

Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way

Alessandro Randazzo, Ilenia Tinnirello

Department of Electrical Engineering

Università di Palermo

Palermo, Italy

Email: {alessandro.randazzo,ilenia.tinnirello}@unipa.it

Abstract—New coming applications will be only possible through Mobile Edge Servers deployed in proximity of the mobile users. Due to the user's mobility and server's workload, service migration will be an integral part of the services. For this reason, a standardized architecture should be designed to accomplish a workload migration in a secure and timely manner. Most research done to date has focused on the use of either virtual machine (VM) or container or a mix of both recently. A final solution might be an architecture only having the advantages of both technologies as the security of the VM and the speed of the containers. Custom solutions, actually, by using both technologies, need continuous optimization from case to case and not least, any solution is not standards-compliant. To this purpose, we present a novel architecture, Kata Containers, that is stewarded by the OpenStack Foundation (OSF), which supports industry standards including OCI container format, Kubernetes CRI interface, as well as legacy virtualization technologies. With this work, we highlight its state of the art and the motivations that may make it the right candidate for deployment of new MEC services. Additionally, we make a qualitative analysis against the most used runtime container, Docker runc, to show what features should be still improved or developed from scratch by future research.

Index Terms—MEC, Kata Containers, Docker, live migration

I. INTRODUCTION

Mobile Edge Computing (MEC) has become a hot topic in many kinds of research in the latest years. Reasons are related to many advantages and benefits that this new model will bring in cooperation with the new 5G mobile network. Main goals of MEC are : *'Mobile resources optimization through computation offloading'*, *'Data Transfer optimization from edge to core network through Big data analytic performed at the edge'*, *'Enabling cloud services bringing the hardware and software resources in the proximity of mobile subscribers '* and finally *'Context-aware services thanks to the RAN information'*.

Each of these goals brings with it different challenges. According to the Cisco Global Cloud Index [1] by 2021, within the consumer segment, social networking and video streaming will be the fastest growing applications and the biggest contributors, and thus, there will be an explosion of content-based applications in mobility. Therefore, a successful dynamic service migration, a.k.a. 'Live Migration' will play a key role. Several solutions have been proposed to live migrate a running service from a source to target MEC server. All of

these approaches have been focused on using either virtual machines (VMs) or containers. VM is a well-consolidated technology by offering more security and robustness than containers, but it is also true that needs to take a snapshot of the entire VM to replicate the service by causing both a slow movement and high storage space (as consequence a high bandwidth request) utilization. Since most of the services will require a very stringent delay and jitter constraints, the latest studies and solutions have been focusing on container technology that can guarantee a faster and lighter migration than VMs. The most widely deployed container platform is Docker [2] [3] that has become a standard de facto in the industrial cloud. One of the latest contributions is given by [4] in which authors focus on reducing the file system transfer size by leveraging Docker's layered storage architecture. A key point of this work is to propose two-layer system-wide isolation for better security during the offloading service. Authors in [4] point out the importance of applying for a defence-in-depth security approach [5] via two layers of the system virtualization hierarchy. This layered security model is a combination of VMs and containers in which the services are, first, isolated running them inside different containers and later further isolated them inside different VMs. This solution further opens the gates to evaluate new technologies that might assure both performance and security in the live migration and not only. In this context, a very new emerging technology, named *Kata Containers* [6], might cover the gap between security and speed and pave the way for a standard solution rather than a customized one. To the best of the authors' knowledge, this is the first paper that contributes to introduce the key aspects of this architecture and compare them with the Docker container platform. More precisely, we make the following contributions:

- 1) Introducing Kata Containers architecture with the aim of letting it know to the scientific community
- 2) Qualitative analysis to steer future researches in improving or developing missing features

The rest of the paper is organized as follows: §II briefly describes the Docker architecture and its main security concern. §III introduces the Kata containers architecture from different points of view. §IV compares, from the quality points

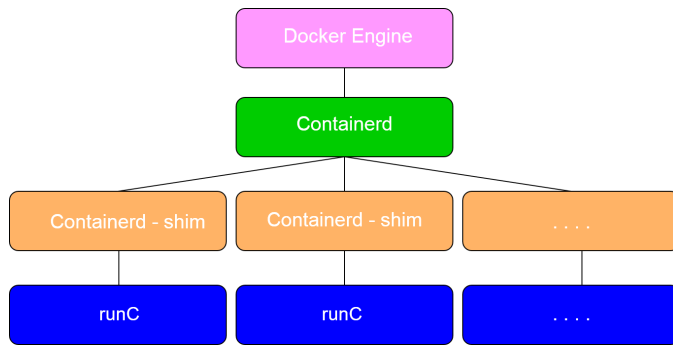


Fig. 1: Docker Engine since 1.11 version.

of view, many capabilities between Docker runc and Kata-runtime to take a picture of its state of maturity, followed by the conclusions in Section §V.

II. BACKGROUND

In this Section we give a high-level overview of Docker engine by introducing its main components and explaining one of the main potential security issues. This explanation is necessary to better follow Section §III.

A. Docker Overview

Docker is an open source software platform to create, deploy and manage virtualized application containers on a common operating system (OS). First Docker's version, released in 2013, was a monolith block including both high-level and low-level runtimes. In 2015, under the push of the Open Container Initiative (OCI) [7], an open container project whose purpose is creating open industry standards around container formats and runtime, Docker introduced the 1.11 engine version. This release involved a massive refactoring of the engine to make it OCI compliant runtime. In detail, since 1.11 version Docker engine is broken down in four components :

- 1) **Dockerd**: a daemon listening for Docker API requests coming from the users through the Docker CLI.
- 2) **Containerd**: a daemon to control runc by managing a full container lifecycle (images, storage, execution by calling runc, network).
- 3) **Containerd-shim**: a demon that allows running daemon-less containers. Thus, Containerd-shim allows exiting either runc or any another runtime container OCI-compliant after starting the container.
- 4) **Runc**: is a command line client for running containers according to OCI format.

From the virtualization point of view, Docker engine, as well as any other technology based on containers, sits on top of the host operating system (OS) and all containers applications share its Kernel. An evident gain in this architecture, as opposed to VMs, is that there is no extra overhead due to the OS guest. Main benefits are a high reduction of the size of snapshots, faster boot up and migration. Figure 1 shows the Docker engine architecture since 1.11 version previously described.

B. Docker Security Concern

Despite being the containers the natural candidates for new MEC applications, they have been suffering from potential security risks still open. Due to the sharing of the OS kernel with all running containers, one of biggest security fears about containers is that an attacker, through a malicious program, could escape from the sandbox and hijack the host system. It would cause the overall compromise of all containers/services managed by the host's kernel. Figure 2 shows the compromise of all host OS services. A very recent bulletin from Common

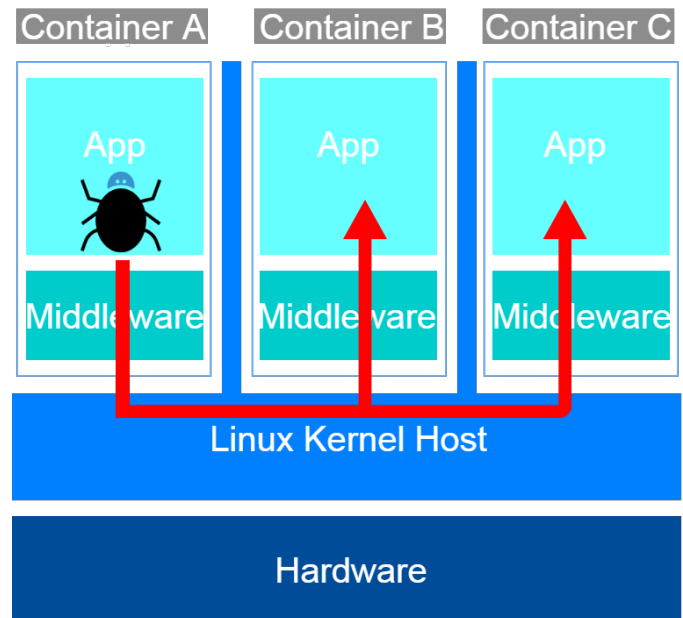


Fig. 2: Security Problem in Docker : Hijacked Kernel Host.

Vulnerabilities and Exposures (CVE), dated March 2019, has announced a Docker security flaw that allows a malicious container to overwrite the host runc binary and thus gain root-level code execution on the host [8]. This flaw occurs because of file-descriptor mishandling in runc. The high gravity of this bug is that besides runc (Docker), this flaw can also hijack container systems using LXC [9] and Apache Mesos [10] container code. In other words, most, if not all, cloud container systems are vulnerable to this potential attack. This security flaw let us understand that only the containers cannot be the full solution for supporting next coming MEC applications, and we urge focusing on a more complex and standardized architecture.

III. KATA CONTAINER

After a brief introduction where Kata Containers comes from, we present several architectural aspects like networking, storage, devices and interfaces.

A. Architecture

Kata Containers project comes from the merging the two projects 'Intel Clear Containers' and 'Hyper.sh runV' in December 2017 whose aim is building extremely lightweight

VMs that perform like containers, but provide the workload isolation and security advantages of adding a virtual machine layer. Kata Containers community released the first version 1.0 in May 2018, and the latest release is 1.7.0 landed in May 2019. Being Kata Containers runtime OCI-compliant, it is seamlessly pluggable with the Docker engine and any other container management platform as Kubernetes [11] and OpenStack [12] by simply replacing runc with Kata-runtime. Figure 3 shows how much easy is to integrate Kata Containers into industry standards including OCI container format, Kurbenetes CRI interface, as well as legacy virtualization technologies.

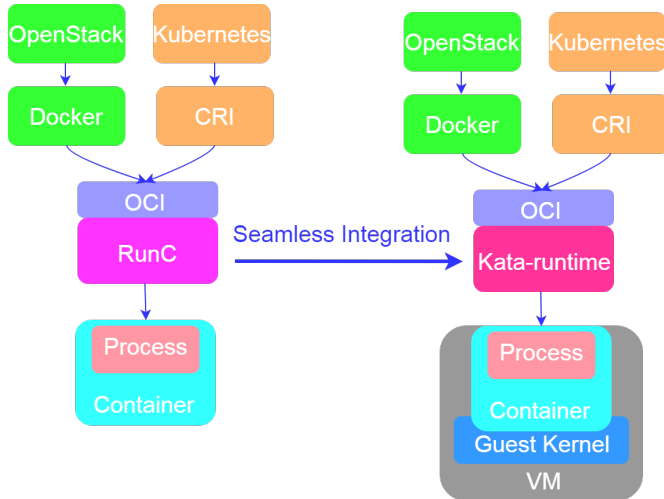


Fig. 3: Kata Containers Integration.

The software architecture includes four main components:

- 1) **Kata-runtime:** is an OCI compatible container runtime and has the role of handling all commands specified by the OCI runtime specification, e.g. invoking the hypervisor for creating a lightweight and fast VM for each container or pod, and launching kata-shim instances.
- 2) **Kata-shim:** is the process that runs on the host by managing all container input/output streams. It acts as though it is the container process (actually running into the VM) making it so visible to the container process reaper (located in the host) that can, thus, monitor it. This component is mandatory to be oci-compliant.
- 3) **Kata-proxy:** is a process offering access to the VM Kata-agent to multiple Kata-shim and Kata-runtime clients associated with the VM. Its main role is to route the I/O streams and signals between each kata-shim instance and the Kata-agent. All gRPC [13] requests are multiplexed by yamux [14].
- 4) **Kata-agent:** is a process running in the guest inside the VM. Its role is to set up the environment for managing containers and processes running within those containers.

The current software architecture is simplified by merging Kata-shim, Kata-proxy and Kata-runtime in one component, named **containerd-shim-kata-v2**. The communication protocol is gRPC via vssock interface among containerd-shim-kata-v2, hypervisor and Kata-agent. This solution avoids using the

proxy component because vssock interface can multiplex all gRPC requests. Figure 4 shows the Kata software architecture.

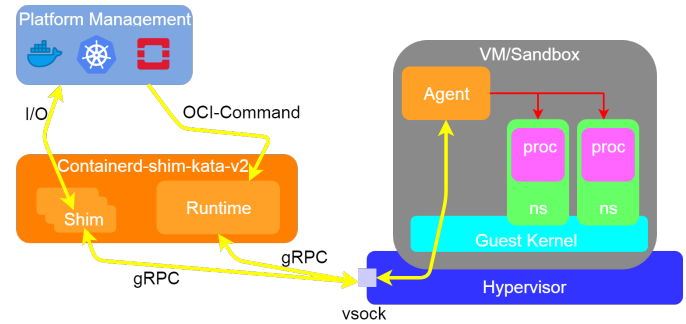


Fig. 4: Kata Architecture

For a better understanding of how Kata components interact with each other, let us suppose to create a new container. Fig 5 shows the exchanged messages' sequence among the main Kata containers components.

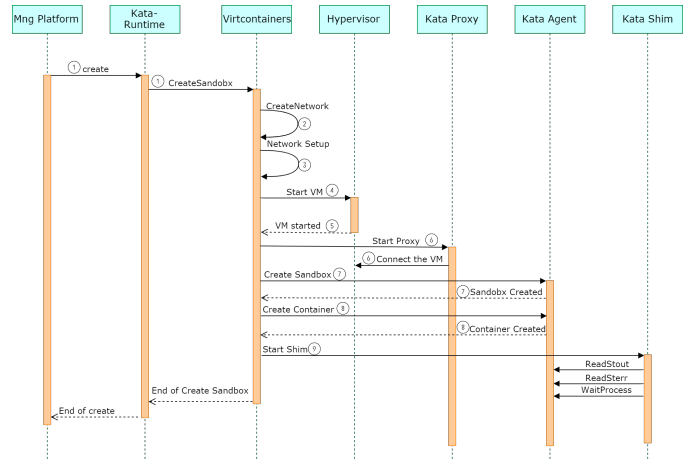


Fig. 5: Kata Containers Creation

Upon receipt of a 'create' command from a management platform (it could be Docker, Kubernetes or others), ① Kata-runtime calls a procedure of 'virtcontainers', which is a generic, runtime-specification agnostic, hardware-virtualized container library, to create a 'sandbox' which is an environment where a set of containers sharing the same networking namespace and storage. In detail, on behalf of Kata-runtime will create and start some network setup operations like: ② ③ 'create' the network guest namespace (guest netns), the *veth* interface pair to link the host network namespace (host netns) with the guest one, the 'TAP' interface needed for linking the VM with the 'veth' interface internal to the 'guest netns' through a 'MACVTAP' connection. Then, a call to the hypervisor ④ and ⑤ will get the VM started in the 'guest netns' by providing the 'TAP' interface. Next step is to start the Kata-proxy ⑥, which will connect to the freshly created VM. (Please note that this component is not needed by using a recent kernel version supporting the new *vssock* feature).

As previously described, the main task of the Kata-proxy is to connect the Kata components located at the 'host netns' with the VM in the 'guest netns'. Next step is to create the container/POD inside the VM ⑦ and ⑧. This operation is done by the Kata-agent that sets up the environment based on the configuration file and spawns the container process. Finally, Kata-runtime gets Kata-shim started ⑨ which will connect with the gPRC server to directly monitor and check the container process status from the 'host netns'. Kata-shim is one per container process and is mandatory because Kata-runtime is a transient component, namely once created the container its process terminates and exits. As we mentioned earlier, one of the most important features of Kata containers is the double security level introduced by running the container process in a lightweight and fast VM whose guest kernel is isolated both from the kernel host and from other containers encapsulated in their own VM. Therefore, the natural question is:

'How does Kata make a VM lightweight and fast?' Many optimizations in the hypervisor, VM and communication between device and VM itself are done to get near-container performance. Regarding the hypervisor, Kata architecture can support multiple hypervisors and uses QEMU/KVM by default. It actually uses a specialized QEMU version named qemu-lite [15] to improve boot time and reduce memory footprint that comes with some features like:

- 1) **Machine accelerators (MA):** are architecture specific for improving the performance. One of these is NVDIMM [16] based on x86 architecture working in direct access storage (DAX) mode. It allows sharing the host rootfs in read-only mode as a persistent memory device to the VMs.
- 2) **Kernel same-page merging (KSM):** KVM feature that allows sharing identical memory pages amongst different VMs, i.e. deduplicating memory to maximize container density on a host.
- 3) **Hot plug Devices (HPD):** VM starts with a minimum amount of resources for faster boot time and very small memory footprint size. The VM as soon as needs more resources, the hypervisor hotplugs the devices on the fly.
- 4) **Fast Template (FT):** Pool of pre-configured lightweight VMs ready to quickly deploy.

VM-side optimizations are:

- 1) **Guest Kernel minimal (GKM):** Highly optimized for kernel boot time and minimal memory footprint, providing only those services required by a container workload.
- 2) **Guest Image (GI):** is a minimal operating system (mini O/S) highly optimized for container bootstrap system. This is possible since the mini O/S has only two running services at startup; systemd and Kata-agent. The former runs Kata-agent, while the latter creates the container environment.

Finally, the communication between VMs and devices is even highly optimized by hardware passthrough (HWP) technique that gives a VM direct access to a host device, e.g. pci network

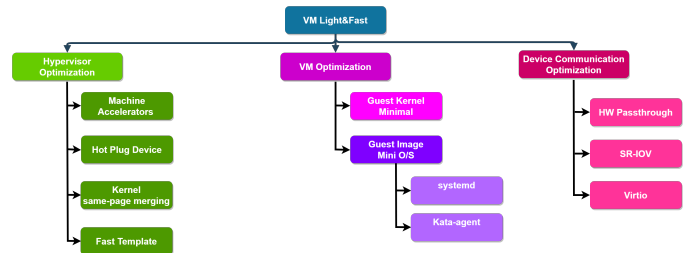


Fig. 6: Kata Optimizations

card, by getting near-native performances. Other kinds of communications are single root input/output virtualization (SR-IOV) [17] that allows a PCIe device to appear to be multiple separate physical PCIe devices and VirtIO interface [18] which allows virtual machines access to simplified "virtual" devices, such as block devices, network adapters and consoles. Figure 6 summarizes all described capabilities that make the VM fast and light like a container.

B. Networking

Generally speaking about networking containers, the container engine, located in the host networking namespace, sets up a container networking namespace, for each guest VM, that is isolated from the host network, but which is shared among containers. Additionally, It creates a virtual ethernet (veth) pair to allow the communication between host and container namespace. One end in the host networking namespace and the other end in the container namespace respectively. This kind of networking is namespace-centric and many hypervisors, like QEMU, can only handle a different interface, named 'TAP', for VM connectivity that is incompatible with the 'veth' one. Thus, being the container also encapsulated in a VM cannot communicate with the host system. Kata-runtime transparently connects 'veth' interfaces with 'TAP' ones using MACVTAP [19] driver to overcome this networking problem. From the traffic monitoring point of view is also possible to control the container network traffic through a traffic mirroring module placed in the middle between veth and tap interface in the container networking namespace. Finally, Kata-container supports both container networking model (CNM) adopted by Docker, and container networking interface (CNI) adopted by Kubernetes and Podman. Both models provide the freedom of selection for a specific type of container networking, can join one or more networks and use multiple network drivers concurrently. Figure 7 summarizes the network connections between host and container network namespace (via veth pair) and between host network namespace and container (via MACVTAP).

C. Storage

Kata containers can manage the storage both at the file level and at the block level. In the first case, it makes use of '9pfs' [20] a network filesystem protocol which allows mounting the rootfs image in cold-plug mode, while at the block level,

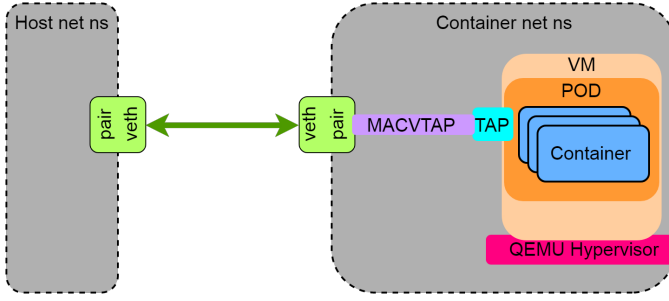


Fig. 7: Kata Networking

can mount volumes in a hot-plug way by using a 'device-mapper' [21] driver. Device-mapper has better performances than 9pfs but has some compatibility issue with recent versions of Docker.

IV. AN OVERALL QUALITATIVE COMPARISON: RUNC VS KATA-RUNTIME

After introducing Kata Containers architecture and described all its main features is time to present a qualitative comparison with the Docker container runtime runc. This work is based on the current limitations and issues still open in Kata-runtime, but also considering other aspects already well consolidated such as security, performance, resource management and networking. To do so, we define a five-grade quality scale (*poor*, *satisfactory*, *good*, *very good*, *excellent*) for assessing several aspects of the two runtime containers. Table I shows for each grade its description and assigned quality point score. We have considered 10 aspects divided by pending features, security, performance, network limitations, resource management and sharing and integration capabilities. Figure 8 shows the results of the following qualitative analysis on a radar chart. The first aspect was the possibility to perform a 'live migration'. At the time of writing, Kata-runtime is not able yet to restore and checkpoint a workload to perform a live migration. For that reason, we scored 1 to Kata-runtime and 3 to runc. Regarding the 'security model' Kata-runtime excels

TABLE I: Five-grade quality scale

Grade	Description	Quality Point
Poor	The feature is not implemented yet.	1
Satisfactory	The feature is very elementary and needs many improvements.	2
Good	The feature has many strengths points, but some still need to improve.	3
Very Good	The feature is almost perfect in any aspect.	4
Excellent	The feature is at the top of its implementation with the highest levels of performances.	5

on runc thanks to a double security layer and thus, we scored 5 and 3 to runc. We also considered the 'security options' like SecComp [22], AppArmor [23] and SELinux [24] that may be enabled. Since Kata-runtime only supports [22] at the moment,

Qualitative Analysis: runC vs Kata-Containers

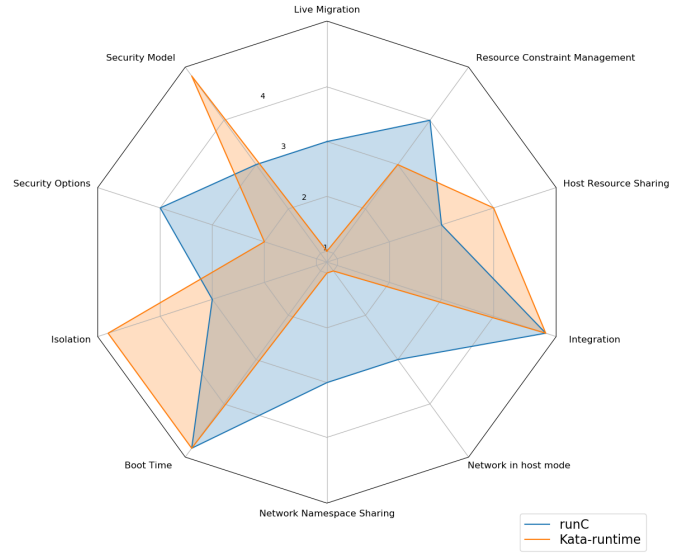


Fig. 8: Qualitive Comparison: runC vs Kata Containers

for this reason, we scored 2 and 4 for runc. In terms of 'isolation' we scored 5 to Kata-runtime and 3 to runc, the main reason is that the Kata Containers architecture is more secure than Docker and avoids hijacking of the kernel host in case of root-escalation of a container. 'Performance', seen as boot time, is almost equal in both architectures on condition that VM has started with one virtual CPU (VCPU) and in case of more computational resources can hot-plug others. Therefore, we scored 5 for both architectures. 'Integration' in container management platforms is excellent, being OCI-compliant both and, thus, we both scored 5. We also compared some current 'network limitations' in Kata-runtime. The first is that Kata-runtime does not support the option to join another container namespace and, thus, may not share a common 'network namespace' and the network interfaces placed in the network namespace. For that reason, we scored 1 to Kata-runtime and 3 to runc. The second limitation is that Kata-runtime does not support the possibility to access the 'host networking configuration' directly from within the VM. It is worth noting that this option actually can be activated, but is recommended not to do it since could break the host networking setup. On the basis of these considerations we scored 1 to Kata-runtime and 3 to runc. Regarding 'host resource sharing', runc can start a container in privileged mode accessing host devices, while Kata-runtime also supports this option, but in this case, gets full access to the guest VM devices. Although this option gets seen as a limitation, from a security perspective is indeed a plus because in the case of root-escalation of the container could not poison the host kernel. Therefore, we scored 4 to Kata-runtime and 3 to runc. Finally, we also considered 'resource constraint management' that is simpler in runc than

Kata-runtime. Indeed, runc only uses *control groups* (cgroups) to limit, prioritize, control and account the resources, while for Kata-runtime, due to the double security layer, it might be necessary, for getting the same result, to apply the constraints to multiple levels. So the resource constraint management is coarse-grained in runc, while is fine-grained in Kata-runtime. For this greater complexity, we scored 4 to runc and 3 to Kata-runtime.

V. CONCLUSIONS

In this work, we introduced a very new emerging architecture, named Kata Containers, that brings the advantages of both VMs and containers in one solution. Indeed, this architecture comes up with a hard security model provided by hardware virtualization technology of the VMs and the great performances as the speediness of the containers. Our goal was to let this new technology know to the scientific community as much as possible since we do believe that Kata Containers may become a standard solution for deployment the coming up MEC services in the next near future. As cited in Section §I, [4] figured out the importance of designing an architecture that is a mix of VM and container for enabling the live migration of MEC services. Indeed, if on one side the containers may guarantee some stringent requirements such as delay, response time typical of augmented reality, gaming online and data analytics, from the other for the security has still some problem due to the sharing of the kernel host with all containers. On the contrary, VMs are much more secure thanks to their hardware virtualization, but their overhead makes them slow to satisfy the requirements of the new MEC services. Here then, Kata Containers takes the best of both technologies to deliver a very secure and fast solution. Other non-technological aspect that makes us believe Kata Containers may be the right solution is that an open-source community stewarded by the OpenStack Foundation (OSF) supported by the most important leading companies in cloud environment and hardware infrastructure sector, such as aws Amazon, Microsoft, Google Cloud, Intel, Huawei, ZTE, Dell EMC and many others. We, thus, faced several key-points of Kata Containers from the architecture's perspective and also compared them to the most popular container runtime Docker by considering the current limitations and some issues still open either. Our qualitative analysis showed that security, performances, integrability are already at the top level, while other capabilities like workload's migration, resource sharing still need to be developed or improved. To conclude, we recap the main key features of Kata Containers in Figure 9. A potential future work will be a quantitative analysis to benchmark Kata containers against Docker runtime.

REFERENCES

- [1] Forecast and Methodology, 2016/2021 White Paper [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html>
- [2] Why-docker [Online]. Available: <https://www.docker.com/why-docker>
- [3] Container Adoption Benchmark Survey pag. 13 [Online]. Available: https://diamanti.com/wp-content/uploads/2018/07/WP_Diamanti_End-User_Survey_072818.pdf






Symbol	Feature	Description
	Security	<ul style="list-style-type: none"> • Double Security Layer • No hackable Kernel Host
	Integration	<ul style="list-style-type: none"> • Seamless integration with many OCI-compliant container management systems
	Isolation	<ul style="list-style-type: none"> • Each workload runs inside its own VM • Possibility to differentiate the Kernel per-workload/Tenant
	Performance	<ul style="list-style-type: none"> • Resource Efficiency <ul style="list-style-type: none"> ◦ Minimal Memory Footprint ◦ Optimal CPU utilization • I/O Optimization • Performance as standard Linux Container
	Use Case	<ul style="list-style-type: none"> • Suitable for deployment within a multi-tenant untrusted environment • Potential solution for many MEC use cases where security&speediness is a must

Fig. 9: Kata Containers Key Features Recap

- [4] L. Ma, S. Yi, N. Carter, and Q. Li, Efficient Live Migration of Edge Services Leveraging Container Layered Storage, IEEE Transactions on Mobile Computing, 2018.
- [5] Defense in depth [Online]. Available: [https://en.wikipedia.org/wiki/Defense_in_depth_\(computing\)](https://en.wikipedia.org/wiki/Defense_in_depth_(computing))
- [6] The speed of containers, the security of VMs [Online]. Available: <https://katacontainers.io/>
- [7] Open Container Initiative [Online]. Available: <https://www.opencontainers.org/>
- [8] CVE-2019-5736 [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736>
- [9] Linux containers [Online]. Available: <https://linuxcontainers.org/>
- [10] Apache Mesos [Online]. Available: <http://mesos.apache.org/>
- [11] Kubernetes [Online]. Available: <https://kubernetes.io/>
- [12] OpenStack[Online]. Available: <https://www.openstack.org/>
- [13] Open-source universal RPC framework [Online]. Available: <https://grpc.io/>
- [14] Yamux (Yet another Multiplexer) [Online]. Available: <https://github.com/hashicorp/yamux>
- [15] QEMU [Online]. Available: <https://github.com/kata-containers/qemu/tree/qemu-lite-2.11.0>
- [16] NVDIMM [Online]. Available: <https://en.wikipedia.org/wiki/NVDIMM>
- [17] SR-IOV [Online]. Available: https://en.wikipedia.org/wiki/Single-root_input/output_virtualization
- [18] Virtio [Online]. Available <https://wiki.osdev.org/Virtio>
- [19] MACVTAP [Online]. Available <https://virt.kernelnewbies.org/MacVTap>
- [20] File System 9pfs [Online]. Available <https://www.kernel.org/doc/Documentation/filesystems/9p.txt>
- [21] Device Mapper [Online]. Available https://en.wikipedia.org/wiki/Device_mapper
- [22] Secure Computing with filters [Online]. Available https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
- [23] AppArmor [Online]. Available <https://wiki.archlinux.org/index.php/AppArmor>
- [24] SELinux [Online]. Available <https://wiki.archlinux.org/index.php/SELinux>