Comparison of Operating System Level Virtualisation Frameworks on UNIX like Systems.

Stefan Lendl, Selberherr, Weinbub

Jan 2018

In recent years operating system level virtualization or more commonly known as containers have become one of the standard means of execution in shared execution environments. This paper gives an overview of the techniques behind the OS-level virtualization technology and compares the most common solutions.

This paper examines OS-level virtualization and gives an overview of the differences compared to traditional virtualization techniques. An examination of several container frameworks like Docker, LXC/LXD, rkt and others shows some of their implementation strategies and characteristics. In particular isolation strategy between containers; availability of resource quotas; user privileges inside the container; the OCI compliance and the compatibility with common orchestration solutions are investigated and summarized.

Contents

1 Introduction **4**

2 Recent Work **5**

3 Technical Background **6**

Types of Virtualization 6
Linux Containers 8
namespaces 8
control groups 8
Seccomp 9
Capabilities 9
Open Container Standards 9
Orchestration 9

4 Container Frameworks **10**

chroot 10
FreeBSD Jails 11
Solaris Containers Zones 11
OpenVZ 11
LXC/LXD 12
Docker 12
rkt 14
Singularity 16
CRI-O 17
Kata Containers 17
Nabla Containers 19
gVisor 19
Firecracker MicroVM 19

5 Comparison **20**

6 Conclusion **21**

References **22**

1 Introduction

Historically every service offered would run on its own hardware. That immediately lead to inefficient use of resources when the service was not running at peak performance. In order to optimize the hardware utilization multiple services can be operated on the same hardware, sharing the hardware resources.

When operating multiple services on a single hardware the question of isolation between the services with regards to security arises. A compromised or malfunctioning service must not be able to compromise or effect other services on the same machine or take control over the entire system. This is also true for mobile devices were the user may install applications that must not be able to access or control certain parts of the underlying system or compromise other applications on the system.

This isolation is traditionally done using virtual machines (VM) were the hardware is abstracted and a full operating system runs inside the VM. In the last decade the approach of OS-level virtualization or containers has become increasingly popular where the OS is abstracted and each service gets an isolated view on the OS kernel.

2 Recent Work

According to researchers in [1] OS-level virtualization imposes a small overhead compared to native execution and offers better or equal performance than virtual machines. Researchers in [2] measured the performance overhead introduced by Docker. With a CPU overhead was shown to be 5% to 10% and the I/O overhead ranges from 10% to 30%. Researches in [3] compared Docker and rkt in Kubernetes and showed that Docker has a better CPU performance and rkt a better disk write performance. Researchers in **???** analyzed the impact of running Docker on top of KVM and showed a small overhead compared to execution directly in the VM. On ARM architecture virtual machine show a better performance for I/O tasks due to the caching mechanisms in the hypervisor [4].

Researches in [5] investigate and compared the security aspect of several — unfortunately outdated — container solutions.

3 Technical Background

There are various ways to isolate services and applications that are used in different scenarios and offer different characteristics and advantages. This section introduces an briefly describes techniques to share hardware resources for different services and describes the level of isolation and virtualization and performance impact.

Types of Virtualization

The simplest approach is to directly share the hardware by running multiple services or multiple instances of a service on a machine. Some examples of this approach are: running multiple websites on a single web server, running multiple web servers on the same machine or sharing a single database for multiple services. This is typically simple to implement and has the advantage of resource efficiency through the use of shared libraries. Communication between services is also simple because there is no isolation between them. One big issue with this approach are incompatible dependencies of services. Common Linux systems only allow the installation of a single version of a library which may lead to incompatibilities if a service requires a specific version of a library while another service requires another version. This issues is often referred to as *dependency hell*. Various techniques have emerged to conquer this issue but are not in the scope of this paper.

Having no isolation immediately leads to security issues. Therefore additional security measures are available to isolate the services from each other and the underlying base system. The common UNIX user privilege system can be used to prevent access from one service to another. Various Mandatory Access Control (MAC) systems go even further by defined more detailed access rules.

To create a maximum of isolation and even run different operating systems on the same hardware simultaneously each service can be executed in a virtual machine. Virtual machines provide the functionality to fully emulate the host system by creating virtual hardware interfaces that can be used by the system running inside the virtual machine. The guest system inside the virtual machine may be unaware of the fact that it is running in a virtual machine. The guest system may be executed on a different processing architecture than the host system. The sharing of the hardware resources is done by the drivers provided by the virtualization software, the so called hypervisor. The hypervisor is responsible for sharing the hardware resources amongst the guests according to configurable rates and limits.

Another techniques to isolate resources on a single host is the so called operating system(OS)-level virtualization which allows the execution of multiple isolated instances which are often called containers or jails [6]. A container does not emulate hardware but the OS kernel provides techniques to share the resources efficiently while isolating resources between individual containers and the host system. Containers use a set of features in the operating system kernel to segment resources of the host system and assign them to the container. The guest system directly uses the host operating system kernel interface and does not run its own kernel inside the container.

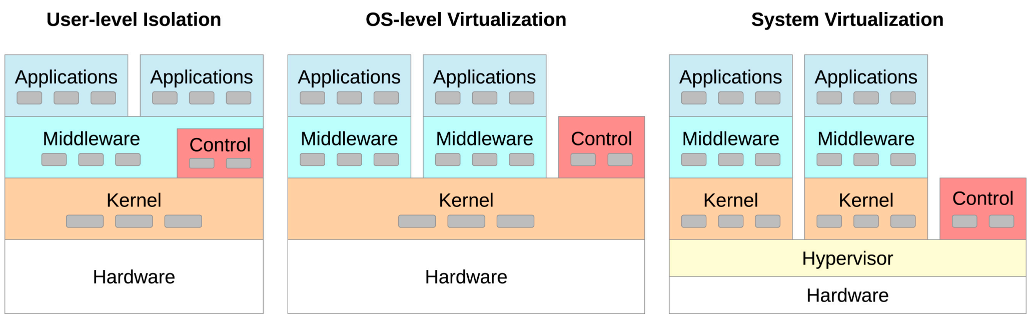

Figure 3.1: Types of Isolation [7]

Figure 3.1 shows the different types of isolation as described in [7] for an Android based evaluation. The left side shows user-level isolation where all applications run on the system system and isolation is provided by filesystem permissions or MAC. The right side shows full system virtualization where each application is executed in a separate virtual machine withe a separate operating system kernel. In the middle it shows OS-level virtualization where applications are executed on the same OS and isolation is provided by OS mechanisms and containers.

One of the main reasons for containers is their lightweight design and simple deployment mechanisms that have emerged in the ecosystem surrounding this technology. Several container technologies have emerged and the recent standardisation of has lead to solutions on top of containers. Containers are considered not save because the OS kernel represents a single point-of-failure [8]. In common Cloud networks the two technologies are combined by executing the container runtime on top of a virtual machine infrastructure. This allows the flexibility of deployment offered by containers while providing full isolation of VMs but also the performance penalty. There are approaches to increase the container security as proposed in [9],[10],[11]. Some of the container solutions presented in this paper focus on that aspect. A former Docker developer claims that containers are save with the appropriate settings [12]. Although configuring these settings is difficult especially for the general purpose.

Linux Containers

This section presents the mechanisms and concepts of isolation provided by the Linux kernel to implement containers.

The techniques used are not new and have been implemented into the Linux kernel over the years. Together the form the base for container solutions available currently. The most important features are namespaces [13] and control groups (cgroups).

namespaces

"Is was a process can see"

Applications necessarily consume named resources that appear on the host in various global namespaces. For example, an application might listen on a particular port, and this port is visible in the global network namespace of the host. To avoid applications clashing with each other and with other programs running in the host, the application is executed inside a collection of container-specific namespaces. For example, a network namespace isolates the IP addresses and ports used by an application from those in other network namespaces. The Linux kernel has been adding support for namespaces for several years, starting with a namespace of file system mount points, and, most recently, support was added for a namespace of users.

general reference on namespaces https://lwn.net/Articles/766124/

control groups

"Is was a process can use"

Applications running inside namespaces are still free to consume anonymous operating system resources such as memory and CPU. They can also operate directly on devices. The Linux kernel provides control groups to enable processes to be partitioned into hierarchical groups and subjected to constraints. Constraints are applied by resource controllers which participate in control groups and interface to the corresponding Linux kernel subsystems. For instance, the memory resource controller can limit the number of pages of real memory that may be consumed by a particular control group and can ensure processes are killed when the limit is about to be exceeded.

Garden uses five resource controllers—cpuset (CPUs and memory nodes), cpu (CPU bandwidth), cpuacct (CPU accounting), devices (device access), and memory (memory usage) and creates a control group for each container. The processes in the container are then subject to the constraints imposed by those resource controllers (except that cpuacct does not impose any constraints but accounts for CPU usage).

In addition to control groups, Garden uses a couple of other Linux features to impose limits on the processes in a container. Specifically, setrlimit restricts the consumption of certain resources by processes in the container and setquota restricts the consumption of certain other resources by users in the container.

cgroups v2 [14]

Seccomp

limit the system call that are available to the container.

Capabilities

Open Container Standards

In order to standardize the different implementations of containers several companies collaborate. The Open Container Initiative (OCI) [15] was formed under the Linux Foundation as an effort to define such standards. Also the Cloud Native Computing Foundation (CNCF) [16]. These are the most important standards [17]:

- the Image Specification [18] which defines the content of container images
- the Runtime Specification (CRI) [19] describes the "configuration, execution environment, and lifecycle of a container"
- the Container Network Interface (CNI) [20] specifies how to configure network interfaces inside containers, which was standardized by the CNCF.

Orchestration

Orchestration as defined in [21] and [22] describes the function of configuring, coordinating and managing a set of computer systems and software services.

There are many solutions available and most container frameworks provide there own solution or parts of it. Orchestration is not the focus of this paper but the integration in the most common generic orchestration frameworks OpenStack [23], Kubernetes [24], CloudFoundry [25] and Mesos [26] are investigated. Several other implementation are build on top of these solutions and are not mentioned here.

4 Container Frameworks

This section presents different container frameworks and highlights their properties and characteristics. The given mechanisms are presented in order of appearance and are not intended to be an exhaustive list. Some of the mechanisms or solutions are not maintained anymore but are mentioned due to their evolutionary importance.

chroot

A chroot environment or also called a chroot jail can not actually be called a container or virtualization system but it can be seen as the predecessor to the container technology.

A chroot environment is created by using the chroot system call of the operating system kernel to change the apparent root directory for a specific process and its child processes. It can be used to create a virtualized copy of the system software. It was added to the UNIX operating system in the year 1982 and is available in most UNIX-like operating systems like FreeBSD, Linux and Solaris. The process can only access files that are withing the assigned chroot directory structure. It is allowed to carry open file descriptors into the chroot environment which can be used for some level of privilege separation be leaving working files outside the chroot directory structure.

This is for example used for dependency management where multiple versions of a program with incompatible dependencies can be executed on a single host.

Limitations

The chroot mechanism only restricts the view of the filesystem for the process. It is not designed to defend against intention tampering by privileged root users. Users with sufficient privileges may break out of the chroot environment {https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html} by again using the chroot system call. A sufficiently privileged user can also create nodes for system devices and mount the filesystem on them which allows access to the hosts filesystem. The chroot mechanism cannot restrict or limit the access to system resources like input/output bandwidth, disk space or CPU time.

The chroot mechanism is limited in its functionality and security features which lead to the development of other mechanism which are explained in the following.

FreeBSD Jails

Jails were first introduced in FreeBSD in the year 2000 {https://www.freebsd.org/releases/4.0R/announce.h
in version 4.0 which added the jail system call.

FreeBSD jails aim to provide the same features as chroot and adds several security features to mitigate the security limitation of chroot. Each jail has its own user accounts, including the root user and separate processes. The activity of the processes in the jail are restricted. A processes in a jail does not see processes outside the jail and cannot interact with them. Modification of sysctl configuration settings is restricted. Each jail has its own IP address and manipulation of the network configuration, routing table and firewall rules are prohibited. Processes in a jail cannot create device nodes or mount filesystems.

resource limitations

jail templates ZFS jails inside jails

check this cite: [5]

Like FreeBSD Jails, Linux VServer is a mechanism to partition Linux resources (file systems, network addresses, memory) and assign them to individual containers. It was introduced in 2001 and requires patching the Linux kernel. Experimental patches are still available, but the last stable patch was released in 2006. The patches were never released in mainline Linux.

https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

Solaris Containers Zones

In 2004, the first public beta of Solaris Containers was released that combines system resource controls and boundary separation provided by zones, which were able to leverage features like snapshots and cloning from ZFS.

SmartOS

OpenVZ

(2005)

This is an operating system-level virtualization technology for Linux which uses a patched Linux kernel for virtualization, isolation, resource management and checkpointing. The code was not released as part of the official Linux kernel.

LXC/LXD

The Linux mainline kernel added cgroups in the year 2007. In 2008 LXC was the first implementation of container management on Linux without requiring any patches. It is mainly based around the already mentioned concepts of cgroups and namespaces.

The Linux Container Daemon (LXD), pronounced “lex-de”, is a layer on top of LXC. It is developted by Canonical and first released in (October 2015). In contrast to Docker, the focus is not on containerising a single application but having a full operating system running inside.

In LXD the entry point for the container is not an application like in Docker but a full operating system init system like systemd [27]. The focus of LXD is to have a full OS running inside. LXD aims to be used in a cloud environment as a replacement for full system virtualization solutions. To manage the containers it exposes a secure REST API to allow management to be done through — for example — cloud orchestration software like Canonical’s own Juju [28], Kubernetes or OpenStack. LXD does not offer any network or storage management leaving that should be done by the orchestration on top. On the other hand it it offers life migration of containers to another host. Which is essentially important enable better dynamic load distribution in a cloud infrastructure. http://www.ubuntu.com/cloud/lxd (stateful snapshots, applications with slow startup time can be stored in the already started process. This is useful if an additional instance of an application has to be started up dynamically)

Docker

With the release of Docker in 2013 the container mechanisms had its break-through and Docker is still the most popular container solution. One of the philosophies behind Docker’s success was to hide complexity behind simplicity. This is especially true for the simplicity of the Command Line Interface (CLI) of Docker that significantly simplified initialisation and execution of previously complex task of container management.

Docker is not a standalone container runtime but consists of many different software components with different purpose. Many of these components have been extracted in order to collaborate with other projects and support further development. Some of the offspring of Docker will be explained in the following sections.

Docker uses a central daemon that handles the various running containerized processes and allows configuration by the user. Together with the configuration interface this is called the Docker Engine [29].

Initially Docker used LXC as its container runtime and later developed its own runtime libcontainer which was later extracted and donated to the Open Container Initiative (OCI) and is now know as runC [30].

Docker had frequent API changes which was difficult to integrate in other systems which favor a stable API. This lead to the development of containerd as a standalone daemon with stable API which was donated as well to the OCI. Later Docker engine also used containerd as shown in Figure 4.1. containerd uses runC to execute the container.

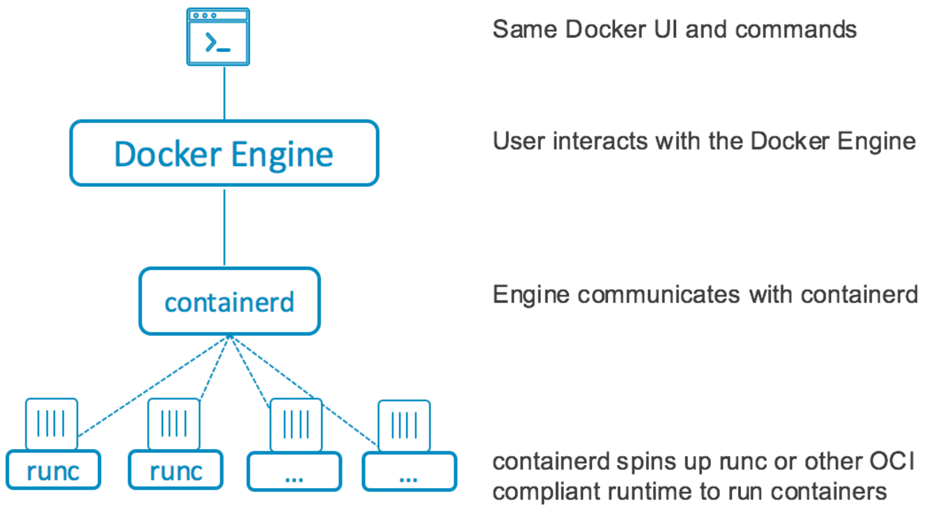
Same Docker UI and commands
Docker Engine
User interacts with the Docker Engine
containerd
Engine communicates with containerd
runc
runc
...
...
containerd spins up runc or other OCI compliant runtime to run containers

Figure 4.1: The Docker Engine architecture [31]

The docker architecture is designed around the concept that a single container only runs a single process. For example a container runs the web server, another container runs the database and yet another container runs a caching proxy in front of the service. This type of container is called an application container. Many of these application containers images are available as ready-to-use images that can be downloaded from a registry. Once the images is started it is referred to as a container.

Docker containers and images use a layered approach where each layer contains a change set to the layer underneath [32]. As shown in Figure **??** an image consists of various read-only layers and the execution of current container is done in an additional read-write layer on top of the image layers. With this layered concept if an image is executed multiple concurrent or sequential times the images layers are only stored on the system once. The same is true for image sub-layers where for example multiple layers can inherit a base layer provided by Ubuntu.

In Docker the life-time of a container matches the life-time of the process that is executed inside. If the process terminates, the container terminates with it. If a Docker container is deleted all changes made inside the container are deleted as well. In order to store the changes in a container a snapshot of the current state can be taken an stored as a new image that can again be started up at that state. Also directories from outside the container can be mapped inside the container to store consistent changes outside the container.

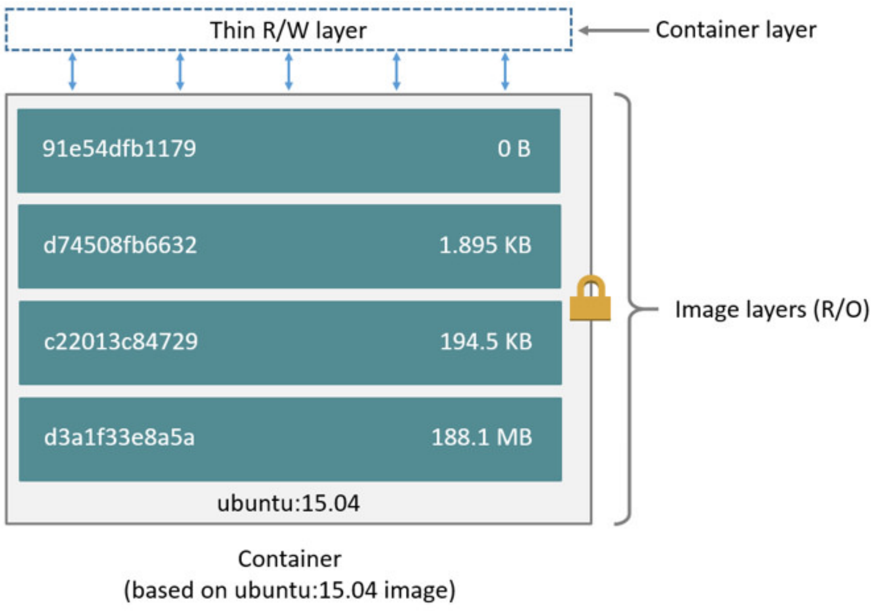

Figure 4.2: Docker Container Layers

Docker images can be built using a Dockerfile that starts off at a specified image and specifies instructions to be executed inside the container before the resulting image is saved. With this method images can be produced automatically at for example a new release of the application packaged inside the Docker image.

Docker supports multiple network drivers [33] which allow networking interfaces to act as *bridge* where the virtual docker network is separated from the host network; *host* where the container has direct access to the hosts network interface; *overlay* where a virtual network is created between multiple hosts that run the Docker daemon; *macvlan* where each container gets a unique mac address on the host's network and Docker even allows third-party networking plugins to be used.

TODO: - orchestration

rkt

rkt [34] (pronounced "rocket") is an open-source container engine developed by CoreOS project. rkt took security into consideration from the early design. rkt offers encryption and signing of images [35]. In addition to rkt's native image format it can execute Docker images.

Same as Docker rkt focuses on application containers where the ideally only a single process is executed inside a container. rkt does not have a centralized daemon for configuration but integrates in systemd to start containers directly under the Linux init process. This eliminates the single point-of-failure of having a centralized daemon like Docker. Figure 4.3 shows the execution hierarchy of rkt compared to Docker.

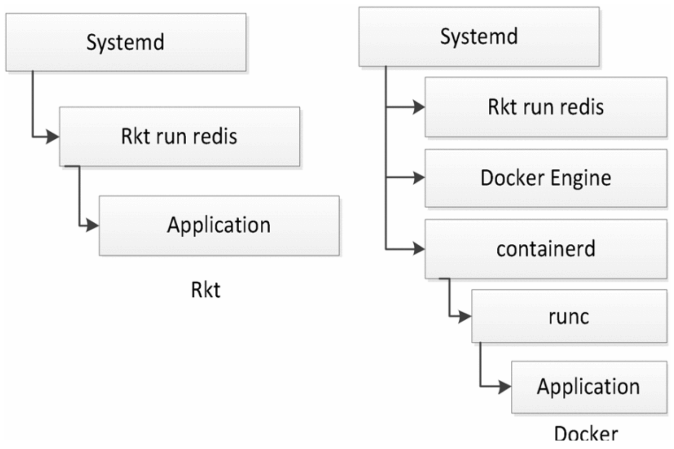

Figure 4.3: Execution hierarchy Docker vs. rkt [36]

rkt developed and uses the App Container standard appC [37] which was later included in the OCI standards. The projects defines in [38] the standards compared to the OCI standard as follows: "*The App Container Image format (ACI) is maps more or less directly to the OCI Image Format Specification, with the exception of signing and dependencies. The App Container Executor (ACE) specification is related conceptually to the OCI Runtime Specification, with the notable distinctions that the latter does not support pods and generally operates at a lower level of specification. App Container Image Discovery does not yet have an equivalent specification in the OCI project.*"

appC uses the concept of pods as the basic unit of execution. "*A pod is a grouping of one or more app images (ACIs), with some additional metadata optionally applied to the pod as a whole.*"[39] This allows configuration of — for example — resource constraints and networking for multiple containers from a single point. The concept of pods in rkt is identical to the concept of pods in Kubernetes [40] which allows a good integration and rkt became the first non-Docker runtime to be supported in Kubernetes [41].

The rkt execution workflow as shown in Figure 4.4 describes three stages. After the pod is started from various sources like the command line, systemd or Kubernetes the container runtime is set up in stage 1. Different execution environments are available in rkt. The default options are:

- **systemd-nspawnd** the default behavior that uses the cgroups and namespaces invoked from systemd.
- **fly** executes the process in a chroot without further isolation
- **KVM** uses a fully virtualized environment

Other options for stage 1 are available as plugins. In stage 2 the process inside the

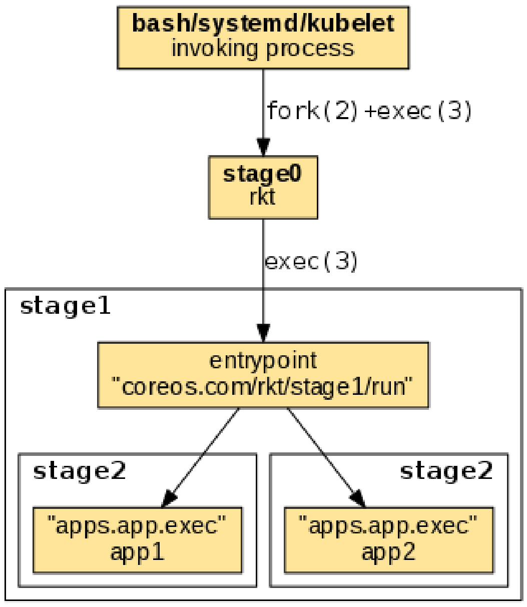

Figure 4.4: rkt execution workflow [42]

container is launched.

rkt is currently changing it's internal architecture to be compliant to the OCI standard [40] and is currently developing the integration of runC as a stage 2 execution environment. The current status rkt's OCI transition can be seen at [43].

To build rkt images scripts are available that are similar to Dockerfiles.

rkt uses systemd-journald for logging which integrates directly in the logging infrastructure of the host system.

Although the benefits of rkt it seems that developers have stopped actively developing rkt when CoreOS was bought by Red Hat in Spring 2018. The last GitHub commit on rkt was in May 2018.

Singularity

[44]

Bocker [45] is an minimalistic experimental implementation of some of Docker's functionality. It is written in bash and has around 100 lines of code. It uses the btrfs filesystem to create snapshots and directly uses a cgroups, namespaces and a chroot environment to execute the process in a containerised environment.

This minimalistic implementation shows that the container functionality is well supported in the Linux kernel allowing more sophisticated functionality to be implemented for specific use cases on top of it.

CRI-O

CRI-O [46] is a lightweight and simple container runtime developed by Red Hat as a minimal container runtime for Kubernetes which was released in October 2017. “CRI-O is designed to be simpler than other solutions, following the Unix philosophy of doing one thing and doing it well, with reusable components.” [47]. The goal for CRI-O is to have maximum compatibility with Kubernetes and provide a stable interface.

“CRI-O is compatible with the CRI (runtime) specification and the OCI and Docker image formats. It can also verify image GPG signatures. It uses the CNI package for networking and supports CNI plugins” [17]. CRI-O can also run Kata Containers.

CRI-O uses runC at it’s underlying container runtime and uses some other open libraries to container management. The new daemon conmon integrates well with systemd. It is possible to restart parts of CRI-O without having to stop other containers which is the case for Docker.

Kata Containers

Kata containers [48] is an OCI-compliant container runtime that executes containers within QEMU based virtual machines.

Kata Container originated as a merge of the two projects Hyper runV [49] and Intel Clear Containers [50] which focused on securing the runtime environment that containers execute in by executing the container inside a Kernel Virtual Machine (KVM) runtime [51].

Kata containers is developed under the OpenStack foundation and provide a virtualized isolation layer to make container execution more secure [52]. The project is supported by several companies to accelerate development and hardware support.

Kata Containers can be integrated into Docker’s containerd and into Kubernetes with Kata containers OCI-compatable runtime or directly into the CRI layer with Frakti [53]. Figure 4.5 shows the different integration options of Kata containers into Kubernetes.[1] Figure 4.6 shows how Kata containers can be executed in Kubernetes under CRI-O alongside other OCI compliant containers.

[1]This is a good representation of the current development trend towards open standardization compliance on all layers.

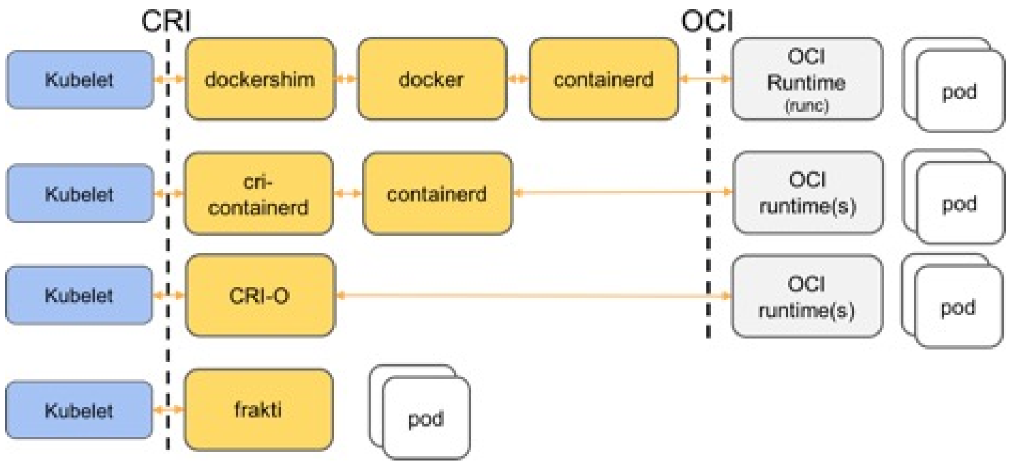

Figure 4.5: Kata Container intergration into Kubernets [54]

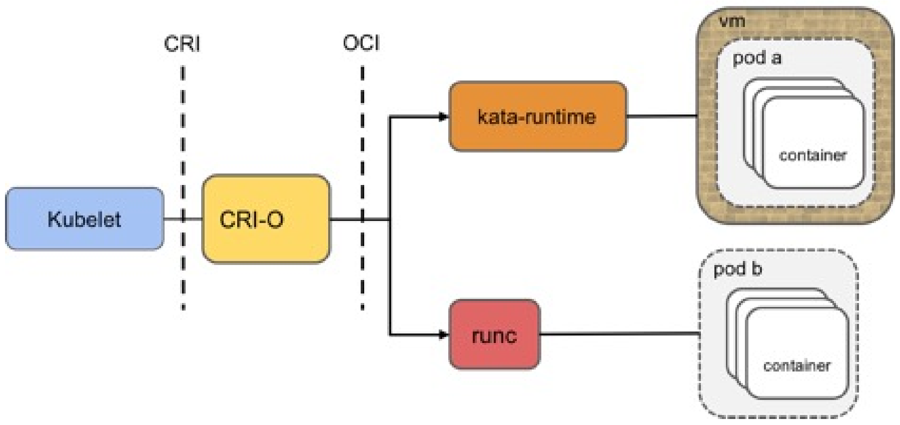

Figure 4.6: Kata Container intergration into Kubernets with CRI-O [54]

Nabla Containers

Nabla containers [55] is a project by IBM and also targets the security problem of running containers on a single Linux kernel. Instead of using virtualization it increases security while sustaining performance "*by relying on unikernel (library operating systems) techniques in conjunction with the Solo 5 middleware.*" [56] It uses only 9 kernel system calls while others are blocked with a seccomp policy. By limiting the number of system calls the attack vector is significantly reduced. Nable can be used as a runtime in Docker but the Github website lists a number of known limitations [57] like not being able to support dynamically loaded libraries, fork() or only supporting Nabla based images.

gVisor

Google's gVisor [58] describes itself as a user-space kernel which it rewrites the Linux system call interface in Go.

https://kubedex.com/kubernetes-container-runtimes/

Firecracker MicroVM

Firecracker MicroVM [59] is a minimalistic virtual machine developed by Amazon as a faster replacement for QEMU. Firecracker only emulates 4 devices for networking, block devices, serial console and a 1-button keyboard controller used only to stop the microVM. This minimalistic design enables a startup time of less than 125ms.

The development of this lightweight VM goes along with the current industry trend of a combination of containers and VMs. An integration with containerd is currently under development [60] and Kata Containers integrates Firecracker with their current beta release [61].

5 Comparison

Table of comparison:

- App/System Container
- maintained
- OCI conform
- Resource Quotas
- Live migration
- Stateful snapshots
- unprivileged execution
- Orchestration

6 Conclusion

References

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, 2015, pp. 171–172.

[2] E. Casalicchio and V. Perciballi, *Measuring Docker Performance: What a Mess !* 2017.

[3] X. Xie, P. Wang, and Q. Wang, “The Performance Comparison of Native and Containers for the Cloud,” in *2018 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 2018, pp. 378–381.

[4] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, “KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing,” in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2015, pp. 1–8.

[5] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, “Security of OS-level virtualization technologies,” in *Nordic Conference on Secure IT Systems*, 2014, pp. 77–93.

[6] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, “A comparison of virtualization technologies for HPC,” in *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, 2008, pp. 861–868.

[7] S. Wessel, M. Huber, F. Stumpf, and C. Eckert, “Improving mobile device security with operating system-level virtualization,” *Computers & Security*, vol. 52, pp. 207–220, Jul. 2015.

[8] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, “A Measurement Study on Linux Container Security: Attacks and Countermeasures,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[9] X. M. Aguilera, C. Otero, M. Ridley, and D. Elliott, “Managed Containers: A Framework for Resilient Containerized Mission Critical Systems,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 946–949.

[10] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, and K. N. B. S. Murthy, “Docker container security via heuristics-based multilateral security-conceptual and pragmatic study,” in *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, 2016, pp. 1–14.

[11] “Measuring container security.” https://lwn.net/Articles/773976/.

[12] “Containers from user space [LWN.Net].” https://lwn.net/Articles/745820/.

[13] “Namespaces article index [LWN.Net].” https://lwn.net/Articles/766124/.

[14] “Understanding the new control groups API.” https://lwn.net/Articles/679786/.

[15] “Open Containers Initiative.” https://www.opencontainers.org/.

[16] “Cloud Native Computing Foundation,” *Cloud Native Computing Foundation.* https://www.cncf.io/.

[17] “Demystifying container runtimes.” https://lwn.net/Articles/741897/.

[18] “OCI Image Format.” Open Container Initiative, Jan-2019.

[19] “OCI Runtime Specification.” Open Container Initiative, Jan-2019.

[20] “Container Network Interface: Networking for Linux containers.” CNI, Jan-2019.

[21] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, 2005.

[22] W. M. P. van der Aalst, “Orchestration,” in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 2004–2005.

[23] “OpenStack: Build the future of Open Infrastructure.” https://www.openstack.org/.

[24] “Kubernetes: Production-Grade Container Orchestration.” https://kubernetes.io/.

[25] “Cloud Foundry: Open Source Cloud Application Platform,” *Cloud Foundry.* https://www.cloudfoundry.org/.

[26] “Apache Mesos,” *Apache Mesos.* http://mesos.apache.org/.

[27] “Systemd.” https://freedesktop.org/wiki/Software/systemd/.

[28] “Jujucharms | Juju.” https://jujucharms.com/.

[29] P. Eder, “An infrastructure agnostic application deployment framework for the Internet of things,” PhD thesis, Wien, 2017.

[30] “runC: CLI tool for spawning and running containers according to the OCI specification: Opencontainers/runc.” Open Container Initiative, Jan-2019.

[31] “Docker 1.11: The first runtime built on containerd and based on OCI technology,” *Docker Blog.* https://blog.docker.com/2016/04/docker-engine-1-11-runc/, Apr-2016.

[32] “About storage drivers,” *Docker Documentation.* https://docs.docker.com/storage/storagedriver/, Jan-2019.

[33] “Docker Networking Documentation,” *Docker Documentation.* https://docs.docker.com/network/, Jan-2019.

[34] “CoreOS rkt.” https://coreos.com/rkt/.

[35] "CoreOS rkt Image Signing and Verification Guide." https://coreos.com/rkt/docs/latest/signing-and-verification-guide.html.

[36] 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)*, 2017, pp. 2137–2141.

[37] "App Container Specification and Tooling." App Container, Jan-2019.

[38] "App Container Specification and Tooling. README." App Container, Jan-2019.

[39] "CoreOS rkt App Container Basics." https://coreos.com/rkt/docs/latest/app-container.html.

[40] "What Kubernetes users should know about the rkt container engine | CoreOS." https://coreos.com/blog/rkt-and-kubernetes.html.

[41] "Rktnetes brings rkt container engine to Kubernetes." https://kubernetes.io/blog/2016/07/rktnetes-brings-rkt-container-engine-to-kubernetes/.

[42] "CoreOS rkt Architecture Documentation." https://coreos.com/rkt/docs/latest/devel/architecture.html.

[43] "CoreOS rkt OCI native support | current project state." rkt, Jan-2019.

[44] "Singularity," *Sylabs.io.* https://www.sylabs.io/singularity/.

[45] P. Wilmott, "Docker implemented in around 100 lines of bash. Contribute to p8952/bocker development by creating an account on GitHub." https://github.com/p8952/bocker, Jan-2019.

[46] "Cri-o." https://cri-o.io/.

[47] "KubeCon + CloudNativeCon North America 2017: CRI-O: All the Runtime Kubernetes Needs,..." https://kccncna17.sched.com/event/CU6T/cri-o-all-the-runtime-kubernetes-needs-and-nothing-more-mrunal-patel-red-hat.

[48] "Kata Containers - The speed of containers, the security of VMs." https://katacontainers.io/.

[49] "Hyper - Make VM run like Container." https://www.hypercontainer.io/.

[50] "Intel Clear Containers." Clearcontainers, Jan-2019.

[51] "Intel, Hyper.Sh merge tech into Kata Containers," *ICT Monitor Worldwide*, Dec. 2017.

[52] "OpenStack Boosts Container Security With Kata Containers 1.0," *ICT Monitor Worldwide*, May 2018.

[53] "The hypervisor-based container runtime for Kubernetes." Kubernetes, Jan-2019.

[54] “Kata Containers - Why Kata Containers doesn’t replace Kubernetes: A Kata Containers explainer.” https://katacontainers.io/posts/why-kata-containers-doesnt-replace-kubernetes/.

[55] “Nabla containers: A new approach to container isolation,” *Nabla Containers.* https://nabla-containers.github.io/.

[56] “IBM attempts to graft VM security onto container flexibility,” *ICT Monitor Worldwide*, Jul. 2018.

[57] “OCI-interfacing Container runtime for Nabla Containers : Nabla-containers/runnc.” Nabla Containers, Jan-2019.

[58] “Container Runtime Sandbox. Contribute to google/gvisor development by creating an account on GitHub.” Google, Jan-2019.

[59] “Firecracker MicroVM.” https://firecracker-microvm.github.io/.

[60] “Firecracker-containerd enables containerd to manage containers as Firecracker microVMs.” firecracker-microvm, Jan-2019.

[61] “Kata Containers documentation | Initial release of Kata Containers with Firecracker support.” Kata Containers, Jan-2019.