open/close and system call durations on par with native system. The traditional container runtime runC has values in the acceptable ranges.

3.3.2 Spin-up Times

Table 3.3 shows the details of average spin up times, their range and confidence values from our comprehensive case studies. We evaluated the spin up times over 6 trials.

Microseconds	Ubuntu 18.04 (Baseline)	runC	overhead over Baseline	gVisor (KVM Platform)	Overhead over Baseline
Syscall	0.262	0.422	61%	0.484	84.7%
Read Op	0.436	0.806	84.8%	0.864	98.1%
Write Op	0.356	0.640	79.7%	0.693	94.7%
Stat	0.809	0.891	10.1%	1.637	102%
fstat	0.401	0.526	31.2 %	0.843	110%
File open/close	1.671	8.365	400%	19.77	1083%
500 FDs	5.064	3.660	-27.7%	3.76	-25.7%
500 tcp FDs	37.339	25.66	-31.3%	80.632	115%
fork+exit	124.233	91.650	-26.2%	221.30	78.1%

Table 3.1: System latency metrics of various container runtimes.

Microseconds	gVisor (Ptrace Platform)	Overhead over Baseline	Firecracker	Overhead over baseline	katacontainer	Overhead over baseline
Syscall	7.936	2929%	0.265	1.1%	0.193	-26.3%
Read Op	8.272	1797%	0.401	-8%	0.254	-41.7%
Write Op	8.155	2191%	0.347	-2.5%	0.229	-35.7%
Stat	9.474	1071%	0.852	5.3%	599.101	73954%
fstat	8.43	2002%	0.428	6.7%	70.232	17414%
File open/close	517.91	30894%	1.789	7.1%	808.4853	48283%
500 FDs	58.62	1058%	4.455	-12%	3.002	-40.8%
500 tcp FDs	210.778	464%	N/A	N/A	24.322	-34.9%
fork+exit	3905.5	3044%	62.667	-49.6%	73.198	-41.1%

Table 3.2: System latency metrics of various container runtimes continued.

From the table it can be observed that gVisor Ptrace platform spin up metrics are on par with native container runtime spin up times. gVisor KVM platform also provides promising start up times. Firecracker which uses virtualization at the core is observed to have almost double the spin up times. Kata-containers suitable for all kinds of workload is far behind in the race recording very high spin up times averaging at 1.3 seconds.

Seconds	runC	gVisor (KVM Platform)	gVisor (Ptrace Platform)	Firecracker	Kata-container
Mean	0.369	0.381	0.361	0.622	1.302
Mean overhead over runC	N/A	3.2%	-2%	68.5%	252%
Upper Val	0.382	0.393	0.388	0.664	1.877
Upper Val overhead over runC	N/A	2.9%	1.5%	73.8%	391%
Lower Val	0.356	0.368	0.335	0.582	0.727
Lower Val overhead over runC	N/A	3.3%	-5.9%	63.4%	104%
Std	0.016	0.016	0.033	0.05	0.719
Std overhead over runC	N/A	0%	106%	212.5%	4393%
Confidence value (alpha = 0.05)	0.013	0.013	0.026	0.04	0.57

Table 3.3: Various evaluation metrics for runC, gVisor, Firecracker and Kata-containers spin-up times.

3.4 Conclusion

After comprehensive case study of the performance evaluation of these solutions, we come to the belief that there always exists a trade off between application compatibility and performance. Non customized kernels are the only way to achieve backward compatibility and only customized kernels can provide better performance. A perfect blend between these two requirements can pave paths for new secure, resource efficient compatible containerization solutions.

4. PROPOSED ARCHITECTURE

In our quest for secure, backward-compatible, resource efficient container solution, we come to the conclusion that there always exists a trade-off between choosing greater security and more overhead or weaker security and less overhead. So hereby, we propose a new approach to build secure, lightweight, backward-compatible containers using the Xen hypervisor. Using VM cloning, a container can be quickly spun up in a VM that is identical to the container-hosting VM. The containers built this way don't need any underlying kernel modifications or external agents or proxies and are more efficient than spinning up a whole new VM for a container.

4.1 Xen Hypervisor

Xen is primarily a Type I hypervisor based on Paravirtualization technology. It uses hypercalls to run privileged instructions. Figure 4.1 gives clear idea of Xen architecture. The guest VMs ran in isolated environments are called domains. Xen supports two types of virtualized guests: Paravirtualized Virtual Machines (PV) and Hardware-assisted Virtual Machines (HVM).

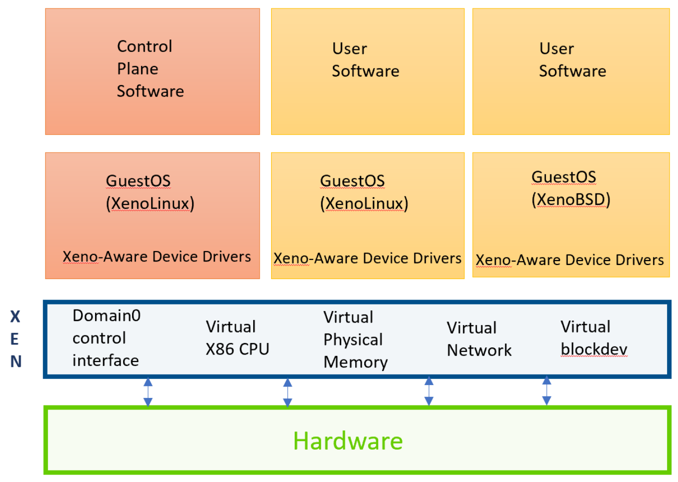


Figure 4.1: An Overview of Xen.

PV guests run modified guest OS and thus able to achieve near native performance whereas HVM run unmodified guests providing scope for backward compatibility but have high overhead. However, with the advent of PV drivers for HVM guests, HVM have also improved a lot in terms of performance.

During bootup, Xen loads dom0 guest kernel which holds super privileges. The other unprivileged domains domUs are created later on demand. Domain 0 handles networking, device drivers and interface to Xen whereas Xen takes care of CPU processing and memory scheduling. Dom0 handles all the communication with hardware drivers and is responsible for multiplexing them to guests. Xen creates new domU guests in response to a hypercall request from dom0.

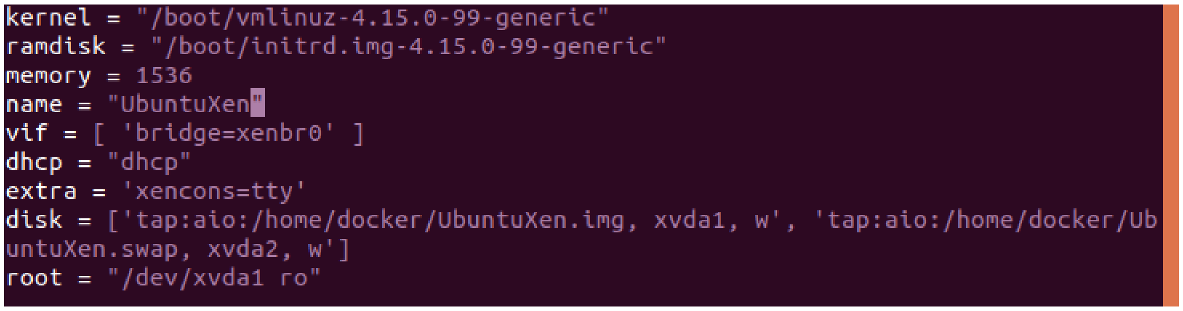
4.1.1 PV Guest Creation

The following process lists the 11 steps that need to be followed in order to create a PV guest

1. Create logical volume for rootfs
2. Create logical volume for swap
3. Create filesystem for rootfs
4. Mount rootfs
5. Install operating system to the guest root filesystem using debootstrap
6. Copy appropriate modules from host OS to guest rootfs
7. Configure root password for new guest
8. Create a VM config file for the guest.
9. Configure system files for the guest OS and modify /etc/fstab
10. Unmount the guest filesystem
11. Boot the guest OS

4.1.1.1 *PV Guest Config File*

The config file for a typical PV guest looks like as shown in Figure 4.2



```
kernel = "/boot/vmlinuz-4.15.0-99-generic"
ramdisk = "/boot/initrd.img-4.15.0-99-generic"
memory = 1536
name = "UbuntuXen"
vif = [ 'bridge=xenbr0' ]
dhcp = "dhcp"
extra = 'xencons=tty'
disk = ['tap:aio:/home/docker/UbuntuXen.img, xvda1, w', 'tap:aio:/home/docker/Ub
untuXen.swap, xvda2, w']
root = "/dev/xvda1 ro"
```

Figure 4.2: Config File for PV Guests.

4.1.2 HVM Guest Creation

HVM guest does not actually need swap space, its creation is very easy compared to PV type and the creation steps are as follows:

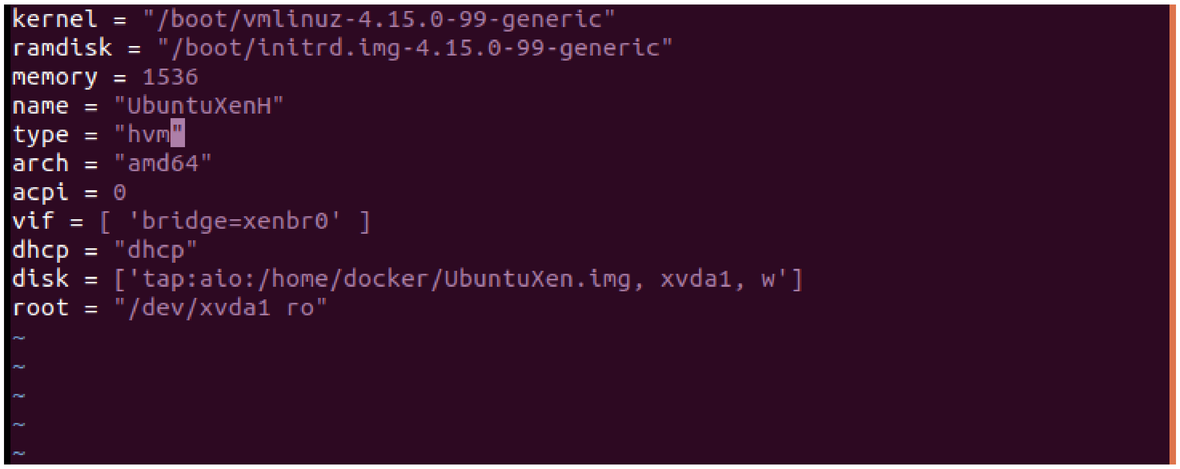
1. Create a logical volume for rootfs
2. Install operating system
3. Create a VM config file for the guest.
4. Boot the Guest OS

4.1.2.1 *HVM Guest Config File*

Figure 4.3 shows a typical config file for a HVM guest.

4.2 Interdomain Communication

The various VMs running on Xen hypervisor need to communicate with each other for multiple reasons. Unprivileged domains DomUs need to interact with dom0 for hardware access. Xen



```
kernel = "/boot/vmlinuz-4.15.0-99-generic"
ramdisk = "/boot/initrd.img-4.15.0-99-generic"
memory = 1536
name = "UbuntuXenH"
type = "hvm"
arch = "amd64"
acpi = 0
vif = [ 'bridge=xenbr0' ]
dhcp = "dhcp"
disk = ['tap:aio:/home/docker/UbuntuXen.img, xvda1, w']
root = "/dev/xvda1 ro"
~
~
~
~
~
```

Figure 4.3: Config File for HVM Guests.

supports two basic interdomain operations on memory pages: sharing and transferring, which facilitate interdomain communication.

Grant table is an interface that provides memory sharing between domains. Each domain has its own grant table that is shared with Xen. It is a data structure that holds information about what permissions other domains hold on current domain pages. Grant references are used for indexing in grant tables. A grant reference is basically an integer assigned to every shared page to uniquely identify it. Grant reference also provides the context of capabilities that grantee holds on granter's memory.

The local domain's kernel first notifies hypervisor of its interest in sharing a page with other domains. The local domain then passes a grant table reference ID to the remote domain that it is "granting" access to this page. Once the intended operation by remote domain is performed, local domain revokes the grant. The exact grant sharing mechanism between two dom's is listed below:

1. domA creates a grant access reference, and transmits the ref id to domB.

2. domB uses the reference to map the granted frame.

3. domB performs the memory access.

4. domB unmaps the granted frame.

5. domA removes its grant.

4.3 Copy-on-Write

Copy-on-Write is an efficient resource management technique while copying process's data resources within a system. It is waste of resources to have two copies of identical replicas in the system. It creates a new resource only when modifications occur to the original. Till then, the resource is shared between the copy and the original.

4.4 Extended Page Tables

Extended page tables is a feature of hardware assisted virtualization to reduce memory overhead. In virtualization, we typically have guest logical address, guest physical address, host physical address and machine frame number. Machine frame number is the actual hardware address where data can stored to or retrieved from.

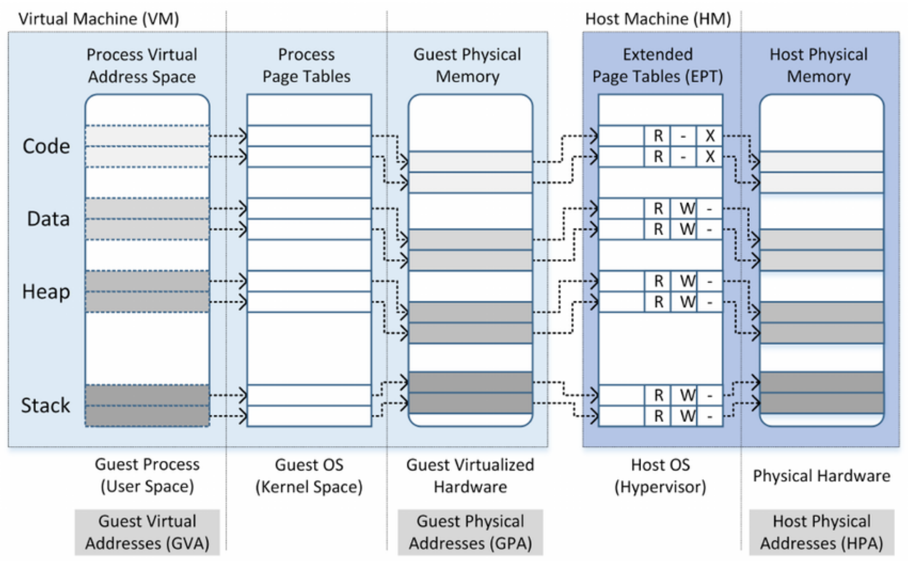

Virtual Machine (VM)
Host Machine (HM)
Process Virtual Address Space
Process Page Tables
Guest Physical Memory
Extended Page Tables (EPT)
Host Physical Memory
Code
Data
Heap
Stack
R - X
R W -
Guest Process (User Space)
Guest OS (Kernel Space)
Guest Virtualized Hardware
Host OS (Hypervisor)
Physical Hardware
Guest Virtual Addresses (GVA)
Guest Physical Addresses (GPA)
Host Physical Addresses (HPA)

Figure 4.4: Extended Page Table.

Figure 4.4, reused from [41] under Springer Nature License, shows how guest logical address is mapped to machine frame number. Each domain has a page table which holds the mappings from guest virtual to guest physical address. However, this page table is of read only type. VMM hosts a EPT table for every domain and performs the mappings from guest physical to the host physical. These EPT tables are both read/writeand whenever guest tries to modify its own page tables, a page fault occurs and traps to VMM. VMM checks modification permissions for this page and acts accordingly.

4.5 VM Checkpointing

Checkpoint is a snapshot of the state of the virtual machine. VM checkpointing is usually performed to create VM backups which can be used to restore VM to the previous state in case of failures in no time.

4.5.1 Suspend-save-restore VM Checkpointing

Save operation preserves the guest's running state intact ranging from CPU, device drivers state to memory etc. As first step of save operation the virtual machine is paused, its current memory state saved to specified location and then VM can be unpaused or stopped as specified.

The restore operation launches a new VM using previously saved memory state file. The virtual machine resumes from the point where it was previously stopped. It won't go through boot process during start up.

4.6 Proposed Architecture

We plan to use all of these existing features of Xen in building our prototype. Our implementation involves modifications to the Xen hypervisor and some library routines in libxl. We inherently assume the support for hardware virtualization.

This prototype initially has a running dom0 and a single domU guest. For our experiment, we plan to use a HVM guest type domU. DomU hosts a container executing hello world program. Our aim is to create an exact clone of domU. Let's call our current domU as domA and it's clone be domB. domB needs to be exactly identical to domA with same memory, CPU and device driver states. domB also need to start from the state where it was created from domA. After launching domB, we kill the container running in domA and establish the communication between domA and domB.

DomU does not have privileges to spin up another VM. Only dom0 holds these priviliges. So as a first step we initialize the communication between dom0 and domA. Secondly, we want dom0 to create domB once it receives "CLONE" message id from domA. We pass this message to dom0 via container hosted on domA.

On reception of the CLONE message from domA, we used Suspend-save-restore checkpointing. Dom0 suspends domA and saves all of domA state to a checkpoint file. After successful saving, domA is resumed. Now, dom0 restores domB using the checkpoint file. Once domB is created, container inside domA is killed.

For establishing the communication between domA and domB, both domU's need to know domId of each other. domB already has domA id given it is actually created from domA and we initially saved domA id in local variable. On the other hand, domA has no clue about domB id and domA does not have sufficient permissions to directly inquire about domB id. dom0 comes to the rescue in this phase. It knows domB id given its supervisor privileges. After domB creation, dom0 passes domB id to domA through previously established communication channels.

4.6.1 Algorithm

1. Create a HVM guest domA and run container inside it.
2. Establish the communication between dom0 and domA
3. Send CLONE message from domA to dom0.
4. Dom0 suspends domA and saves domA's current running state including cpu, memory, disk, device driver states to a checkpoint file.
5. Dom0 resumes domA after successfully saving the state.
6. Dom0 now launches a new domU i.e., domB using the checkpoint file.
7. Dom0 communicates domB id to domA

8. Once domB creation finishes, container running inside domA is exited

9. Communication between domA and domB is established for future operations.

Figures 4.5 to 4.9 show the above discussed steps of the algorithm.

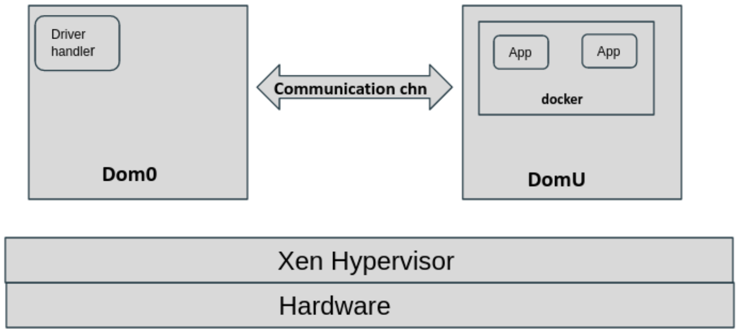


Figure 4.5: Initial State.

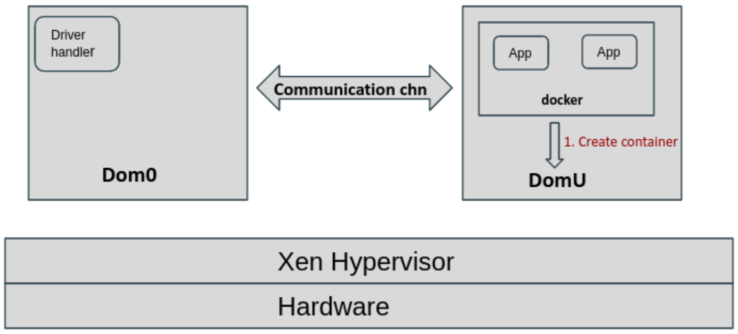


Figure 4.6: Step 1.

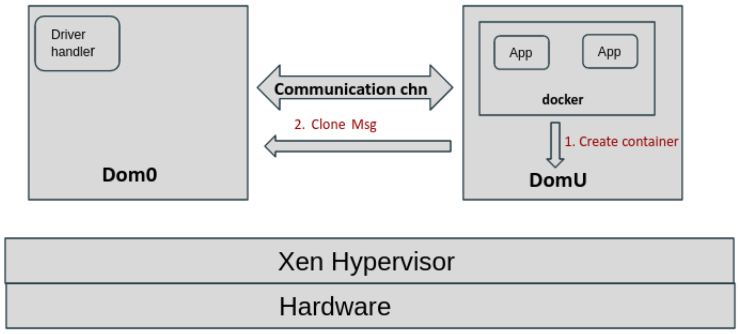


Figure 4.7: Step 2.

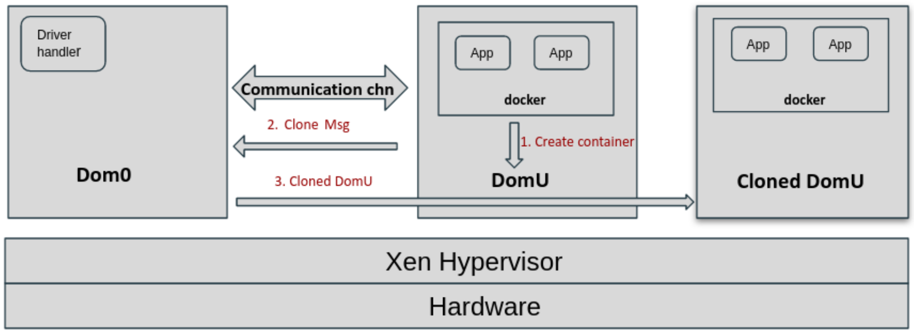


Figure 4.8: Step 3.

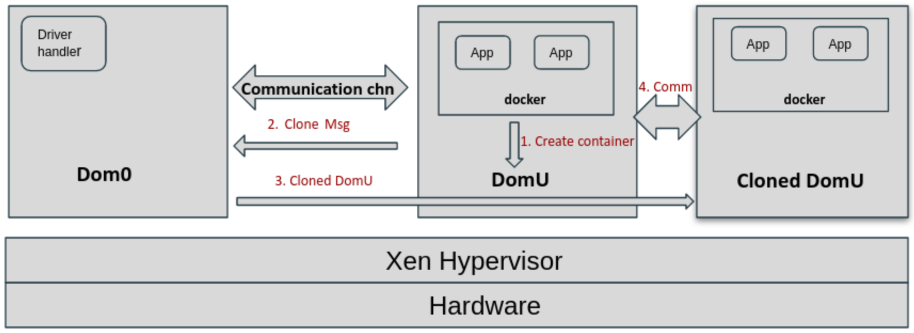


Figure 4.9: Step 4.

4.7 Conclusion

We have built our prototype without needing any kernel changes or external agents. This is more efficient than spinning up a new VM for a container. Thus our prototype demonstrates the feasibility of a virtualization based containerization solution.

5. SUMMARY AND CONCLUSIONS

The prototype we built is based on existing Xen features. It only needs slight modifications to the hypervisor code and no other OS features of host or guest system are modified. We prototyped with HVM guests (which run unmodified OS). Thus, the solution we built is completely backward compatible. Also, as we have a individual VM for each container, this solution is secure. For creating a new container we cloned the original VM hosting the container using VM cloning techniques. Thus, our prototype establishes the workflow for using VM cloning for building backward compatible, secure containers.

Now, we will evaluate our prototype. We are mainly interested in system latency and spin-up time metrics.

5.1 Performance Evaluation

In this section, we will evaluate the startup times and system latencies of container runtimes runC, gVisor, Firecracker, Kata-runtime, to get a sense of the performance of these products.

5.2 Evaluation Setup

The evaluation was performed on Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz. The CPU frequency is 3GHz. It has 16GB RAM and RAM speed 2666MHz (0.4 ns). The disk size is 1863GiB (2TB), with 180MB/s read speed and 204MiB/s write speed. The system has Ubuntu 18.04 OS running and linux kernel version 4.15.0-76-generic. The CPU has support for nested virtualization and VT-X is also enabled.

We hosted Xen inside a virtual machine created using VirtManager. The version of these container runtimes and Xen is listed down:

- gVisor: release-20191213.0-37-g822d847ccaa1-dirty
- runC: 1.0.0-rc10+dev
- Firecracker: v0.15.2

- Kata-runtime:1.10.0-rc0
- Xen: 4.14-unstable
- Virt-manager: 1.5.1

We used lmbench-2.5 to benchmark system latency metrics. We performed evaluation for both KVM and Ptrace platforms of gVisor.

5.3 System Latency Metrics

It is evident from table 5.1 that our prototype is able to achieve almost similar or better performance compared to traditional container runtime. System call metrics of our prototype are much better than gVisor and almost similar to runC. However, still our prototype system call duration is twice the normal PC system call duration. We can say that our prototype works well for system call heavy applications. And as our prototype does not do kernel modifications, it supports all system calls.

Micro seconds	runC	gVisor (KVM Platform)	gVisor (Ptrace Platform)	Firecracker	Kata container	Normal PC	Our Protoype
Syscall	0.422	0.484	7.936	0.265	0.193	0.262	0.3982
Read Op	0.806	0.864	8.272	0.401	0.254	0.436	0.4822
Write Op	0.640	0.693	8.155	0.347	0.229	0.356	0.4566
Stat	0.891	1.637	9.474	0.852	599.101	0.809	0.7245
fstat	0.526	0.843	8.43	0.428	70.232	0.401	0.5005
File open/close	8.365	19.77	517.91	1.789	808.4853	1.671	2.7149
500 FDs	3.660	3.76	58.62	4.455	3.002	5.064	3.7849
500 tcp FDs	25.660	80.632	210.778	N/A	24.322	37.339	26.2927
fork+exit	91.650	221.30	3905.5	62.667	73.198	124.233	284.8947

Table 5.1: System latency metrics of various container runtimes along with our prototype.

Read operation of our prototype is almost equivalent to normal pc and almost performed at double the rate compared to runC. This implies our prototype can manage read heavy applications. Our prototype's write operation is also 1.5 times faster than runC but it is slower compared to normal PC by narrow margin. Table 5.2 lists the various evaluation metrics of container runtimes.

Stat and fstat operations which fetch meta data about inodes and files respectively are performed almost at same speed as normal PC. File open/close speeds are also very promising in comparison. Our prototype performed better than all other container runtimes in file related operations. However, our solution is very slow during fork and exit.

5.4 Spin-up Times

Seconds	runC	gVisor (KVM Platform)	gVisor (Ptrace Platform)	Firecracker	Kata-container	Our Prototype
Mean	0.369	0.381	0.361	0.622	1.302	28.298
Upper Val	0.382	0.393	0.388	0.664	1.877	32.160
Lower Val	0.356	0.368	0.335	0.582	0.727	24.435
Std	0.016	0.016	0.033	0.05	0.719	3.862
Confidence value (alpha = 0.05)	0.013	0.013	0.026	0.04	0.57	5.35

Table 5.2: Various evaluation metrics for runC, gVisor, Firecracker, Kata-containers and our prototype spin-up times.

Our prototype currently has huge spin up times in the order of 28 seconds. This is largely because of using Suspend-save-restore cloning technique used in the prototype. This cloning involves snapshoting the current VM's entire state to a checkpoint file and then creating a new VM out of that checkpoint file. During the creation phase itself, memory allocation (i.e., actual physical frames are allocated) takes place which causes very high spin-up times.

5.5 Future Work

We plan to implement COW based cloning in our future work. Suspend-save-restore VM checkpointing technique incurs huge VM down times whereas COW-based checkpointing [42] is more resource efficient and have less VM down times. We will use EPT provided by Intel-VT.

We will modify EPT of the domain whose clone we are interested in. We will walk through domU's entire EPT and change all the read/write type pages in the last level to shared type. During clone operation, we will replicate an exact copy of the original domU for the cloned domU. This obviates the necessity for unnecessarily creating duplicate physical frames until both domains hold the same information.

Now in this case when one of the dom's want to modify its page, a usual page fault occurs. VMM first checks the permission bits for that page and if the pages are shared type, VMM will now do actual new physical frame allocation and reverts the page types from shared to read/write type in both domU tables. At this point, this page of the two domUs deviate and each have their own copy. This new allocation ensures that a change in the memory of one domU is not visible in another domU.

REFERENCES

[1] J. Voas and J. Zhang, "Cloud computing: New wine or just a new bottle?," *IT Professional*, vol. 11, no. 2, pp. 15–17, 2009.

[2] J. Jing, A. S. Helal, and A. Elmagarmid, "Client-server computing in mobile environments," *ACM Comput. Surv.*, vol. 31, p. 117–157, June 1999.

[3] K. Y. Tam, "Dynamic price elasticity and the diffusion of mainframe computing," *Journal of Management Information Systems*, vol. 13, no. 2, pp. 163–183, 1996.

[4] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *2008 Grid Computing Environments Workshop*, pp. 1–10, 2008.

[5] W. Kim, "Cloud computing: Today and tomorrow.," *Journal of object technology*, vol. 8, no. 1, pp. 65–72, 2009.

[6] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[7] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess, "Server virtualization in autonomic management of heterogeneous workloads," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 139–148, IEEE, 2007.

[8] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[9] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective hardware/software memory virtualization," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 217–226, 2011.

[10] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[11] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[12] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.," in *USENIX Annual Technical Conference, General Track*, pp. 1–14, 2001.

[13] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, Feb. 2008.

[14] D. A. Menascé, "Virtualization: Concepts, applications, and performance modeling," in *Int. CMG Conference*, pp. 407–414, 2005.

[15] T. Abels, P. Dhawan, and B. Chandrasekaran, "An overview of xen virtualization," *Dell Power Solutions*, vol. 8, pp. 109–111, 2005.

[16] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[17] R. Rosen, "Linux containers and the future cloud," *Linux J*, vol. 240, no. 4, pp. 86–95, 2014.

[18] D. Kakadia, *Apache Mesos Essentials*. Packt Publishing Ltd, 2015.

[19] A. Polvi, "Coreos is building a container runtime, rkt," *Retrieved*, vol. 4, p. 2015, 2014.

[20] "gvisor." https://github.com/google/gvisor.

[21] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[22] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 419–434, USENIX Association, Feb. 2020.

[23] "Kata container." https://github.com/kata-containers/runtime.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pp. 51–62, 2009.

[25] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[26] J. Watson, "Virtualbox: bits and bytes masquerading as machines," *Linux Journal*, vol. 2008, no. 166, p. 1, 2008.

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[28] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization.," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[29] "Amd-v." https://www.amd.com/en.

[30] "Xen project." https://github.com/xen-project/xen.

[31] K. Agarwal, B. Jain, and D. E. Porter, "Containing the hype," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[32] X.-L. Xie, P. Wang, and Q. Wang, "The performance analysis of docker and rkt based on kubernetes," in *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pp. 2137–2141, IEEE, 2017.

[33] "Nabla containers." https://nabla-contain-ers.github.io/.

[34] "runtime-spec." https://github.com/opencontainers/runtime-spec.

[35] "runc architecture." https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes.

[36] "Cve-2019-5736." https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[37] "gvisor architecture." https://github.com/google/gvisor/blob/master/website/assets/images/2019-11-18-security-basics-figure1.png.

[38] "Firecracker architecture." https://github.com/firecracker-microvm/firecracker/blob/master/docs/images/firecracker_threat_containment.png.

[39] "virtcontainers." https://github.com/containers/virtcontainers.

[40] "Kata architecture." https://github.com/kata-containers/kata-containers/blob/2.0-dev/docs/design/architecture.md.

[41] A. Luțaș, A. Coleșa, S. Lukács, and D. Luțaș, "U-hipe: hypervisor-based protection of user-mode processes in windows," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 1, pp. 23–36, 2016.

[42] M. H. Sun and D. M. Blough, "Fast, lightweight virtual machine checkpointing," tech. rep., Georgia Institute of Technology, 2010.