A Perfect Fit? – Towards Containers on Microkernels

Till Miemietz
Barkhausen Institut
Dresden, Germany

Viktor Reusch
Barkhausen Institut
Dresden, Germany

Matthias Hille
Barkhausen Institut
Dresden, Germany

Max Kurze
TU Dresden
Dresden, Germany

Adam Lackorzynksi
TU Dresden / Kernkonzept GmbH
Dresden, Germany

Michael Roitzsch
Barkhausen Institut
Dresden, Germany

Hermann Härtig
Barkhausen Institut
Dresden, Germany

Abstract

Containers are a lightweight alternative to virtual machines, building on sandboxed processes whose permissions are restricted by additional security mechanisms such as `seccomp-bpf`. However, these mechanisms increase the kernel's attack surface, thus prompting new security challenges. In this paper, we ask the question of whether a system with processes properly restricted by design enables a container infrastructure with better security posture. For instance, microkernels with capability-based access control provide container-style isolation out of the box. On the basis of real-world CVEs, we argue that this conceptual simplicity actually results in a better security posture than that typically found on monolithic systems.

We propose *Oak*, a container engine built on top of L4Re, a state-of-the-art microkernel-based operating system. For startup as well as for network microbenchmarks, containers running on L4Re exposed performance characteristics similar to that of containers on Linux. We thus conclude that building containers on microkernel is an approach worth pursuing further under both a performance and a security perspective.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Security and privacy → Virtualization and security**.

ACM Reference Format:
Till Miemietz, Viktor Reusch, Matthias Hille, Max Kurze, Adam Lackorzynksi, Michael Roitzsch, and Hermann Härtig. 2024. A Perfect Fit? – Towards Containers on Microkernels. In *10th International Workshop on Container Technologies and Container Clouds (WoC '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702637.3702957

1 Introduction

Currently, virtual machines (VMs) and containers are the two prevalent mechanisms for isolating untrusted workloads running on a shared system in the cloud. Generally speaking, containers are a more lightweight approach to isolation as they do not attempt to emulate an entire computing platform. Instead of spawning distinct operating system instances, multiple containers running on the same host share a single kernel and only expose separate instances of userland to applications. This allows them to achieve runtime performance close to that of the bare metal system [24]. Moreover, starting a container is much faster than booting a virtual machine [22]. These advantages made containers the premier choice for isolating workloads in settings such as serverless computing.

Interestingly, when looking at containers from the perspective of an OS engineer, they are only little more than hardened processes (or process groups) that are restricted in their resource usage through dedicated in-kernel mechanisms such as `cgroups` or `seccomp-bpf` [2]. These facilities however, add a lot of complexity to core OS abstractions. This additional complexity in turn harms cross-container isolation by exposing a larger in-kernel code base that is shared between distrusting containers [4, 5].

In fact, the intricacies of implementing the isolation mechanisms required for containers on monolithic OSes stem from the fact that these OS architectures are not designed with the principle of least authority (PoLA) in mind. Hence, on such systems, the implementation of containers needs to prevent applications from exercising ambient authority that they should not have in the first place. As a consequence, `seccomp-bpf` is used to retroactively restrict the of system calls (and their parameters) that are available to a container. This not only adds complexity but also introduces a slight performance penalty as the respective checks need to be carried out at runtime [20]. Even though this overhead has been reduced recently [3], it will never disappear completely.

Modern microkernel-based OS designs [15, 21] in contrast, enforce PoLA by default. On such systems, processes have to explicitly request access to any system service (drivers, file systems, etc.), instead of implicitly having the authority to use them. Also, upon granting a process access to a system resource, the administrator can directly restrict the set of possible request parameters that this process can issue to the service managing said resource. As a consequence, implementing container-style isolation on such platforms does not require additional resource constraining mechanisms or runtime checks as the OS itself provides strong compartmentalization of processes by default. Hence we argue, that on a microkernel-based OS, processes provide the same isolation properties as containers on monolithic systems.

From a security perspective, a microkernel design can significantly decrease the amount of code that mutually distrusting containers need to share since the core system that each process has to rely on is rather small. Because all other system services are implemented as separate processes in userspace, the trusted computing base (TCB) of each container is much smaller than on monolithic

WoC '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-1339-2/24/12
https://doi.org/10.1145/3702637.3702957

systems, as it does not include any system components not used by the respective container. Consequently, this leads to an overall improved security posture of the system.

However, microkernel-based OS architectures are mostly used in embedded systems today, raising the question of how well microkernel concepts scale to large machines typically used in a cloud setting. In order to answer this question and investigate the feasibility of our concept for containers on microkernels, we built a prototype container engine on top of L4Re [15], a state-of-the-art microkernel-based OS. To this extent, we demonstrate how to provide container-grade isolation, including resource restriction, accounting, and the necessary system services, on such a system. Following a discussion on the security properties of the resulting approach, we show the results of several microbenchmarks, getting first insights into the performance of containers on L4Re.

2 Background

This section explains the background of (Linux) Containers and lightweight virtual machines, as both are used for isolating workloads in the cloud. We then introduce L4Re and capabilities, an access-control mechanism used with modern microkernels.

2.1 Containers on Monolithic Operating Systems

Broadly speaking, a container is an OS mechanism that virtualizes the execution environment of applications while sharing a single kernel. Examples for such facilities can be found on a variety of operating systems such as BSD (*Jails* [19]) and Linux [13]. Even though the exact implementation differs, containers on all platforms rely on three common mechanisms with respect to security: Access to unnecessary (kernel) interfaces is denied. For necessary interfaces, the visibility of resources is restricted. Where resources have to be shared, resource accounting is enforced.

Interface restrictions: As shown by Canella et al. [10], many cloud applications only require a fraction of the large kernel interface exposed by monolithic OS designs. Container implementations usually exploit this fact to increase security by preventing a container from executing certain system calls, thus reducing the attack surface of the shared kernel. Current solutions like `seccomp-bpf` on Linux build on top of the Berkeley Packet Filter (BPF) [23] and allow the administrator to specify small filtering programs that the kernel executes upon every system call of a container to determine whether it adheres to the container's security policy.

Visibility Restrictions: Containers typically impose additional visibility restrictions on top of traditional processes. By using kernel features like Linux' *namespaces*, the administrator is able to hide certain parts of the system, such as other processes, from applications running inside a container. Namespaces further allow for implementing a limited form of virtualization.

Within the scope of containers, visibility restrictions are applied to, e.g., the file system by setting a separate root directory using the `chroot` system call. Furthermore, container implementations often virtualize network interfaces, process IDs, and mount points. Also, container-grade virtualization assures that processes running inside a container cannot communicate across container boundaries.

Resource restrictions: For reasons of accountability, operating systems also restrict the resource usage of containers using dedicated mechanisms that extend traditional resource control mechanisms like `ionice`. One prominent example of such resource constraining frameworks is Linux' `cgroups` feature. Resource restriction frameworks for containers are often capable of organizing restrictions in a hierarchical manner, allowing containers to pass on a fraction of their already constrained resource budget [8]. Furthermore, they can prioritize access to system resources, thus guaranteeing a minimum share of a certain resource to each container.

Beyond the core mechanism, most implementations of containers also provide an ecosystem for managing containerized applications. For instance, on Linux, a *high-level container runtime* like `containerd` manages container images and the resources that a container requests for execution such as access to the network. The Open Container Initiative (OCI) provides standards for the container runtime interface and container images, which allows OCI-compliant container managers to interact with any OCI-compliant *low-level container runtime*. Low-level container runtimes like `runC` are responsible for setting up a container by configuring the lightweight virtualization mechanisms provided by the OS.

2.2 Lightweight Virtual Machines

Virtual machines (VMs) provide a higher degree of isolation than containers, as distrusting applications do not share a single kernel instance. Lightweight virtual machine implementations like Firecracker [6] try to overcome traditional performance shortcomings of VMs by optimizing virtual machine monitors for use cases like serverless computing, thus providing performance close to that of containers [7]. Furthermore, the use of unikernels as a VM guest [22] is a way to increase the performance of VMs, as unikernels run in a single privilege level, saving context switches during execution.

2.3 A Primer on the L4Re Microkernel OS

Being a microkernel, the L4Re kernel only implements functionality that cannot be realized in userspace, such as page table manipulation or mechanisms for inter-process communication (IPC). The L4Re microkernel represents each of the abstractions that it offers to userland processes as *kernel objects*. Examples for kernel objects are threads and *IPC gates* which represent an IPC channel to a process.

Every other component of the operating system such as the memory management or device drivers run as processes in userspace (dubbed *tasks* in L4Re). During bootup the L4Re microkernel starts a *root task* called `moe`. `Moe` provides the basic system abstractions of L4Re such as memory allocation or access to the read-only boot file system to other applications. `Moe` allows to define quotas for resource allocations, thus being able to constrain the resource consumption of application tasks.

2.4 Capabilities

L4Re, like most modern microkernel-based operating systems, uses capabilities [11] for implementing access control. A capability is an unforgeable token of authority, which grants the possessing process the power of carrying out operations on kernel objects. For instance, such an operation could be an IPC call to an other processes.

A central feature of capability-based access control is the absence of ambient authority. Tasks in L4Re start with no authority, i.e. no capabilities by default. To perform useful work and interact with the system they must be granted the required capabilities explicitly. Other parts of the system cannot be accessed and are in fact not even visible to said task. Sharing object access is simple as the owner of

a capability is free to grant it to other tasks (*delegation*). The reverse is also possible. The owner of a capability can *revoke* (i.e., void) it, preventing its future use.

In L4Re, capabilities manifest as the permission to interact with a certain kernel object. Depending on the type of the respective object and the *rights* of the capability pointing to it, such interactions could e.g. send an IPC message to another thread via an IPC gate, or destroy a kernel object. With L4Re, a capability can be delegated by sending it over an IPC gate connected to another process, whereat the kernel takes care of copying the capability into the target process of the IPC. Similarly, the owner of a capability can instruct the kernel to revoke it at any time. Consequently, the L4Re microkernel is responsible for maintaining the system's security by shielding the capability tables of all userspace processes against unauthorized manipulation. Note, that the destruction of a kernel object implicitly revokes all capabilities pointing to it. This property enables L4Re processes to revoke all capabilities handed out to a certain client in a single operation.

3 Containers on L4Re

This section describes how the functional features and the security properties of Linux containers can be implemented on a microkernel-based OS like L4Re. In the following, we will use the term *compartment* for referring to an entity with container-style isolation running on L4Re, differentiating it from the Linux equivalent which we keep calling *container*.

3.1 On the Compartment Architecture

The management of compartments on L4Re is handled by a *compartment service* called *Oak*. In Linux terminology, Oak would be similar to a program like runC. As common for microkernel architectures, most other system services run in separate tasks, yielding strong isolation between them.

Just as with Linux, at the core, a compartment on L4Re consists of a set of processes. Hence, the smallest compartment possible is a single task (L4Re process). All tasks of a compartment share a common set of capabilities, which define which parts of the system the tasks belonging to the respective compartment can see and interact with.

In order to make these restrictions transparent to the application running inside a compartment, L4Re also provides `namespaces` that allow a task to refer to a capability using a name. Similar to `namespaces` on Linux, this abstraction provides some form of virtualization as the compartment service can set the mapping of names to capabilities for each compartment individually. For instance, two compartments could both see a capability named "`/usr`", but for each compartment the capability linked to the name could grant access to a different file system service. This mechanism enables Oak to present different views of the system to compartments.

Upon receiving a request for launching a compartment, Oak first gathers the resources that the respective compartment needs to run. To this extent Oak creates new sessions with each of the system services in charge of managing the requested resources. As a result of creating a new session with a system service, the compartment service receives a capability to a new IPC gate. This gate acts as a handle for accessing the session with the system service.

Now, the compartment service delegates the capabilities gathered during the resource allocation phase to each task of the compartment, giving them access to the corresponding system services. It then launches the compartment's tasks and monitors their exit status.

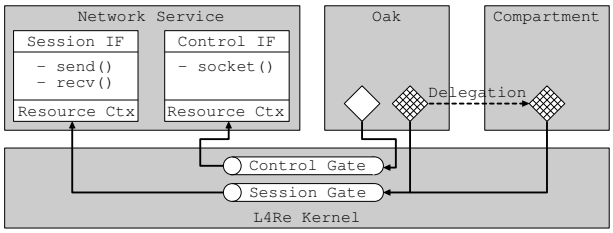

Figure 1. Example for an L4Re system service exposing different interfaces (IF) in multiple sessions (white boxes). The diamonds represent capabilities to the respective IPC gates.

As the last step in the life cycle of a compartment, Oak collects the artifacts left over from the execution of a compartment. This also implies that the compartment service closes all remaining sessions allocated for a compartment by deleting the corresponding kernel objects. This revokes all capabilities pointing to the compartment's sessions and also frees the resources attached to them.

3.2 Enforcing Compartment Restrictions

A system service in L4Re can securely identify a client based on the IPC gate used by the client for sending messages. System services leverage this property to adapt the interface that they offer via an IPC gate. As shown in Figure 1, system services often expose a dedicated IPC gate for invoking control plane operations like the creation of a new session. In the context of compartments, only the compartment service gets access to such gates as untrusted applications running inside a compartment must not be able to spawn arbitrary sessions and thus break their resource constraints. The compartments receive capabilities to session-specific gates that merely expose data-plane operations such as sending or receiving data from the network. This scheme allows for a fine-grained limitation of the API accessible to a compartment. Dedicated mechanisms for restricting the system interface available to a compartment (such as `seccomp-bpf` on Linux) are hence not required with L4Re.

The secure identification of clients furthermore allows L4Re services to implement *resource groups*, a resource constraining mechanism similar to Linux' `cgroups`. By attaching a resource consumption context to each IPC gate as shown in Figure 1, a service is able to control the resource usage of each of its clients. Depending on the concrete resource, a consumption context may contain a priority or a budget. Whenever a request arrives via a certain gate, a service would first decide whether to admit it based on the remaining resource budget in the corresponding resource consumption context.

In L4Re, the compartment service takes care of creating appropriate resource groups for constraining a compartment. To this end, it adds a description of the respective resource limits to each session creation request executed before launching the compartment. By delegating the resulting capability, the compartment is then implicitly added to the resulting resource group.

3.3 Implementing Compartments

In order to perform meaningful work in a compartment, L4Re needs to provide more functionality than the plain Oak compartment service. We chose to implement system services (networking etc.) as

native L4Re applications. Compared to using an *L⁴Linux* VM [18] that provides the respective features, native system services do not come with the drawback of a large TCB shared by all compartments.

In addition to writing the compartment service described before, we therefore implemented several components to provide a prototype execution environment to L4Re compartments. First, we built a network stack to make the Oak service accessible remotely. To this end, we ported the driver for Intel's X540 NIC from the Ixy driver framework [12] to L4Re. We furthermore designed and implemented *LUNA*, a network service that multiplexes a NIC between multiple applications and implements a simple UDP/IP stack. Second, we created an in-memory file system service called *spafs*. Spafs has read and write support and implements directories.

Lastly, we enabled `moe` to run on multiple cores, avoiding expensive cross-core IPC when interacting with it. However, the internals of `moe` are still serialized, as they are protected by a single lock.

4 Security Evaluation

To substantiate our claim, that a container infrastructure based on a microkernel is more secure compared to the Linux implementation, we present a short comparison of the respective TCBs and a brief vulnerability discussion based on selected, past CVEs.

4.1 Trusted Computing Base Comparison

The Linux kernel, a monolith, has a large code base, supporting a multitude of system calls, submodules, and device drivers. Even with a selective configuration, Linux runs a lot of code in privileged mode, thus resulting in a large TCB. In contrast, the L4Re microkernel is much smaller, only implementing basic mechanisms. Many functionalities are outsourced to unprivileged userspace services. This simplicity and modularity allows for the smaller, more manageable TCB of Oak.

In general, the reduced kernel code size of microkernels makes them amenable to formal verification [21] as well as security certification as [9]. Outside the kernel, microkernel-based systems benefit from the absence of an all-powerful root account, and capability-based access control which encourages a system design following the principle of least authority.

4.2 Vulnerability Study

The smaller TCB size and privilege-reduced components should result in a better security posture. To demonstrate this point, we conducted a study of existing vulnerabilities in container infrastructure on Linux. In the following, we will discuss some sample vulnerabilities that highlight how a microkernel approach can reduce the attack surface of a container infrastructure.

Seccomp uses the Berkeley Packet Filter (BPF) [2] to filter system calls at kernel level to restrict the kernel interface available to containers. With BPF [23] being a code interpreter at kernel level and seccomp using BPF to express its filter rules, security issues arise from flaws in either seccomp or the underlying BPF.

Oak does not require such filtering mechanisms. Only interfaces, for which capabilities are granted are visible to a container. To bypass this access control, the capability implementation itself would need to be compromised, which is part of the microkernel. While implementation bugs in a microkernel may exist, due to little code running in CPU privileged mode, we consider such a compromise unlikely.

Futhermore, the namespace isolation is also a possible source of bugs. CVE-2018-18955 [1] describes a vulnerability, allowing privilege escalation via mishandling of nested user namespaces. In L4Re, resource access is enforced by capabilities, so such an escalation can only occur by compromising the capability system itself. Thus, such a vulnerability is less likely to arise on L4Re.

Similarly, cgroups have also had vulnerabilities. CVE-2022-0492 [4] describes an exploit, where a container can escalate privileges and bypass namespace isolation due to a flaw in a cgroups feature. In Oak, resource restrictions are implemented by resource contexts in userspace components. As these components may exhibit similar implementation flaws, resource restrictions may be equally circumventable. However, such a compromise would only affect a single resource and would certainly not affect the kernel or inter-container memory isolation.

5 Performance Evaluation

In the following, we compare the implementation of compartments on L4Re with standard Linux solutions for containers from a performance perspective. The microbenchmarks we use aim at creating preliminary insights into the performance of compartments on L4Re. As we want to focus on the bare virtualization mechanisms, we did not deploy warm start optimizations such as provisioning of hot standby containers with pre-initialized runtimes.

As a comparison for L4Re compartments, we measured Linux stock processes, runC [17], and Kata Containers [14] with Firecracker [6]. While standard Linux processes do not offer the same security properties as containers, they represent an optimal baseline with respect to performance.

5.1 System Setup

All measurements presented in the following were done on two identical dual-socket servers that are equipped with two Intel Xeon Platinum 8358 CPUs and 500 GiB of main memory each. For all benchmarks, we disabled both hyperthreading (SMT) and temporary overclocking (TurboBoost). We furthermore set the CPU's pstate configuration to maximum performance mode. Additionally, the servers both feature a 10 Gbit (Intel 82599 / Intel X540) Ethernet NIC.

The results for benchmark setups running on Linux were recorded using kernel version 6.7.4, runC version 1.1.10, containerd version 1.7.9 and Kata version 3.3.0 with a small patch to enable the measurement of startup times in Firecracker.

5.2 Container Startup Latency

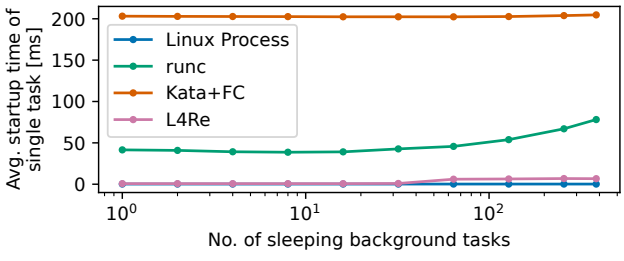

Figure 2. Average startup latency of a single container as function of the number of idle containers present in the system.

First, we measured the time spent for creating and launching an empty container. The containers benchmarked in this setup all consist of a single process and only measure a timestamp before terminating. For runC the container creation phase includes the setup of `cgroups`, `seccomp` filters, and file systems. To obtain best-case performance, we set empty `seccomp` filters. With Kata Containers the boot time of the Firecracker microVM is measured by using the boot-timer device provided by Firecracker. On L4Re, meta operations for spawning a compartment, like creating a new session with the Oak service contribute to the startup latency.

Figure 2 shows the startup latency of a single container as a function of idle containers already present in the system. We chose this metric to examine whether the overall number of running containers influences their startup performance. Spawning a standard process on Linux is the fastest option with a startup latency of roughly 190 μs, regardless of the number of processes in the system. Starting a compartment on L4Re takes around 720 μs for less than 32 background compartments. If there are more running compartments, the startup latency increases slowly up to 5 ms with 384 compartments active. We attribute this increase to NUMA effects that occur since the compartment engine of L4Re schedules compartments round-robin on all cores and L4Re currently has no awareness of NUMA memory architectures. Starting an empty container with runC is considerably slower, with a startup latency of 45 ms for a single container. We attribute the difference to L4Re to the setup overhead of in-kernel constraining mechanisms like `cgroups`. Kata with firecracker shows the highest boot times (around 200 ms), as the setup of a virtual machine is more expensive than creating a container.

As a second benchmark, we recorded the container startup latency when starting multiple containers in parallel. Figure 3 shows the results of these measurements. Linux processes again have an almost constant startup time, independent of the number of processes spawned in parallel. The same holds for Kata Containers with Firecracker that is able to retain its boot time from the sequential benchmark (200 ms) regardless of the number of instances spawned in parallel. On L4Re, it takes roughly 15 ms to start two compartments in parallel. This number grows to 100 ms for eight parallel compartment launches. RunC takes significantly longer for a low parallelism of container starts (30 ms for two parallel launches). When starting more than 16 containers in parallel, runC and L4Re compartments performed roughly similar, both showing high tail latency.

5.3 I/O Performance

In order to obtain an estimate of the I/O performance that a microkernel OS like L4Re is able to provide, we compared the network performance available to compartments on L4Re to that achieved by Linux containers. We performed a UDP-based ping-pong benchmark over a 10 GiB/s Ethernet interface. The remote host that was pinged always ran a native Linux process, while the latency was recorded on the client side. We varied the client platform and measured for Linux processes, L4Re tasks, and RunC containers. The latency was roughly the same on all platforms with an average value of 40 μs. We observed runC to have more outliers, which we attribute to the additional in-kernel layers traversed during the processing of a packet.

Figure 4 shows the bandwidth measured using multiple sockets in parallel. As expected, Linux and runC performed similar as they use the same network stack. With a low degree of parallelism, Linux achieved a higher bandwidth than L4Re (900 MiB/s vs. 350 MiB/s) since L4Re does not implement advanced NIC features such as receive side scaling (RSS), and all driver processing is done on a single core, which is also used by the first thread of the benchmark. With increasing parallelism and thus more threads sending traffic from different cores, this effect diminishes.

6 Related Work

The emergence of microservices motivated research for optimizing lightweight virtualization with respect to performance and security. Chestnut [10] allows for automatically generating `seccomp-bpf` filters from application binaries, only granting access to system calls the application needs. Such Linux-related container-hardening techniques become superfluous with our approach because a capability-based approach like L4Re promotes adherence to the principle of least authority.

BlackBox [16] implements secure container execution on an untrusted operating system. Containers are shielded against the operating system using virtualization and system call sanitization. Although BlackBox achieves a small TCB for confidentiality, the otherwise untrusted operating system needs to be trusted to guarantee availability. The same applies to container implementations on both L4Re and Linux.

In pursuit of increasing the security of lightweight virtualization, lightweight VMs gained traction. Amazon's Firecracker [6] proved that these VMs are competitive in large deployments. Manco et al. demonstrated the use of unikernels together with carefully designed VM infrastructure, yielding startup times and application performance even better than that of containers on Linux [22]. In contrast to unikernel-based VMs, containers on L4Re have native access to operating system features like multithreading for performance and address-space-based compartmentalization for security.

7 Conclusion and Future Work

In this paper we investigated the design and implementation of containers on a microkernel-based OS. While Linux needs to implement additional restriction mechanisms for providing container-grade isolation to processes, such mechanisms are either not needed or conceptually much simpler on a capability-based microkernel, since it fully isolates processes by default. Due to the fundamental security benefits of these microkernels, their container design awards a more robust security posture compared to containers on Linux, since the overall attack surface exposed to malicious clients is much smaller. Early performance measurements showed that even without major optimizations, microkernel platforms are able to provide competitive performance both with respect to container startup latency as well as for I/O performance.

We thus believe that our approach is worth exploring further. To demonstrate the applicability of our ideas to real-world use cases, we currently work on implementing a `python3`-based Function-as-a-Service environment on L4Re. Moreover we plan to take actions towards enabling our container engine to execute OCI-compliant container images.

Acknowledgements

This research is funded by the German Research Council DFG through grant 502457159 (FOSSIL).

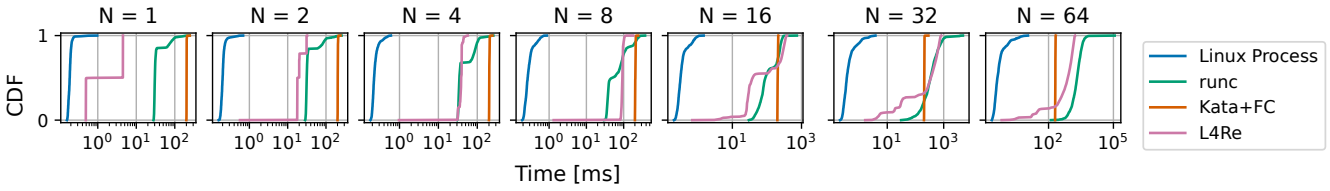

Figure 3. Cumulative distribution function of the container startup latency when launching N containers in parallel.

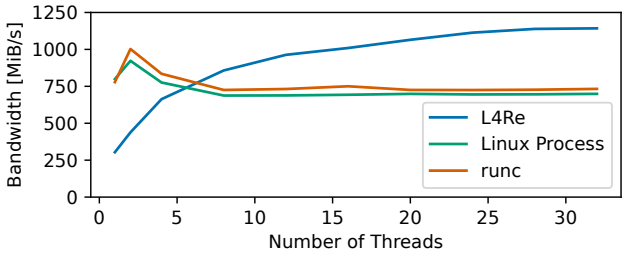

Figure 4. Bandwidth for transmitting data over a 10 GBit Ethernet interface (using 1472 B UDP packets) as a function of the number of parallel data streams.

References

[1] NVD - CVE-2018-18955. https://nvd.nist.gov/vuln/detail/CVE-2018-18955, 2018. [Online; last accessed on April 10, 2024].

[2] Seccomp BPF (SECure COMPuting with filters) — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.18/userspace-api/seccomp_filter.html, August 2018. [Online; accessed 18. Apr. 2024].

[3] New seccomp mode aims to improve performance. https://lore.kernel.org/linux-security-module/c22a6c3cefc2412cad00ae14c1371711@huawei.com/t/, 2020. [Online; last accessed on November 14, 2023].

[4] NVD - CVE-2022-0492. https://nvd.nist.gov/vuln/detail/CVE-2022-0492, 2022. [Online; last accessed on April 10, 2024].

[5] NVD - CVE-2022-30594. https://nvd.nist.gov/vuln/detail/CVE-2022-30594, 2022. [Online; last accessed on April 10, 2024].

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020.

[7] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: a study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113. Association for Computing Machinery, 2020.

[8] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 45–58. USENIX Association, 1999.

[9] BSI. BSI-Schrift 7164: Liste der zugelassenen IT-Sicherheitsprodukte und -systeme. https://www.bsi.bund.de/SharedDocs/Zulassung/DE/Produkte/L4Re_Secure_Separation_Kernel_VS_BSI-VSA-10624.html. [Online; last accessed on April 10, 2024; in German].

[10] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In Yinqian Zhang and Marten van Dijk, editors, *CCSW@CCS '21: Proceedings of the 2021 on Cloud Computing Security Workshop, Virtual Event, Republic of Korea, 15 November 2021*, pages 139–151. ACM, 2021.

[11] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[12] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, pages 1–12. IEEE, 2019.

[13] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 171–172. IEEE Computer Society, 2015.

[14] OpenInfra Foundation. Kata containers. https://katacontainers.io/, 2024. [Online; last accessed on April 18, 2024].

[15] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 66–77. ACM, 1997.

[16] Alexander Van't Hof and Jason Nieh. Blackbox: A container security monitor for protecting containers on untrusted operating systems. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 683–700. USENIX Association, 2022.

[17] Open Container Initiative. Runc. https://github.com/opencontainers/runc, 2024. [Online; last accessed on April 19, 2024].

[18] Hermann Jork, L Frank, Mehnert Reuther, Martin Pohlack, and Alexander Warg. An i/o architecture for microkernel-based operating systems. September 2003.

[19] Peter Van Der Kamp and Robert N. M. Watson. Jails: confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[20] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 139–144. USENIX Association, 2013.

[21] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[22] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 218–233. ACM, 2017.

[23] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[24] Prateek Sharma, Lucas Chaufournier, Prashant J. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 1. ACM, 2016.