Performance Analysis Between RunC and Kata Container Runtime

Rakesh Kumar
International Institute of Information Technology
Bangalore, India
rakesh.kumar092@iiitb.org

Prof. B Thangaraju
International Institute of Information Technology
Bangalore, India
b.thangaraju@iiitb.ac.in

***Abstract*—Kata container unifies the security advantages of virtual machines with the speed and flexibility of containers. It is designed along the Open Container Initiative's specification to perform like containers, enforcing the workload isolation. Since Docker containers share the kernel of the host operating system, it is exposed to potential security threats. Kata container mitigates these security issues by introducing a lightweight virtual machine with separate kernels to create an isolated environment for the container. This research work aims at providing a quantitative analysis between the Docker and Kata container runtimes. The performance of both the technologies has been evaluated based on boot tests, system utilizations, and benchmarks, showing a detailed analysis of the security and performance tradeoff between both the container technologies.**

Keywords—container, docker, kata container, performance, benchmark, runC;

I. Introduction

In the past few years, containers have been very popular in the software industry. Container makes it easier to pack, ship, and run codes reliably. It packages the entire runtime environment, application, libraries, dependencies, and other necessary binaries along with configuration files needed to run an application. A containerized application does not have to worry about the underlying hardware or the operating system to run itself, thus it can move across different environments seamlessly. This makes the container lightweight and easily deployable. Additionally, containers help in breaking a monolithic application into microservices running as a separate module inside a container. It has greatly simplified the application build, test, and deployment phases allowing developers to work on identical development environments without any platform constraints.

Docker is the most popular containerization platform. It offers an entire open-source ecosystem for building, deploying, and managing containerized applications. It simplifies and accelerates the workflow while offering developers the flexibility to innovate with their choice of application stacks, tools, and deployment environments for each project [1]. Docker uses *runC* as its default low-level container runtime. The runC container runs over an operating system, where multiple containers share the same operating system kernel, but each container can be constrained to use only a predefined amount of resources available in the host operating system. The shared components of the operating system are read-only and each container has a mount point of its own for writing. Docker brings security to the applications running in a shared environment, but containers alone are not an alternative to taking proper security measures. Due to the involvement of a shared kernel, the containers in a cluster are exposed to the possibility of security threats if any of the containers is compromised. The need for complete isolation in containers to combat the security limitations led to the development of Kata containers.

Kata container emerged as a collaboration of Intel's Clear Container and Hyper's runV runtime, both of which are open-source projects. This brings together the best of both technologies with a common vision of retooling virtualization to fit container-native applications, in order to deliver the speed of containers, and the security of virtual machines [2]. Kata container supports industry standards and has been designed along with the Open Container Initiative (OCI) specifications. It aims at solving the traditional container's security problem. In Kata containers, each container is equipped with its lightweight virtual machine and a mini kernel, providing container isolation via hardware virtualization which improves the security layer [3].

Considering the potential security benefits provided by Kata containers, together with a significant increase in use cases, this research work focuses on providing a quantitative performance analysis. The rest of the paper is organized as follows, Section II presents an architectural difference between Docker container and Kata container runtime, Section III explains the methodology and system used to perform the test, Section IV consists of the performance evaluation and results, and Section V summarizes the assessments.

II. Architecture

Before we dive into the architecture of runC and Kata container runtime, we need to look at some of the components of Docker. Docker is used as a reference to the whole set of Docker tools responsible for running and managing a container [4]. The core part of the Docker ecosystem is the Docker Engine. It is an application based on client-server architecture, which uses the Rest API to interact with Docker clients. The Docker daemon, *dockerd*, is an important component of the Docker Engine. It is a persistent process that listens for API requests, processes the request, and manages containers. The Docker command-line interface, a tool that is responsible for communication with the user, takes commands such as build, run, and invokes the dockerd API. The docker daemon uses *containerd*, a high-level container runtime that manages the lifecycle of containers. containerd introduces a new layer on top of low-level container runtime to abstract the system calls and operating system specific functionalities needed to run and manage the container lifecycle. It is also responsible for managing storage and network namespaces. Docker uses runC as its default low-level container runtime.

RunC is a lightweight OCI compliant, high performance, and low-level container executor. At the core, runC uses *libcontainer* to do the heavy task of creating containers and works as a wrapper for abstracting the system calls [5]. It

978-1-7281-6828-9/20/$31.00 ©2020 IEEE

provides implementations to create a container using namespaces, control groups, network interfaces, file access controls, security profiles, and capabilities. RunC depends on *seccomp*, *SElinux*, or *AppArmor* for security policies. It requires a root file system and a configuration file containing the metadata to set up a sandbox and environment variables. Since runC relies on Linux kernel features, such as control groups for allocating computing resources and namespaces to provide isolation when creating containers, the host kernel is shared among all containers, as shown in Fig 1. This brings in security limitations and exposes runC to high severity vulnerabilities. There have been occurrences where a malicious container was able to break out from its isolated environment by overwriting the runC binary and gain root access to the host system [6]. This threatens the existence of containers and undermines their sole purpose of providing isolation.

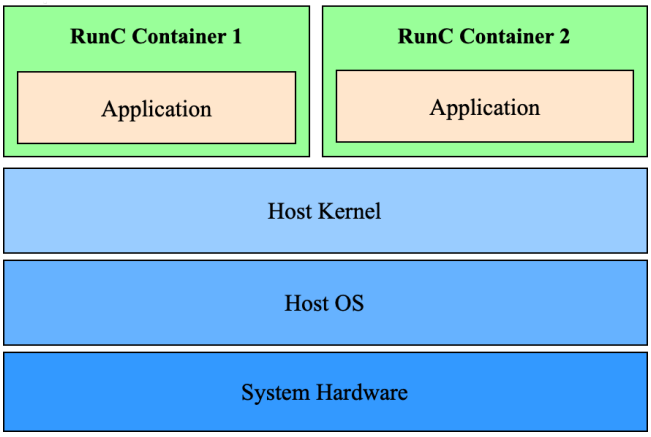

Fig. 1. Architectural representation of runC containers.

Kata container addresses these security issues by hosting the container inside a lightweight virtual machine. The Kata container project consists of two core components, the *kata-runtime* and the *kata-shim* which is used to integrate with the container managers like containerd. Kata container runtime follows the OCI specifications for container runtime but does things differently when it comes to creating an isolated environment. It uses a hypervisor to provide isolation by creating a lightweight virtual machine using QEMU/KVM for each container and then running the container inside the virtual machine [7]. For managing the containers, the Kata container project has introduced a daemon that runs inside the virtual machine called *kata-agent*. This agent uses libcontainer to manage the lifecycle of a container. Kata containers use a guest kernel to boot the virtual machine, which is highly optimized for quick boot time and minimal memory usage. Since the containers use a guest kernel instead of the host kernel, Kata container provides better-isolated environments as compared to the traditional containers and eliminates the security threats. Kata container uses the *qemu-lite* machine emulator and virtualizer to improve start time and memory footprint. Figure 2 shows an overview of the architecture of Kata containers. This architecture boasts about how it mitigates the security concerns of the traditional containers, but at what cost of performance? The next section talks about the performance metrics, system configurations, and methodologies used to evaluate the performance between runC and Kata container runtime.

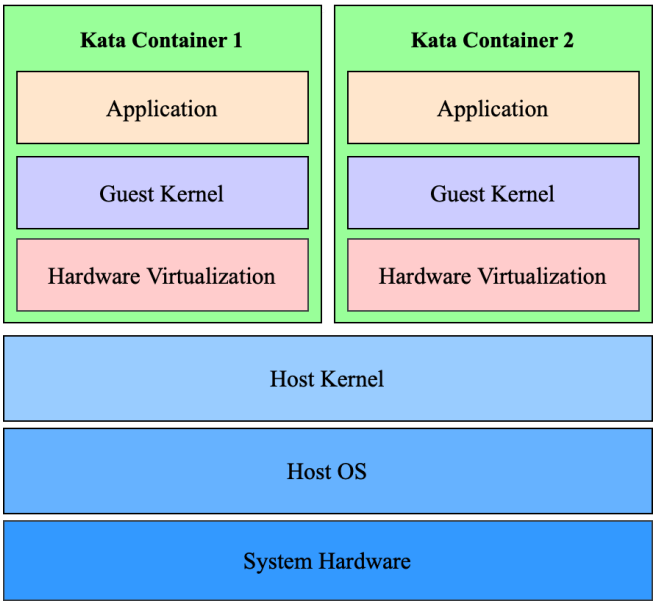

Fig. 2. Architectural representation of Kata containers.

III. Methodology

To conduct the experiments, a bare-metal system with support for hardware virtualization was used as a host, to create two identical virtual machines using VirtualBox. Each system was equipped with 4 cores of a 2.9 GHz Quad-Core Intel Core i7 CPU, 8 GB of physical memory and 10GB of a solid-state drive with support for nested virtualization. Ubuntu 18.04 LTS was installed on the systems.

The next task was to set up the environment for experiments. Since Docker adheres to the OCI specifications, Docker v2.2.0.3 was configured to work with runC and Kata container runtime on separate virtual machines. Docker already packages runC as their default low-level container runtime but it can be swapped with other OCI compliant runtimes. Kata runtime has a runtime pluggable architecture, which seamlessly works with Docker. Figure 3 shows a pictorial representation of the configuration.

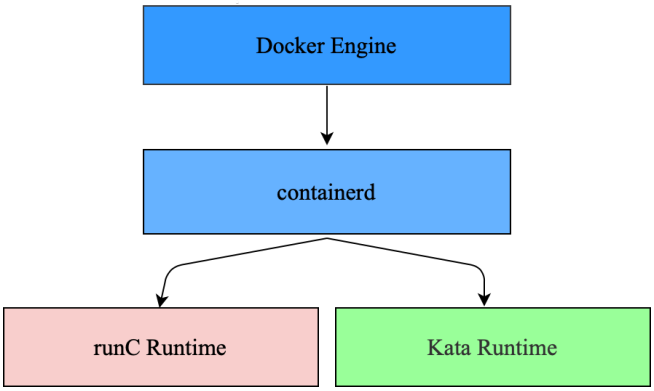

Fig. 3. Docker configured with runC and Kata runtime.

Focusing on the key indicators, this research work evaluates the performance of both the container technologies based on the container boot test, IO performance, CPU utilization, memory utilization and network performance.

IV. Performance Evaluation

A series of stress tests were performed to evaluate the performance of both machines. Each machine was rebooted prior to performing any test. The results are averaged over 15 runs. This section discusses the results obtained from the tests.

A. Container boot test

Container boot test measures the startup time, or the time taken to launch an app in a container. For this experiment, Apache *httpd 2.4.41* image was used with a static webpage hosted in the system. This image was chosen to maintain a standard benchmark environment. A script was written to perform the test and provide output with the average startup time based on 15 iterations. Table I shows the comparison between the startup time of both runtimes.

TABLE I. Average time of boot tests

Startup Time Results		
	Docker + runC	***Docker + Kata***
Average Time	0.633 seconds	3.707 seconds

The average startup time for the runC container is 0.633 seconds, whereas the average startup time for the Kata container is 3.707 seconds, which is significantly higher. This performance overhead in Kata container is due to the architecture, where a container runs inside a lightweight virtual machine. However, this result is impressive if compared to a virtual machine [8][9].

B. IO performance

To measure the IO performance, a very popular benchmarking tool *Bonnie++* was used [10]. Bonnie++ performs a number of hard drive tests and checks the file system performance. It benchmarks the file system with respect to read and write speed of data, the number of seeks and file metadata operations that can be performed per second [11]. Bonnie++ was configured to run tests with 2GB of file size, without write buffering. Table II shows the recorded output of Bonnie++ in both runtimes.

TABLE II. Bonnie++ benchmark results

Bonnie++ output					
File Size - 2GB	***Sequential Output (Block)***		***Sequential Input (Block)***		***Random Seeks***
	K/sec	***CPU %***	***K/sec***	***CPU %***	***/sec***
Docker + runC	1549371	89	4512265	99	14779
Docker + Kata	1103322	89	3748241	99	14841

Bonnie++ recorded an output rate of 1549.371 Mb/s in the runC container, whereas an output rate of 1103.322 Mb/s was recorded in the Kata container. This significant difference in speed is due to the inability of Kata container's virtual machine to interact directly with the hardware, on the other hand, runC containers have almost direct access to the system hardware. The results make it clear that runC performs better than Kata container, and the tradeoff to choose Kata container over runC is significant in the case of IO-intensive applications like databases.

C. CPU utilization

CPU utilization is the amount of time for which CPU was used to execute and process instructions. It can be used to estimate system performance. A heavy utilization with only a few running programs may indicate a system to be substandard. To measure CPU utilization, a Python based tool *psutil* [12] was used. Python system and process utilities is a cross-platform library for retrieving information on running processes and system utilization in Python. It is useful mainly for system monitoring, profiling, limiting process resources and the management of running processes [12]. Table III shows the CPU times of both runC and Kata container runtime machines.

TABLE III. Psutil CPU times of the system

CPU times output (sec)					
Machine	***User***	***Nice***	***System***	***Idle***	***IO wait***
runC	90.94	4.64	66.88	1368.96	15.84
Kata	97.13	306.54	46.8	988.99	19.95

Every attribute of Table III represents the seconds CPU has spent in the given mode. The description of every attribute is as follows.

- *User* shows the amount of CPU time spent in user mode executing the normal processes.
- *Nice* shows the amount of CPU time spent in non-realtime processes.
- *System* shows the amount of CPU time spent in kernel mode executing processes.
- *Idle* shows the amount of time CPU was not busy. Idle time actually measures unused CPU capacity.
- *IO wait* shows the amount of CPU time spent waiting for IO to finish.

This test shows a noticeable difference in CPU usage between both systems. CPU spends more time in the execution of user and non-real time processes in the Kata container. This CPU overhead is limited to Kata container due to the VM layer, which implies that the Kata container machine uses more CPU times compared to runC machine.

D. Memory utilization

Memory utilization shows the amount of memory used by a particular system at a certain unit of time. It depicts the statistics about system memory usage. Python system and process utilities was used to check the memory utilization. Table IV shows the memory utilization of both runC and Kata container runtime machines.

TABLE IV. Psutil memory utilization of the system

Memory utilization output (Megabytes)						
Machine	***Total***	***Used***	***Free***	***Active***	***Buffer***	***Cached***
runC	8348.545024	897.396736	4956.205056	1373.134848	116.535296	2378.407936
Kata	8361.545728	947.13856	5018.537984	1972.912128	200.159232	2195.709952

Every attribute of Table IV is expressed in Megabytes. The description of every attribute is as follows.

- *Total* shows the total amount of physical memory excluding swap.
- *Used* shows the total amount of memory that is being used.
- *Free* shows the total amount of memory that is readily available for usage.
- *Active* shows the amount of memory currently in use or very recently used.
- *Buffer* shows the amount of memory, which acts as a cache for things like file system metadata.
- *Cached* is the cache memory. This is used for improving the performance of system.

This test also shows a noticeable difference in memory utilization between both systems. The Used and Active values deduce that the Kata container has a higher memory footprint compared to the runC containers. Prior to each of the test, systems were rebooted. The result implies that Kata container machine uses more memory compared to runC machine.

E. Network performance

To check the network performance, Python system and process utilities was used to measure the network IO counter. Network IO counter shows the system-wide network I/O statistics. This test was performed after rebooting the machines. The webpage *https://www.w3schools.com* was accessed using curl. Table V shows the network stats of both runC and Kata container runtime machines.

TABLE V. PSUTIL NETWORK IO COUNTER STATS

Network utilization output				
Machine	***Bytes Sent***	***Bytes Received***	***Packets Sent***	***Packets Received***
runC	62076	1273228	766	857
Kata	68785	1280631	878	963

Since the result of the network IO counter for drop and errors were 0, these attributes were ignored. The description of every attribute is as follows.

- *Bytes Sent* is the total number of bytes sent from a system.
- *Bytes Received* is the total number of bytes received.
- *Packets Sent* is the total number of packets sent from a system.
- *Packets Received* is the total number of packets received.

According to the test, the packets sent by the Kata container machine were greater than the runC machine. In runC, the network bridge has one level of virtualization of the host networking stack through the docker daemon, whereas in the Kata containers the level of virtualization is greater due to the inclusion of the VM layer.

V. CONCLUSION

From the experiments performed, it is clear that runC performs better than Kata container runtime. Kata container boosts the security of a container but takes a hit at the performance due to its architecture. While there have been many adoptions to improve the performance, such as reusing QEMU machine protocol connection to speedup container creation, still, a lot of work needs to be done to be on par with traditional container performance. Keeping the sandbox and container structure in memory can avoid disk IO to reconstruct them. An improved memory trace and reduction of hypervisor footprint may improve the performance of Kata containers. On the other hand, with the growing user interest and widespread adoption of Docker containers, it will be interesting to see further development of Docker to introduce strict security measures.

REFERENCES

[1] Why Docker? (Webpage), https://www.docker.com/why-docker

[2] Kata Containers, https://katacontainers.io/collateral/kata-containers-1pager.pdf

[3] Three questions about Kata Containers (Webpage), https://opensource.com/article/18/4/kata-containers

[4] Docker Components Explained (Webpage), http://alexander.holbreich.org/docker-components-explained/

[5] Libcontainer source code on Github (Webpage), https://github.com/opencontainers/runc/tree/master/libcontainer.

[6] Aleksa Sarai, "CVE-2019-5736: Runc container breakout", https://www. openwall.com/lists/oss-security/2019/02/11/2, 2019.

[7] Kata containers on Github (Webpage), https://github.com/kata-containers/ kata-containers.

[8] A. Randazzo and I. Tinnirello, "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way," *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, Granada, Spain, 2019, pp. 209-214

[9] A. Lingayat, R. R. Badre and A. Kumar Gupta, "Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine," 2018 3rd International Conference on Communication and Electronics Systems (ICCES), Coimbatore, India, 2018, pp. 1019-1023.

[10] Bonnie++ (Webpage), https://www.coker.com.au/bonnie++/

[11] Preeth E N, F. J. P. Mulerickal, B. Paul and Y. Sastri, "Evaluation of Docker containers based on hardware utilization," 2015 International Conference on Control Communication & Computing India (ICCC), Trivandrum, 2015, pp. 697-700.

[12] Psutil (Webpage), https://giamptest.readthedocs.io/en/latest/