# A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing

CHRISTIAN BARGMANN and MARINA TROPMANN-FRICK, Hamburg University of Applied Sciences

Container virtualization has become the tool of choice for running isolated applications in cloud environments. Linux-Containers virtualize at the operating system level, with multiple containers running atop the operating system kernel directly. Therefore, threats to one container are potentially threats to many others. Especially for PaaS and Serverless providers, the secure execution of untrusted workloads on their platform in order to mitigate software vulnerabilities from spreading has high priority. Containers face a variety of different threats, vulnerabilities and historical weaknesses that need to be considered and defended against. This paper presents current approaches to securing container workloads. gVisor, Kata Containers and Firecracker are presented and compared with each other. Although sandbox containers have different attack surfaces such as the container daemon process, network, or storage, this paper focuses on the Linux kernel itself as a vulnerability in sandbox containers and examines how each approach implements protection.

## 1. INTRODUCTION

Due to their flexibility and scalability, containers have gained popularity in recent years. Despite their name, containers are not completely closed. The guest system of each container uses the same host operating system and its services. This reduces overhead and improves performance, but can cause potential security or interoperability problems. The degree of isolation provided by the Linux kernel combines process isolation with namespaces. This concept works well, but does not close all possible security gaps due to construction, so malware can break out and gain access to the host directly or other containers sharing the same host system kernel. This is particularly critical in multi-tenant scenarios where multiple clients run containerized workloads in a shared cloud environment and the containers can run as isolated processes belonging to different clients on the same shared host. The isolation of runtime environments is also becoming increasingly important for Serverless Computing where the cloud provider provides a runtime environment for executing server-side logic which can be potentially harmful to other runtime environments running on a shared host system.

The Linux kernel itself provides a number of mechanisms to isolate processes from each other and restrict system calls. Although a high level of isolation can be achieved with these mechanisms, the shared kernel remains a security risk [MITRE Corporation 2019] [Heise Online 2019]. The rise of container virtualization led to the discussion whether and how containers could inherit classic hard-

Author's address: Christian Bargmann, University of Applied Sciences Hamburg, Berliner Tor 5, 20099 Hamburg, Germany, email: christian.bargmann@haw-hamburg.de and Marina Tropmann-Frick, University of Applied Sciences Hamburg, Berliner Tor 5, 20099 Hamburg, Germany, email: marina.tropmann-Frick@haw-hamburg.de.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

#### 1:2 • C. Bargmann and M. Tropmann-Frick

ware virtualization. Time has shown that container virtualization and hardware virtualization are not opposed to each other. Rather, the technologies are complementary, especially when the focus is on security. Hardware virtualization is able to isolate computing resources while Containers focus on isolating processes instead of resources. A fully virtualized system gets its own set of resources assigned to it and minimal sharing takes place. Virtual Machines (VMs) offer a comparatively high degree of isolation and thus security for workload isolation. However, the replication of a hardware environment including the operating system leads to an overhead and the start up time of a virtual machine is significantly slower than with container virtualization. Compared to VMs, containers offer a significantly less isolation level but can be deployed immediately without complex installation processes, therefore container virtualization is not suitable for implementing security concepts [Docker 2016].

To secure container workloads, multi-layered security is necessary and issues from both approaches are needed. On the one hand, the performance of container deployment and on the other hand the isolation of the host system, which a virtual machine offers. In this paper, we have reviewed which approaches are currently being driven forward for the implementation of an additional security layer. We have addressed the question of whether these approaches can be distinguished from each other on an abstract level. First, scenarios for container isolation and currently used security mechanisms for containers are presented and the question why an additional security layer is necessary is answered. On the basis of this, *gVisor*, *Kata Containers* and *Firecracker* are presented. In conclusion, the approaches are compared with each other and an overview of the current development approaches is given.

## 2. BACKGROUND

This section covers backgrounds for container isolation. It starts with use cases and scenarios for container isolation. It then describes the status of containerized runtime environment security and the need for layered security.

#### 2.1 Use Cases & Scenarios

2.1.1 Sandbox untrusted/vulnerable code. Containers are ideal for isolation on application level. There are use cases where a user wants to run an application with a number of previous vulnerabilities on a host machine. There is a risk that an untrusted attacker may be aware of these vulnerabilities and new vulnerabilities may be easily found. To protect the host system, the use of a highly isolated container environment for the application is useful. Another use case would be a scenario in which code is to be executed on a host machine that cannot be trusted. This may be relevant if access to the source code is not given. The lack of transparency can lead to a security risk. Container isolation is also suitable for protection here.

2.1.2 *Defense in depth scenarios.* For the processing of sensitive workloads and data, defense in depth is of high relevance. In use cases where the code is trusted, it may still be important to protect it from possible attacks. This can be especially important for medical or financial services.

2.1.3 Serverless Computing Scenarios. Serverless Computing is a cloud computing paradigm where the cloud provider provides a runtime environment for executing server-side logic. Server-side logic is implemented by means of short-lived, stateless functions called *serverless functions*. The execution environment for these functions is called *Function-as-a-Service* (FaaS) [Spillner 2018]. Serverless Computing is the ideal of an event-based application architecture - an application architecture that emerges from the current trend in container and microservices architectures [Baldini et al. 2017]. As a result, functions are executed in a serverless application in response to the triggering of certain events. If a serverless function is triggered, the cloud provider dynamically allocates the computing capacity required to execute the function. After the execution has been completed, the computing capacities are

released again. In this scenario, the cloud provider does not know the code executed by users in the runtime environment provided. This code can be potentially harmful to its environment. For serverless platform providers, it is therefore of great interest to isolate the individual runtime environments of users strongly.

2.1.4 *Multi-Tenancy Scenarios.* There are more than one type of multi-tenancy, and multiple definitions of a tenant. Depending on the tenant scenario, the required degree of isolation may vary. Assuming a multi-tenancy model in which multiple untrusted users (Tenants) may execute untrusted code in a shared cloud environment, thus code deployed by tenants can potentially be executed on the same host machine on which code from other tenants is simultaneously executed. Compared to Serverless Computing, this means that the cloud provider makes the compute resources available to the user, but the user himself can use his own runtime environments, e.g. in the form of his own containers. The cloud provider does not know the user's application and runtime environment in advance. Unlike multi-tenant scenarios, where users of a cloud environment trust each other and also trust the executed code, e.g. within a company with different departments as tenants sharing a compute cluster, the described model requires a strict isolation of resources belonging to different clients.

#### 2.2 Current Mechanisms used for Container Isolation

Today many mechanisms based on Linux technologies are used to isolate container runtime environments.

2.2.1 AppArmor / SELinux. AppArmor / SELinux are security frameworks for Linux. As Mandatory Access Control (MAC) systems they control applications individually. For these, access rights can be defined in profiles that are finer than the general file rights. Beside the predefined ones, own profiles can be set up. The purpose of AppArmor / SELinux is to protect security-critical applications, i.e. primarily applications / processes with network access, but also office applications which could possibly compromise the system by loading infected documents [Debian man-pages 2018] [Linux man-pages project 2018d].

2.2.2 *Capabilities.* The basic idea behind Linux capabilities is to break up the monolithic root privilege that Linux systems have had, so that smaller more specific privileges can be provided where they're required. This helps reduce the risk that by compromising a single process on a host an attacker is able to fully compromise it [Linux man-pages project 2018a].

2.2.3 *Namespaces*. An instance of a namespace defines a new environment that virtualizes certain operational resources such as process users, the file system, or the network in a very lightweight way. They abstract the respective global system resource in such a way that it looks like an independent isolated instance for a process within the respective namespace. Lightweight means that no hypervisor is needed, but the processes simply do not see the other instances of a resource, but they continue to run in the same kernel [Linux man-pages project 2018b].

2.2.4 *User Permissions.* For the isolation of container runtime environments the Linux own user system is used. Discretionary Access Control (DAC) is restricting access to objects based on the identity of subjects and groups to which a certain subject belongs to [Rusling 1999].

2.2.5 *cgroups*. Using "control groups" a user can combine several processes into one group. The user can then provide these processes and all child processes with parameters for specific subsystems. For example, a subsystem is a resource controller that manages the available memory. cgroups can be used to define limits that must not be exceeded by resources, such as a lot of memory. Users can give priority to some resources over others, such as more frequent processing by the CPU. cgroups can also

#### 1:4 • C. Bargmann and M. Tropmann-Frick

measure how many resources have been consumed. With the help of cgroups, entire groups of processes can also be controlled, e.g. they stop or continue to run [Linux man-pages project 2018c].

2.2.6 *seccomp-bpf.* "Secure computing mode" is a simple and effective sandboxing tool. It allows the user to attach a system call filter to a process and all its descendants, thus reducing the attack surface of the kernel. Seccomp filters are expressed in Berkeley Packet Filter (BPF) format [Mozilla Foundation 2018] [Linux man-pages project 2018c].

## 2.3 Why is Multi-Layer Security is needed for Container Isolation?

Attacks aimed at breaking out of a containerized runtime environment are particularly critical and require a lot of attention. There is always a risk that processes will break out. If this happens, not only the individual container instance is affected, but also all other processes that also run on the host system.

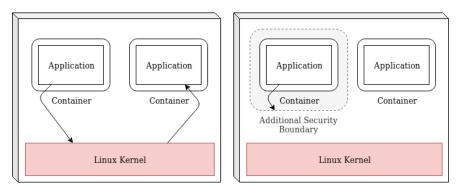


Fig. 1. Multiple layers are needed to form a trust boundary around untrusted code. Containerized applications should have an additional security layer above the container boundary.

The primary target of such an outbreak is the operating system kernel. If an attacker controls it, he has the entire system in his hands. Container technologies like *Docker* [Docker Inc. 2019] or *Rkt* [CoreOS 2019] change much faster than the underlying kernel technologies, so newly discovered vulnerabilities in the kernel can affect a number of different kernel versions. The presented security mechanisms are excellent to ensure defense in depth, but the Linux kernel itself remains the biggest vulnerability [Bettini 2013]. It is sufficient for an attacker to exploit a known vulnerability in the Linux kernel to compromise all security mechanisms.

Using a container runtime is therefore not sufficient for isolating applications. To safely isolate applications from each other, one or more additional security layers beyond the boundaries of the container runtime environment are required as shown in F4906384219579641ig. 1.

## 3. SECURING CONTAINER WORKLOADS

This section introduces *gVisor*, *Kata Containers*, and *Firecracker* as approaches to isolating container workloads. It is being discussed whether the approaches meet the requirements of multi-layer security.

## 3.1 gVisor

On May 2, 2018 Google released their container runtime sandbox gVisor [Lacasse 2018]. gVisor is a user-space kernel specifically for containers, written in the Go programming language. The software implements a substantial part of the Linux system interface and thus emulates the Linux system call

API in a userspace process. gVisor draws a boundary between the application running in a container and the host kernel by using the runtime environment of the Open Container Initiative (OCI) called *runsc*.

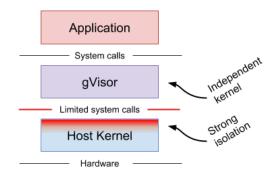


Fig. 2. gVisor creates a dedicated buffer layer for each container. The new layer running as a user space process that intercepts the container's system calls [gVisor 2018].

Applications that run in containers on the host system access system resources in the same way that applications that are not in containers do. System calls are sent directly to the host kernel. Even without gVisor, the kernel limits the access that a containerized application can make to the system resources, but such a kernel still offers a large attack surface. gVisor does not use hardware virtualization to isolate containers, i.e. containers are not packed into virtual machines to provide an additional security boundary and therefore no virtualized hardware is passed to a guest kernel via a Virtual Machine Manager. Instead gVisor uses a kernel that is active as a normal, non-privileged process and supports most Linux system calls. gVisor intercepts application system calls and acts as the guest kernel, without the need for translation through virtualized hardware. Just like in a normal virtual machine, an application running in a gVisor sandbox gets its own kernel and a selection of virtualized devices isolated from the host and other sandboxes. Unlike containers packed in virtual machines, gVisor is more lightweight, but provides a similar level of isolation and therefore security [gVisor 2018]. A major disadvantage of gVisor is that at the time of this paper not all system calls of the Linux System Call API are implemented, which is why not all applications run with gVisor yet. Also, applications that are sandboxed with gVisor create a higher per-system call overhead.

With gVisor, multi-layer security can be implemented. The emulated kernel in the userland represents the first security boundary that an attacker must overcome. The second security boundary is the process isolation mechanisms of the host kernel. Kernel features such as secomp filters can be used to provide better isolation between host and gVisor kernels. However, they require the user to create a predefined whitelist of system calls. Due to the additional abstraction layer offered by gVisor, filter rules can be used much more universally. Creating system call filter rules for applications that are not known in advance is often a difficult task. The application of filter rules on gVisor as an interface between host and application is easier to configure.

## 3.2 Kata Containers

Kata Containers version 1.0 was released on May 22, 2018 [Bertucio 2018]. The technical foundations of Kata Containers are two other projects that have been in existence for quite some time. The first are the Clear Containers, which Intel launched in 2015 [Clear Linux Project 2019]. Number two is runV from Hyper [Hyper 2018]. Kata Containers is the fusion of both initiatives. The objective of

#### 1:6 • C. Bargmann and M. Tropmann-Frick

Kata Containers is to develop a standard implementation of lightweight virtual machines that feel and function like containers, but offer the workload isolation and security benefits of hardware virtualization. It connects both worlds and lets containers run inside slim virtual machines to create a sandbox environment.

The architecture of Kata Containers consists of six elements [Kata Containers 2019a]. The isolation and insertion of another kernel is done by virtualization software using a minimal operating system. A slimmed down version of QEMU [QEMU 2018] is used on which Clear Linux runs. Clear Linux provides the kernel and is optimized for running containers. QEMU and Clear Linux together form the Kata framework. The remaining four components are responsible for the container structure, all developed in the Go programming language and available under Apache license. The runtime environment corresponds to the specifications of the Open Container Initiative (OCI). It can be installed and used parallel to the standard container runtime runc. The Kata runtime and the so-called shim form the interface to container tools such as Kubernetes or OpenStack. The shim as a kind of auxiliary construct enables the classic container tools to get an interior view of the processes of the virtual machines. Without this help, tools like Kubernetes would not be able to look behind the scenes of QEMU-Lite. Kata Containers uses the proxy and agent to connect to the virtual machine. They communicate via grpc using QEMU's serial port. The agent is running as a process in the QEMU instance. It acts as a supervisor for managing containers and processes running within those containers. There is one agent per virtual machine. The proxy is the counterpart on the host side and is used for communication between the hypervisor on the host system and the virtual machine running containers. Also here the Kata runtime environment starts exactly one proxy per QEMU instance. Kata Containers combines the properties of virtual machines and containers. Virtualization protects the kernel of the host system. However, a disadvantage is that the high number of components required for managing and communicating containers within the virtual machine is another potential attack surface. The kernel and operating system used by Clear Linux must also be kept up to date in order not to be affected by possible security vulnerabilities.

With Kata Containers multi-layer security can be implemented. The hypervisor provides an additional security boundary and thus increases the security of the sandbox. This makes it more difficult for a harmful process to break out. A sandboxed process must first convince the guest kernel to trigger a malicious virtual machine exit. This also significantly reduces the attack surface at the same time. Afterwards the hypervisor has to be exploited to break out of the sandbox and gain control over the host system. The degree of isolation by the first security boundary is reduced with the number of rights a container receives within the virtual machine. If the user is able to start his own guest systems in the virtual machine, the first security boundary can be omitted.

## 3.3 Firecracker

On November 27, 2018, Amazon Web Services introduced Firecracker, a virtual machine manager that enables secure, multi-tenant, minimal overhead execution of container and function workloads, as an open source project [Arun Gupta 2018]. Amazon's own serverless computing platform Lambda uses Firecracker as a foundation for deploying and operating sandboxes that run untrusted code from customers. Firecracker boots a minimal kernel config without relying on an emulated bios and without a complete device model [Firecracker 2018b].

The main component of Firecracker is its own hypervisor which is directly based on Linux Kernelbased Virtual Machine (KVM) capabilities. The foundation for this is the Crosvm project [Chromium 2018] by Google's Chromium team, written in Rust programming language and used for Linux applications in Chrome OS. Firecracker is also completely written in Rust. The Firecracker application utilizes the hypervisor to set up and run minimal virtual machines. Firecracker itself is started by an

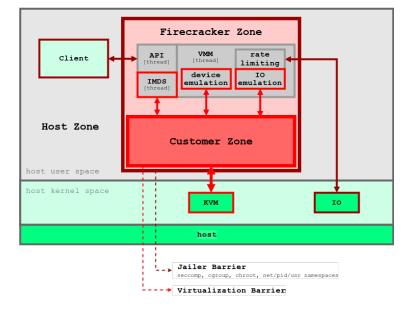


Fig. 3. Firecracker mitigates security risks by defining different nested trust zones [Firecracker 2018b].

application called Jailer, which provides and configures system resources such as cgroups. The Jailer also manages permissions, or withdraws them from the system as soon as they are no longer needed. In addition, Jailer sets seccomp-bpf filters to restrict guest code access [Firecracker 2018c]. Kernel namespaces are used as an additional barrier. The communication with Firecracker runs via a Restful API, with which the virtual machines can finally be started and controlled. By default, the virtual machines use a virtualized CPU core and 128 MB RAM, which can be adjusted via the API endpoints. In addition, the VMs only use a Virtio-provided network device and a block device, both of which have rate limiting. There is also a serial console and a minimal keyboard driver to reset the virtual machines. To start a VM, users need an unpacked kernel image and a root file system. Alpine Linux is a Linux distribution based on musl and BusyBox, which focuses on security, simplicity and resource efficiency and also enjoys great popularity as a basic image for containers. At the time of this paper, Firecracker still only uses Intel hardware virtualization, but support for AMD and ARM hardware is expected to follow. Work is also underway to integrate Firecracker into existing container ecosystems such as Kubernetes or OpenStack [Firecracker 2018a].

Firecracker implements multi layer security by defining different nested trust zones. The jailer process sets up system resources that require elevated permissions, drops privileges, and then executes into the Firecracker binary, which then runs as an underprivileged process. In addition, seccomp-bpf filters are used to limit system calls to the guest kernel. This represents the first security boundary. In order to take over the host system, an escape from the virtual machine must be successful and thus represents the second security boundary.

#### 4. COMPARISON OF CONTAINER ISOLATION APPROACHES

Although their concrete architecture is quite different, all three projects follow a similar approach. They build an additional layer between the application to be isolated and the kernel of the host system. The main difference between the three projects presented lies in the virtualization technology used to form the isolation layer.

#### 1:8 • C. Bargmann and M. Tropmann-Frick

Kata Containers chose QEMU as the foundation for its virtualization technology using "QEMU light". A sandboxed container runs within a minimal, lightweight virtual machine. Via an internal agent process and an external shim, runtime commands and IO request are intercepted. Firecracker follows a similar approach. Again, virtual machines are used to separate sanboxed applications from host kernels. Unlike Kata Containers, Firecracker does not use QEMU, but implements its own virtual machine manager that uses Linux Kernel-Based Virtual Machine (KVM). It emulates a minimal device to achieve low latencies when starting the VM and low memory footprint on the host system. At the same time, a trusted sandbox environment is provided for an isolated application within the virtual machine. Unlike Kata Containers, Firecracker does not start any containers in the virtual machine. Google's gVisor, on the other hand, has a completely different strategy than the other two projects presented. Neither microVMs like Firecracker nor a combination of virtual machine and container like Kata Containers are used. Instead, a dedicated buffer layer is created for each individual container to provide an additional security boundary between host system and application. The layer created by gVisor runs as a process in userspace and allows the container to perform system calls without directly accessing the host kernel.

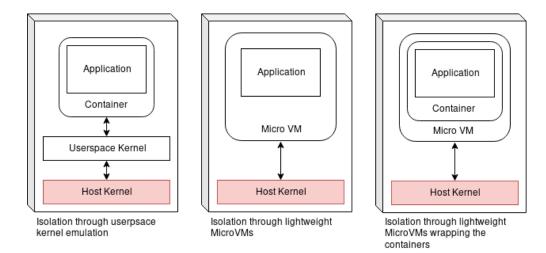


Fig. 4. Comparison of container isolation approaches.

Although there are currently several projects with different ideas for the realization of container isolation, it must be mentioned that they are all more or less experimental at the time of this paper. Even though large cloud providers such as Google or Amazon are behind some of the concepts presented, all approaches are at a very early stage of development. Although Firecracker or gVisor are used in the cloud environments of the providers, it is not certain whether the solutions are also suitable for general production environments outside the provider-specific offerings.

An important step for the productive use of sandbox environments is the compatibility to common container tools like Docker or orchestration tools like Kubernetes and OpenStack. Both Docker and Kubernetes use *containerd* as container runtime. While gVisor and Kata Containers can already be used as plugins for *containerd* and are therefore compatible with Docker [Docker 2016], this is not yet the case with Firecracker. Currently there is a project to run Firecracker as a plugin for containerd [Firecracker 2019]. At the same time Kata Containers supports the Firecracker hypervisor since the release of versionDanke 1.5 on January 16, 2019 [Kata Containers 2019b]. gVisor is also working on

the further implementation of Linux system call API, so that more applications can be sandboxed with gVisor in the future.

# 5. CONCLUSION AND FUTURE WORK

This paper presented and compared approaches to isolate container workload. Different use cases and scenarios were presented in which the secure handling of containerized applications is of high priority. Furthermore, state of the art mechanisms for the isolation of containers were investigated and the question why multi-layer security is relevant to provide a secure sandbox environment was answered. Based on this, *gVisor*, *Kata Containers* and *Firecracker* were presented and explained whether the concepts can implement multi-layer security. Finally, the concepts were compared with each other and an outlook on further development was given. In future we seek to develop a comparison criteria catalogue that allows the presented, as well as emerging approaches to be classified and differentiated on the basis of an evaluation framework.

In summary, securing container workload is becoming increasingly important. Not only for providers of big public cloud environments, but for every cloud environment with multi-client capability and the execution of untrusted code on host machines. The development of the approaches presented remains exciting and it will be shown which of the approaches will prevail for general productive use.

#### REFERENCES

- Linda Lian Arun Gupta. 2018. Announcing the Firecracker Open Source Technology: Secure and Fast microVM for Serverless Computing. (2018). https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/ [Online; accessed 2019-04-23].
- Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. abs/1706.03178 (2017), P. 13.
- Anne Bertucio. 2018. Kata Containers 1.0. (2018). https://katacontainers.io/posts/kata-containers-first-release/ [Online; accessed 2019-04-17].
- Anthony Bettini. 2013. Vulnerability exploration in docker container environments. (2013). DOI:http://dx.doi.org/10.1111/papa.12013
- Chromium. 2018. crosvm The Chrome OS Virtual Machine Monitor. (2018). https://chromium.googlesource.com/chromiumos/platform/crosvm/+/master/README.md [Online; accessed 2019-04-24].
- Clear Linux Project. 2019. Containers Clear Linux\* OS. (2019). https://clearlinux.org/containers [Online; accessed 2019-04-17].
- CoreOS. 2019. Rkt A security-minded, standards-based container engine. (2019). https://coreos.com/rkt/ [Online; accessed 2019-02-18].
- Debian man-pages. 2018. AppArmor kernel enhancement to confine programs to a limited set of resources. (2018). https://manpages.debian.org/stretch/apparmor/apparmor.7.en.html [Online; accessed 2019-03-27].
- Docker. 2016. Introduction to Container Security: Understanding the isolation properties of Docker. (2016). https://www.innovate-systems.de/downloads/docker/WP\_IntrotoContainerSecurity\_08.22.2016.pdf [Online; accessed 2019-03-21].
- Docker Inc. 2019. Docker Enterprise Container Platform. (2019). https://www.docker.com/ [Online; accessed 2019-02-14].
- Firecracker. 2018a. firecracker-containerd architecture. (2018). https://github.com/firecracker-microvm/firecracker-containerd/blob/master/docs/architecture.md [Online; accessed 2019-04-26].
- Firecracker. 2018b. Firecracker Design. (2018). https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md [Online; accessed 2019-04-24].
- Firecracker. 2018c. The Firecracker Jailer. (2018). https://github.com/firecracker-microvm/firecracker/blob/master/docs/jailer.md [Online; accessed 2019-04-26].
- Firecracker. 2019. Firecracker-Containerd. (2019). https://github.com/firecracker-microvm/firecracker-containerd [Online; accessed 2019-04-27].
- gVisor. 2018. gVisor Documentation. (2018). https://github.com/google/gvisor/blob/master/README.md [Online; accessed 2019-04-11].

#### 1:10 • C. Bargmann and M. Tropmann-Frick

- Heise Online. 2019. Sicherheitsforscher brechen aus Docker-Container aus. (2019). https://www.heise.de/security/meldung/Sicherheitsforscher-brechen-aus-Docker-Container-aus-4276108.html [Online; accessed 2019-01-15].
- Hyper. 2018. Hypercontainer: Hypervisor-agnostic Docker Runtime. (2018). https://hypercontainer.io/ [Online; accessed 2019-04-17].
- Kata Containers. 2019a. Kata Conainers Architecture Overview. (2019). https://github.com/kata-containers/documentation/ blob/master/architecture.md [Online; accessed 2019-04-18].
- Kata Containers. 2019b. Kata Containers Release 1.5.0. (2019). https://github.com/kata-containers/runtime/releases/tag/1.5.0 [Online; accessed 2019-04-27].
- Nicolas Lacasse. 2018. Open-sourcing gVisor, a sandboxed container runtime. (2018). https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime [Online; accessed 2019-04-11].
- Linux man-pages project. 2018a. capabilities overview of Linux capabilities. (2018). http://man7.org/linux/man-pages/man7/ capabilities.7.html [Online; accessed 2019-03-27].
- Linux man-pages project. 2018b. Linux Programmer's Manual: Namespaces. (2018). http://man7.org/linux/man-pages/man7/ namespaces.7.html [Online; accessed 2019-03-24].
- Linux man-pages project. 2018c. seccomp: operate on Secure Computing state of the process. (2018). http://man7.org/linux/man-pages/man2/seccomp.2.html [Online; accessed 2019-03-25].
- Linux man-pages project. 2018d. SELinux Command Line documentation. (2018). http://man7.org/linux/man-pages/man8/ selinux.8.html [Online; accessed 2019-03-28].
- MITRE Corporation. 2019. Linux Kernel: Security Vulnerabilities. (2019). https://www.cvedetails.com/vulnerability-list/vendor\_ id-33/product\_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html [Online; accessed 2019-02-17].
- Mozilla Foundation. 2018. Intro to seccomp and seccomp-bpf. (2018). https://wiki.mozilla.org/Security/Sandbox/Seccompl [Online; accessed 2019-04-02].
- QEMU. 2018. QEMU the FAST! processor emulator. (2018). https://www.qemu.org/ [Online; accessed 2019-04-27].
- David Rusling. 1999. Linux Processes. (1999). https://www.tldp.org/LDP/tlk/kernel/processes.html [Online; accessed 2019-03-22].
- J. Spillner. 2018. The Prospect of Ser-verless Scientific Computing and HPC. In High performance computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017:, Esteban Mocskos and Sergio Nesmachnow (Eds.). Number 796 in Communications in computer and information science. Springer, P. 15-16. DOI:http://dx.doi.org/10.1007/978-3-319-73353-1\_11