Security-Performance Trade-offs of Kubernetes Container Runtimes

William Viktorsson$^+$, Cristian Klein^{+*}, Johan Tordsson^{+*}
$^+$Department of Computing Science, Umeå University, Umeå, Sweden; *Elastisys AB, Umeå, Sweden
{williamv,cklein,tordsson}@cs.umu.se

***Abstract*—**
The extreme adoption rate of container technologies along with raised security concerns have resulted in the development of multiple alternative container runtimes targeting security through additional layers of indirection. In an apples-to-apples comparison, we deploy three runtimes in the same Kubernetes cluster, the security focused Kata and gVisor, as well as the default Kubernetes runtime runC. Our evaluation based on three real applications demonstrate that runC outperforms the more secure alternatives up to 5x, that gVisor deploys containers up to 2x faster than Kata, but that Kata executes container up to 1.6x faster than gVisor. Our work illustrates that alternative, more secure, runtimes can be used in a plug-and-play manner in Kubernetes, but at a significant performance penalty. Our study is useful both to practitioners – to understand the current state of the technology in order to make the right decision in the selection, operation and/or design of platforms – and to scholars to illustrate how these technologies evolved over time.

I. Introduction

Application containers are stand-alone software packages that consist of one or more applications and the libraries, programming runtimes, and other dependencies required to operate the applications. Containers provide application portability across platforms and are light-weight in both resource usage and required storage. These features simplify development and deployment of applications. The two key sandboxing mechanisms commonly used in containers are namespaces for isolation (of processes, networks, filesystems, etc.) and resource management (of CPU, memory, network, etc.), often implemented using cgroups. A potential shortcoming with container sandboxing is that the host operating system is shared among multiple containers on the same host. This contrasts the Virtual Machine (VM) approach where multiple applications share a host but each have their own host operating system.

Container have gained widespread adoption across a broad spectrum of industries, in particular following the releases of first Docker[1] and later Kubernetes[2]. A wave of Linux kernel vulnerabilities, such as Dirty COW[3], which can be used to escape containers, increased fears of containers providing insufficient isolation. This gave rise to multiple alternative runtimes that aim to increase security, e.g., by adding additional layer(s) of indirection and/or reducing the attack vector.

We analyze three container runtimes with respect to security and performance: Kata [1], gVisor [2], and runc (the default Kubernetes runtime). These were selected as they integrate seamlessly with Kubernetes, which is required for a runtime to gain broad adoption. For this reason, we excluded several interesting runtimes identified in the cloud native landscape[4], including the Firecracker microVM[5], and Nabla[6] that reduce the attack surface by limiting the set of accessible system calls. Firecracker when ran under Kata Containers nearly made it to the list of technologies evaluated, but compatibility issues with the benchmark applications prevented it from being properly evaluated. Nabla Containers failed at the compatibility step as well due to requiring applications and container images to be ported and built specifically for Nabla.

While performance evaluation of various containerization platforms is not new ([2]–[7]), our work distinguishes itself by evaluating container runtimes in a typical Kubernetes cluster using benchmark applications that are representative of the workloads that run in such clusters (Section II-B).

Our contributions include an analysis of the security mechanisms of the three studied runtimes (Section II), where we discuss to what extent they fulfill a set of requirements with respect to isolation and resource management. We also empirically evaluate the performance of the three runtimes (Section III) using a microservice benchmark (TeaStore), an in-memory data-store (Redis), and a big data processing framework (Spark). In our experiments, runc performs up to 5x better than Kata and gVisor in both container deployment time and application execution. Notably, gVisor deploys containers faster than Kata (up to 2x) but executes applications slower.

II. Background

In this section, we present the relevant background of our work. We first summarize the relevant architectural choices of the selected container runtimes and quantify how well each container runtimes fulfills the security requirements. Finally, we summarize earlier benchmarking efforts.

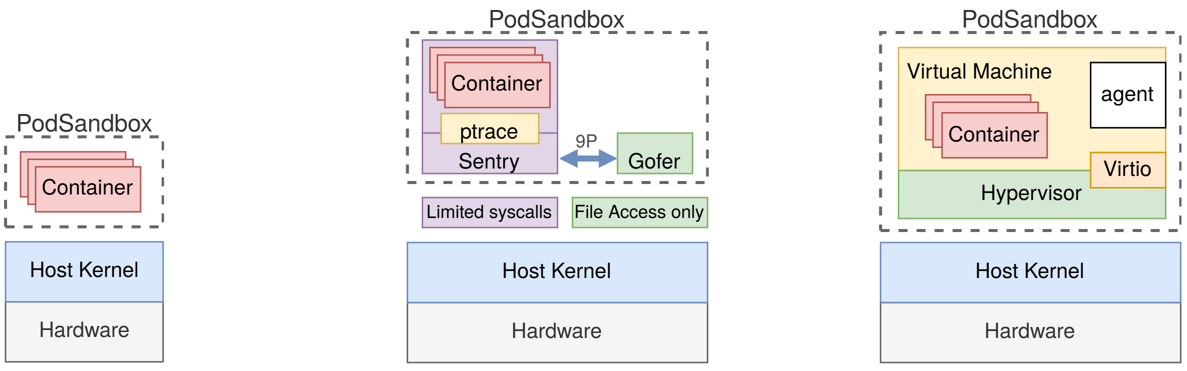
A. Security-Relevant Architecture

Container runtimes need to ensure process isolation, filesystem isolation, device isolation, Inter-Process Communication (IPC) isolation, network isolation and resource management. Fig. 1 compares the architecture of runC (the default container runtime), gVisor and Kata.

gVisor introduces a novel approach of provisioning a virtualized environment by moving system interfaces normally implemented by the host kernel into a user space kernel.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[1]https://www.docker.com
[2]https://www.kubernetes.io
[3]https://dirtycow.ninja/
[4]https://landscape.cncf.io/category=container-runtime
[5]https://github.com/firecracker-microvm
[6]https://nabla-containers.github.io/



(a) runC (b) gVisor (c) Kata

Fig. 1: Comparison of the high-level architectures of the studied container runtimes.

System calls are intercepted and performed on behalf of the container, without the overhead imposed by a hypervisor [2].

gVisor consists two main components. **Sentry** is a user-space kernel that intercepts system calls made by the application. The platform responsible for intercepting syscalls is ptrace by default, but gVisor can be configured to use the Kernel-based Virtual Machine (KVM) [8] instead. The sentry provides a full network stack (called netstack) that isolates all networking from the host network stack. **Gofer** is a separate process running in each gVisor instance that provides file system access to the container using the 9P protocol [9].

Kata containers provide lightweight VMs with the goal of achieving performance equal to containers, but with a second layer of isolation using hardware isolation. It supports multiple hypervisors including QEMU, Firecracker and Cloud Hypervisor [1]. A VM is booted using a compact guest OS image, optimized for low kernel boot time and minimal memory footprint. Libcontainer is used to create the execution environment inside the VM, and run containers within that environment [10]. This implies that the containers run on the guest with cgroups and namespaces applied in the same manner a runC container uses these mechanisms on a host. To enable communication from the host into the guest, a Kata-agent is created, a process running inside the VM as a supervisor for containers running on the guest. Commands to the Kata runtime are forwarded to the Kata-agent over gRPC [11] that in turn makes use of Libcontainer to manage the lifecycle of containers. Network and storage are accessed via a set of paravirtualized devices through the virtio virtualization framework [12].

Table I presents the number of layers of indirection for each container runtime. More layers of indirection entail more security, as an attacker who gained control of the application would need to find more vulnerabilities to compromise security of the container runtime.

As expected, runC, being optimized for performance, only provides a single layer of indirection for each security requirement. Next comes gVisor with two layers of indirection for almost all requirements except resource management. In our security analysis, Kata comes out as the winner, having two layers of indirection for all security aspects.

TABLE I: Layers of indirection for each runtime.

Security Requirement	runC	gVisor	Kata
Process isolation	1	2	2
Filesystem isolation	1	2	2
Device isolation	1	2	2
IPC isolation	1	2	2
Network isolation	1	2	2
Resource management	1	1	2

B. *Earlier Benchmarking Results*

Earlier work, including projects behind the secure runtimes, compare container runtimes using micro-benchmarks [13]. Various stressors are for CPU, memory or I/O. In some cases, syscalls are called repeatedly to highlight the careful design that went into optimizing a specific use-case or code path. Most of these micro-benchmarks are executed on bare-metal without needing to deal with nested virtualization [4]. The latter is more likely to be encountered typical Kubernetes clusters. Other run micro-benchmarks in nested virtualization environments [14], even on top of Kubernetes [15].

Our benchmarks complemented the above works as follows:

- We use full applications deployed on top of Kubernetes as benchmarks, as opposed to micro-benchmarks.
- We evaluate the container runtimes on Kubernetes clusters in a nested virtualization environments, as is common with public cloud providers.

III. Performance Evaluation

We evaluate the performance of each container runtime, focusing on deployment and execution time for a set of typical microservice applications running in Kubernetes.

A. *Experiment Setup*

TABLE II: Components used in experiments.

Component	Version / Configuration
Kubernetes	1.17.0
Containerd	1.3.0 (overlayfs)
runC	1.0.1-dev
gVisor	2019-11-14 (ptrace)
Kata	1.9.0 (QEMU+KVM)

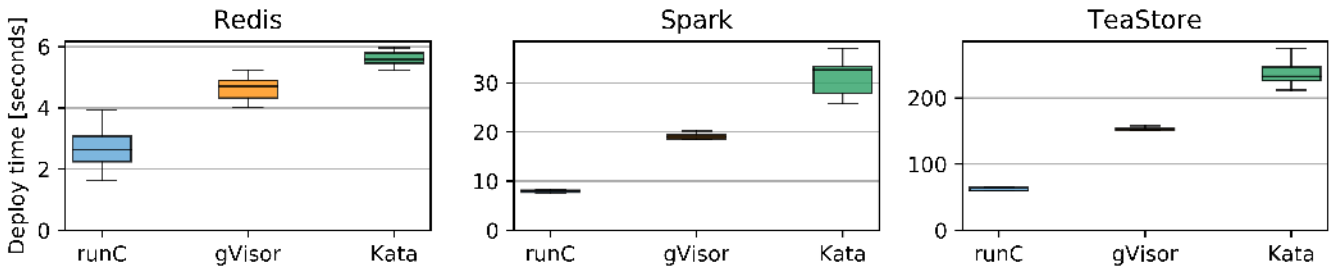


Fig. 2: Deploy time – less is better – for each container runtime.

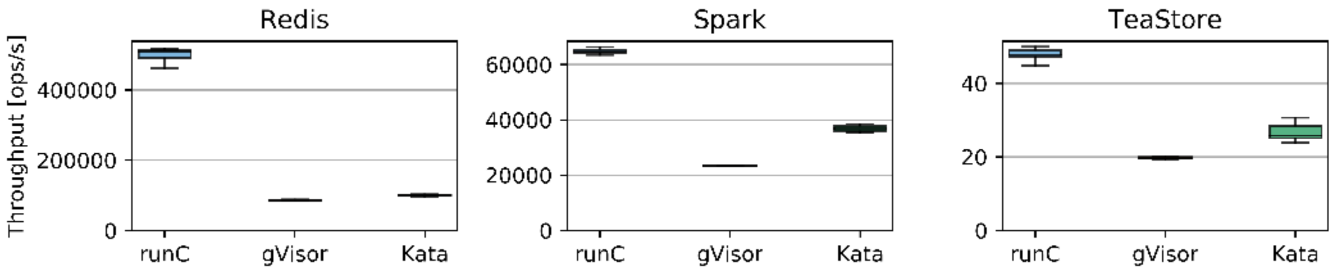


Fig. 3: Application performance – more is better – for each container runtime.

Infrastructure: All experiments were conducted by deploying and running applications on a two node Kubernetes cluster deployed on Google Compute Engine VM instances. The master ran on a n1-standard-2 (2 vCPUs, 7.5 GB memory) machine and the worker ran on a n1-standard-8 (8 vCPUs, 30 GB memory) machine. These machines ran Ubuntu 16.04 with nested virtualization enabled and were configured to use the Intel Haswell CPU platform. All Pods ran only one container and each container was limited to 1 vCPU and 2GiB of working memory.

All container runtimes were installed and exposed for usage within the same Kubernetes cluster through the RuntimeClass Kubernetes feature gate [16]. This ensured that all experiments were conducted in the same hardware and software setting without the need of tearing down and reconfiguring the Kubernetes cluster for each runtime. This setup also serves as a proof of concept for multi-runtime Kubernetes clusters. All benchmarks were repeated 10 times. Performance-relevant configuration details are presented in Table II.

Applications: **TeaStore** [17] is a microservice benchmark. It emulates a web store that features browsing, selecting and making orders of tea and tea supplies. It features five services each with different responsibilities. In the experiments, each service had only one replica. The benchmark score for TeaStore is defined as the average *requests per second* throughput for eight available operations in the TeaStore API such as adding and removing items or listing and browsing items. The operations are performed sequentially in a round robin fashion by two benchmark clients running simultaneously. Each client performed 200 of each request. Time to deploy was measured as the time from requesting all Kubernetes objects required to run TeaStore until each service reported itself as successfully running on the TeaStore status page. TeaStore deployment includes time consuming training of a ML model for a recommendation system in the web store.

Redis [18] is an in-memory data-store that features data structures such as hashes, lists, sets and more. It has built-in replication in a primary-replica manner. It can also provide high availability through a Redis Sentinel that monitors Redis instances and takes action to promote replicas to primary when a primary seems to be failing. For the benchmark, one primary, two replicas and two sentinels were deployed. The benchmark score for Redis is defined as *request per second* throughput for the O(1) GET operation. The Redis built-in benchmark tool *redis-benchmark* was used to perform and measure the throughput of the operations. The tool was configured to spawn 50 clients that each ran 16 operations in parallel until each operation was executed one million times. Time to deploy was measured as the time from requesting all Kubernetes objects required to run Redis until the Redis master was available for external requests.

Spark [19] is a distributed general purpose computing framework for big data processing. A Spark cluster was deployed with one master and 3 workers. The benchmark score for Spark is defined as the average amount of *primes found per second* when finding all prime numbers in the first million numbers. Apache Zeppelin [20] was used to submit workloads using *PySpark*, the Spark Python API. Time to deploy was measured as the time from requesting all Kubernetes objects required to run Spark until all workers had joined the master.

B. Experimental Results

Figs. 2 and 3 presents the results for deployment time and application performance, respectively. The x-axis represents the three benchmark applications, while the y-axis presents the metric. All experiments are repeated 10 times, each box-plot represents the quartiles.

Deployment Time: When comparing the deployment of TeaStore to runC with the other runtimes, gVisor take about three times as long to deploy and Kata almost four times as long. The same relationship in deployment times are true for deploying Spark. The deployment time for Redis shows a slightly different ratio between the runtimes, with Kata and gVisor both deploying Redis close to twice the time of runC.

Given their architecture, these results are expected and represent the cost of launching two extra processes (Sentry and Gofer) for gVisor, as well as a VM and hypervisor for Kata.

Application Performance: When comparing application performance, gVisor and Kata achieved around 40% and 60%, respectively, of the runC score for TeaStore. A similar relationship exist for Spark, with Kata outperforming gVisor performance-wise. Regarding Redis performance, both gVisor and Kata scored a request throughput of around 20% of that of runC. This is due to the fact that Redis performance is measured using a simple GET requesting in-memory data. The low CPU and memory requirements of the operation itself implies that the application performance is mostly determined by networking performance.

C. Overhead

Besides impacting deployment and application performance, container runtimes also impose an overhead in terms of CPU and memory usage. We measured such overhead by running an idle container for five minutes and scraping resource usage metrics for the entire Pod using cAdvisor [21]. The results (not shown for briefness) highlight that the CPU overhead is negligible for all runtimes, whereas for memory this is not the case: gVisor imposes a 14 MB memory overhead and Kata 143 MB.

IV. Outlook

In this paper, we evaluate the security capabilities and performance of two alternative Kubernetes container runtimes: Kata and gVisor. Overall, our results show that higher security comes at a very high cost, both in terms of deployment time and application performance loss. When focusing on the two security-hardened runtimes, Kata – which provides the most security layers – also features the highest deployment time, but – surprisingly – better application performance than gVisor, both for a compute-bound and network-bound applications.

Increasing the security of container runtimes by adding layers of indirection is still a fast-evolving landscape. Scholars need to regularly evaluate said landscape to better understand trends, design trade-offs and fundamental limits. For our study, Kata and gVisor were selected due to their plug-and-play compatibility with Kubernetes, but other container runtime – in particular Firecracker and Nabla – are soon expected to challenge their favored position. In the future, although each container runtime is expected to conceptually stay the same, the underlying technology that they use may change, significantly altering their performance. For example, gVisor is switching from ptrace-based syscall interception to KVM.

Although the performance of security-hardened container runtimes will increase as the technology matures, it is unlikely that there will be a one-size-fits-all container runtime. Therefore, besides comparing the container runtimes in a competition, future work should also focus on how container runtimes can complement each other, perhaps providing the user with a wider range of choices than only between strong security and high performance.

References

[1] Kata Containers. The speed of containers, the security of VMs. https://katacontainers.io/. Accessed: 2020-06-20.

[2] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[3] 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*, pages 101–113, 2020.

[4] Xu Wang. Kata containers and gVisor: a quantitative comparison. https://www.openstack.org/videos/summits/berlin-2018/kata-containers-and-gvisor-a-quantitative-comparison. Accessed: 2020-06-20.

[5] Allison Randal. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Computing Surveys*, 53(1):1–31, 2020.

[6] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *17th International Middleware Conference*. ACM, 2016.

[7] Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. Virtualization techniques compared: performance, resource, and power usage overheads in clouds. In *ACM/SPEC International Conference on Performance Engineering*, pages 145–156, 2018.

[8] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[9] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Computing systems*, 8(2):221–254, 1995.

[10] Kata Containers. Kata containers architecture. https://github.com/kata-containers/documentation/blob/master/design/architecture.md. Accessed: 2020-06-20.

[11] gRPC. A high performance, open-source universal RPC framework. https://grpc.io. Accessed: 2020-01-03.

[12] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[13] The gVisor Authors. gVisor performance. https://gvisor.dev/docs/architecture_guide/performance/. Accessed: 2020-06-20.

[14] Shiny Sebastian Eric Ernst, K. Y Srinivasan. Kata containers: the speed of containers, security of VMs - even in a nested environment! https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Eric-Ernst_KY-Srinivasan_OSS-slides-kata-nested.pdf. Accessed: 2020-06-20.

[15] Stig Telfer Bharat Kunwar. I/O performance of kata containers. https://www.stackhpc.com/kata-io-1.html. Accessed: 2020-06-20.

[16] The Kubernetes authors. Runtime class. https://kubernetes.io/docs/concepts/containers/runtime-class. Accessed: 2020-06-20.

[17] Jóakim von Kistowski et al. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *MASCOTS*, 2018.

[18] Redis Labs. Redis. https://redis.io. Accessed: 2020-06-20.

[19] Matei Zaharia et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[20] Apache. Zeppelin. https://zeppelin.apache.org. Accessed: 2020-06-20.

[21] Google. cadvisor. https://github.com/google/cadvisor. Accessed: 2020-06-20.