National College of Ireland

Performance Evaluation of Various Container Runtimes and Process ID Based Escape Detection

MSc Research Project
Msc In Cloud Computing

Jogindersingh Ramani
Student ID: 22129588

School of Computing
National College of Ireland

Supervisor: Dr. Rashid Mijumbi

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Jogindersingh Ramani
Student ID:	22129588
Programme:	Cloud Computing
Year:	2023/2024
Module:	MSc Research Project
Supervisor:	Dr. Rashid Mijumbi
Submission Due Date:	14/12/2023
Project Title:	Performance Evaluation of Various Container Runtimes and Process ID Based Escape Detection
Word Count:	6998
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Jogindersingh Ramani
Date:	13th December 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Performance Evaluation of Various Container Runtimes and Process ID Based Escape Detection

Jogindersingh Ramani
22129588

Abstract

Due to various benefits like portability and resource efficiency, containers have become a fundamental technology for cloud-native applications. However, vulnerabilities like a shared host kernel and potential image flaws still pose security risks. This research focuses on evaluating leading open-source container runtimes like Docker, Kata Containers, Gvisor, Quark Containers, and Youki across metrics like performance and container lifecycle management. Its data-driven evaluation will help organizations with detailed insights to select container runtimes based on their workload. Along with that, a novel method is proposed to detect container escape by analyzing process ID namespaces and process IDs to identify unauthorized access attempts from containers to hosts. Multiple test cases validate the proposed technique to detect escapes in the container environment. The results will contribute to detecting container escapes even before deployment, as the proposed approach can also be integrated with Jenkins. Overall examination shows that quark containers demonstrate excellent lifecycle speed despite heightened isolation, while the proposed methodology accurately captures the containers that have escaped to the host. Lastly, the organization should balance performance security and complexity when choosing optimal runtimes for cloud deployments.

1 Introduction

The origin story of containers goes back to the late 1970s when IBM's VM370 virtual machine system was made for mainframes Creasy (1981), which allowed multiple VMs with their own OS. Following that, in the Nineties, Solaris Zones and FreeBSD Jails advanced virtualization Price and Tucker (2004) came into the market that were more effective when compared to VM/370. Then in 2013, Docker introduced containers, which revolutionized the software industry. A container is a lightweight, standalone executable package that contains all the essentials that are required to run an application portably in an isolated environment in Docker (2020). It was quickly adopted to build, run, and distribute encapsulated applications for easy deployment, transportability, resilience, and security. Major organizations and cloud providers like Google, Amazon, and Red Hat adopted Docker and rolled out modern technologies like ECS, Kubernetes, and OpenShift.

Containers became the fundamental building blocks for cloud-native applications due to:

- Agile development benefits from portable, immutable images.
- Microservices architecture using separate containerized services.

- Dynamic cloud infrastructure requiring scalable and movable deployment.
- DevOps culture that needs consistent environments.

In recent years, technology has been growing at an exponential rate, and in the fast-paced landscape of containerization, selecting an appropriate container runtime plays an important role. This is because most organizations have already been moving towards micro-services, and the use of containers complements the micro-service architecture. Thus, selecting the appropriate container runtime is essential for an organization, as its performance and security are based on it. So to address that challenge, it is vital to conduct a detailed performance evaluation of various container runtimes like Docker (runc), Gvisor, Kata Container, Youki, and Quark Container, which are currently present in the market. Also, some of this runtime can play a significant role as it has the potential to be used on Fog and Edge servers and also in Web assembly, which is a current trend in the market.

When a container is talked about, the first thing that people think about is Docker. This is because it is used everywhere in software development. Other container technologies, like gVisor, have most of the same features as Docker, but they focus more on improved security and give a unique approach to isolation between containers. Kata Container combines the benefits of virtualization and containers, which stands out as a promising solution in the market Randazzo and Tinnirello (2019). Quark Container also focuses on virtual machine-level isolation, security, and simplicity. Youki, with its user-friendly features, caters to the evolving needs of containerized workflows. So selecting these various container runtimes across various verticals will provide detailed insights when an organization is thinking of selecting container runtimes for their workload.

As technologies evolve, they face various threats and risks from attackers who are constantly looking to gain access to the system. So as a result, containers can still be vulnerable to security breaches through weaknesses in the runtime, application, or image. A 2017 Docker Hub breach showed that malicious images could lead to security risks when downloaded. So, it is essential to secure containers since most of the mission-critical workloads are run on containers. Among the various threats that containers face in the containerized environment, the most common are container escapes **?**. It means that the attackers try to gain unauthorized access to the host through a container application or workload. So a breach of this isolation can lead to a compromise of data, system degradation, or even a complete takeover. So detecting container escapes has become one of the most important aspects of security in a containerized environment. This motivates a **research question:** Evaluating various container runtimes like Docker, Kata Container, gVisor, Youki, and Quark Container in terms of key performance metrics while concurrently implementing novel solutions to detect container escapes?

Potential contributions that this study will make:

- The examination of leading open-source container runtimes encompassing Docker, Kata Containers, gVisor, Quark Containers, and Youki.
- Inform container runtime selection for organizations based on security and performance data.
- A deeper understanding of various runtimes, like startup time and resource utilization, can be used to decide where critical applications should be deployed.

- Enhancing container security by detecting container escape through the process running on the host and container.

In summary, containers evolved from early virtualization and are now fundamental but can still be vulnerable. So evaluating various container runtimes can provide important insights to the organization, and detecting container escapes will help enhance security.

2 Related Work

There are several advantages that come along with cloud computing like scalability and on-demand services but along with that it also brings shared maintenance and security duties of both service providers and end users. The flexibility that cloud computing brings is by permitting multiple clients to access the same physical infrastructure by virtualization of hardware resources. However, old virtualization techniques have their own drawbacks in expenses, configuration complexity and limits to scaling. These obstacles were overcome through building application logic and runtime into portable, isolated packages by containerization which completely changed that technological industry. The reason due to which containers have become popular is because of deploying applications in a virtualized way. Runtimes like Docker, Containerd, and Kata Container are widely adopted in the industry. Even though containers provide various advantages in mobility and resource efficiency, it also introduces new security risks due to reliance on a shared kernel and potential vulnerabilities in container images.

During the literature review phase, I have examined various academic papers written by various researchers in the field. The goal of the literature review is to gain understanding of previous work done related to containerization technology and how those observations could help in my own research. Since the initial release of docker in 2013, many container images containing vulnerabilities have been uploaded to the docker hub which is the docker public repository. Malicious actors have taken the advantage of this open repository as an opportunity and distributed the compromised images which enables containers attacks like escalation of privileges, data theft, and crypto jacking when the container using this image were launched (Abhishek and Rajeswara Rao 2021; Brady et al. 2020). Additional security risk was present due to misconfigured container runtimes and networking allowing easy path for attackers to attack like inadequate access controls permitting insider threats, and weak isolation controls that allow breakouts across containers running on the same host Sultan et al. (2019). While some of the monitoring solutions that are available in the market are used for detecting known exploit attempts, it cannot prevent zero-day attacks German and Ponomareva (2023). A proactive security approach would involve hardening the container images and host, scanning of vulnerabilities in the environment, minimizing the attack surface area and restricting unnecessary capabilities Brady et al. (2020).

2.1 Kernel Sharing Issues

Containers have become prominent as an effective way to deploy applications by virtualizing resources at the operating system level. However, the main security concern that comes with containers is that all the containers use the same host kernel(Sultan et al. 2019; Abhishek and Rajeswara Rao 2021). This violates the protocol of least access and gives a larger surface attack to the attackers. So, the host kernel becomes a tempting

target for attackers since compromising allows a complete bypass of any isolation between containers. So, the advice given by experts is to regularly patch the kernel German and Ponomareva (2023), but this alone cannot protect the host from zero-day exploits by the attackers who are continuously targeting the kernel code Flauzac et al. (2020). Any kernel bugs or issues can be abused to escalate the privileges, log into multiple containers, and even gain access to the containers Brady et al. (2020). While the namespace and cgroups and their capabilities give some level of separation, the underlying kernel security is still a major concern. So, it is important to implement additional safeguards to restrict kernel access when working with containers. To tackle this issue, host hardening is advised with a specific container runtime and best practices for kernel-based risks.

The underlying shared kernel remains a major vulnerability, even when container security measures like namespaces and control groups that provide some level of security boundaries are in place. The broad access enabled by the shared kernel leaves containers susceptible to many potential exploits like escalation of privileges, lateral movement between containers, data leakage through /proc and /sys files, and denial of service through resource exhaustion attacks (German and Ponomareva 2023; Flauzac et al. 2020). Even with the container security measures in place, like dropping capabilities and seccomp syscall filtering, the critical kernel flaw can still be exploited to break out of the container, as demonstrated by the frequency of container escapes in the wild (Jian and Chen 2017; Combe et al. 2016). On top of being a favorable target for attackers, the large monolithic kernel design also lacks the fine-grained isolation to securely separate the containers.

Containers still face an ongoing struggle to achieve robust isolation of the kernel to reduce the attack surface while maintaining performance Abhishek and Rajeswara Rao (2021). Techniques like hardening and regularly patching the host kernel help improve security, but they are not sufficient to protect against zero-day attacks. Methods that can provide partial mitigations like virtualization, mandatory access control policies, and restricted capabilities are advised by experts Rangnau et al. (2020), but they can induce some overhead. Most critically, a comprehensive examination of the kernel isolation techniques used in containers—including namespaces, control groups, capabilities, etc.—has yet to occur Yang et al. (2021), and their efficacy against the targeted attacks is uncertain. The basic model of sharing the host kernel challenges the assumption that containerized processes will remain separate from the underlying host. Further research is needed to re-establish kernel isolation for containers with acceptable performance. So the core kernel sharing approach contradicts the container security of segregating processes from the host itself. Improving container security requires reevaluating the fundamentals of core isolation due to performance limitations.

2.2 Container Image Vulnerabilities

There has been historical evidence that the major security flaws in the shared kernel were successfully exploited by the attacker to break out of the container and compromise the underlying host. From these previous events, it clearly indicates that the isolation protections that are used to segregate containers are not performing as robustly as intended. For example, the vulnerability known as Dirty COW [1] was a race condition bug that permitted an attacker to overwrite read-only memory mappings. This could be used from within a container sharing the host kernel to modify any file owned by the root user

[1] `https://nvd.nist.gov/vuln/detail/CVE-2016-5195`

on the host itself. The Dirty COW was leveraged to overwrite the host's runc binary, execute commands as root, and fully escape the container sandbox. Similarly, other kernel attacks, like CVE-2021-4034 [2], also allowed the attackers to breach the container. So these real-world container escape attacks that target kernel vulnerabilities showcase the severity of relying on shared kernel security. The ability to break out of container containment through kernel-level attacks demonstrates that current isolation controls between containers and hosts are insufficient. Improving container security requires reevaluating basic isolation assumptions against the risks of core sharing.

2.3 Studies on Container Runtime Environment

The default container engines like runC have inherit kernel sharing issues which leads to isolation limits between containers and hosts. So, new methods are needed for robust segregations that can tackle existing kernel-sharing issues. Technologies like gVisor and Kata Containers use additional sandboxing between containers and hosts. Kumar and Thangaraju (2020) discuss, Kata Containers leverage lightweight virtual machines to isolate processes and kernels via hardware. However, virtualization can reduce efficiency Kumar and Thangaraju (2020) found Kata Containers had slower startup, higher CPU use, and lower I/O than native Docker. Other approaches also aim to strengthen container isolation. This previous research gives critical insights into selecting appropriate container runtimes based on specific security and performance requirements.

2.4 Research gap.

While the current literature which is reviewed above focuses on identifying the security challenges in containers like container escape and proposed the solution by implementing various monitoring tools and security tools. However, it lacks in answering ways to detect container escape which this research fulfills by detecting the container escape by the means of pid namespace and other mandatory checks. Also in the previous research container runtime evaluation is only done on docker, gVisor, and classic kata container runtimes. As new emerging runtimes like quark containers, youki, and kata containers are introduced in the market which follows technology trends like wasm and IOT there is an opportunity to contribute to the domain by evaluating these runtimes on various metrics.

So, quantitative comparison of various runtimes, including Docker, can help inform decisions based on specific isolation needs and complexity tolerance, directly addressing the research goal of evaluating the runtime performance of various container runtimes. Also detecting the container escape at the time of deployment can help to contribute toward zero-day attacks.

3 Methodology

For resource efficiency and portability, containers have become an obvious choice for cloud-native application deployment. However, the shared kernel architecture of popular container runtimes like Docker makes it vulnerable to attacks and breaches. So other container runtimes like gVisor, Kata Containers, Quark Containers, and Youki tackle this

[2] `https://nvd.nist.gov/vuln/detail/CVE-2021-4034`

problem in their own unique way by complying with the open container initiative. Based on the previous work done by various researchers, an evaluation of only a few technologies was done, and as new runtimes like Quark Containers and Youki are introduced to the market, a more detailed evaluation needs to be done to determine their effectiveness against various scenarios.

3.1 Container Runtime Evaluation.

So evaluating various container runtimes will provide insights on their performance, which will help individuals cherry-pick runtimes that can favor them according to their requirements. Also, some of these runtimes are written in Go and Rust, which have their own advantages and drawbacks from a performance standpoint. So benchmarking these runtimes mentioned in Table 1 will also let people know which runtime they should favor, considering parameters like CPU, memory, I/O, and container lifecycle. To carry out the evaluation on these container runtimes, tools like Hyperfine [3] and Sysbench [4] are used along with some Linux commands because they provide various system-level assessments to fetch various metrics that will help in research to get insights.

3.1.1 Methodological Considerations and Justification.

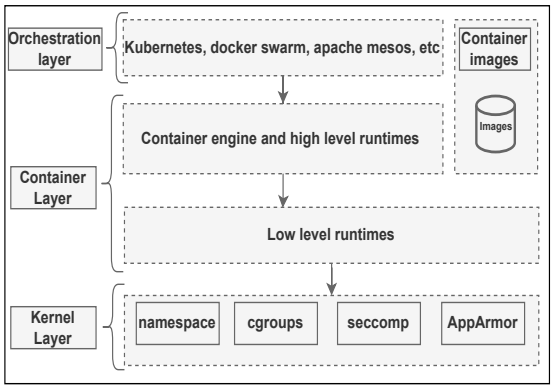
There were various methodologies and tools considered for the container runtime evaluation, but lastly, Hyperfine and Sysbench were used. When looking for various open-source benchmarking tools like obench, Bonnie, Kube-Bench, Kubestr, touchstone, doclite, etc., they all had one flaw: they only supported a particular runtime. For instance, Touchstone only supported Docker, Gvisor, and Kata containers. So these tools were avoided, and in this project, the requirement was that the tools that are used for evaluation should be consistent, versatile, and precise. It should be widely adopted by the community. So all these requirements were fulfilled by Hyperfine and Sysbench.

One of the novel contributions of this research will be a detailed, data-driven evaluation of container runtimes that will help people select a tailored solution by selecting the runtime.

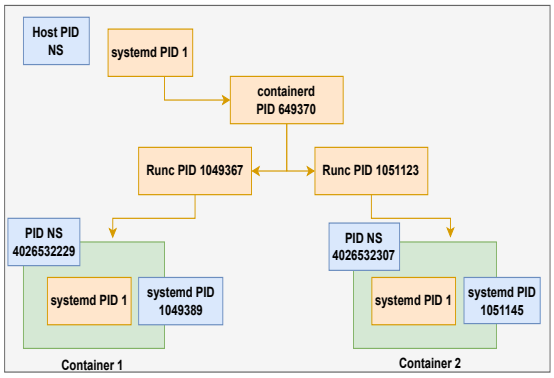
3.2 Container Escape Detection.

As shown in Figure 1 the first layer is the kernel layer, which is the main substructure of the container. It uses the same shared kernel on the host and leverages its features to isolate each container. Following that, there is a container layer, which has container runtimes (both low and high). Its main job is to create and manage containers by using kernel layers. Finally, there is an orchestration layer that is used for managing the various containers running across the cluster of VMs. It is evident that most of the container runtimes that are available on the market all use Linux kernel features to provide isolation between containers. But if there are vulnerabilities in the kernel, then attackers can leverage them, escape the container, take over the host, and also get access to other containers running in the environment. So it becomes very essential to detect container escapes in the environment to make it secure. In Linux, there are various namespaces like PIDS, MNT, USER, UTS, etc. So container runtime uses the namespaces in Linux and

[3] `https://github.com/sharkdp/hyperfine`
[4] `https://github.com/akopytov/sysbench`



(a) Basic building block of container



(b) PidNS working

Figure 1: Container architecture and PidNS working.

allocates resources like PIDS, which is used to provide isolation from other containers. So this research methodology focuses on one of the Linux kernel features, which is the PID namespace, that plays an important role in providing isolation between processes running on the host and processes running on various containers.

The PID namespace also allocates each process a PID, which acts as an independent system. So when the container starts, the container processes are derived from the Docker daemon process, and when a container is launched, each container gets its own PID namespace. Also, the process running in the container will start from 1 within the container. The same process running inside the container can also be seen on the host by looking at processes by container namespace. The process on the host will have a different pid than the process running inside the container. So to get the same process pid that is running inside the container, one can check it by getting the pid on the host by container pid namespace, and after getting the process id on the host, one can open the file /proc/PID(found on the host)/status and look for the key **NSpid**, and the value that one will be getting is of the process running on the host. So this concept is used to create an algorithm that will check various phases, like getting the container metadata, process tree checking, namespace matching, etc., to detect potential container escape. The evaluation of this methodology and design will be checked by simulating various test cases to check the robustness of the algorithm.

3.2.1 Justification of the chosen methodology.

There are other alternative methods that can help detect container escape, like syscall monitoring, file system integrity checks, and network activity access, but each has its own drawbacks. For example, syscall monitoring can detect abnormal system-call patterns, but it is susceptible and vulnerable to evasion tactics. The Pid-based container detection method is lastly chosen as the fundamental layer of isolation between the container and host. The solution provided is lightweight, generally applicable, and is used to detect early anomalies in the container.

4 Design Specification

In this section the explanation of why container runtimes that are used for experiments are selected and the functionality of the proposed algorithm is given.

4.1 Container Runtime selection and justification.

- Docker: It is widely accepted as per Open Container Initiative industry standards which is written in Go and evaluation would be incomplete without including docker [5] as it is most commonly used in container runtime.
- Kata Containers: It provides hardware-level virtualization, offering enhanced isolation with a combination of lightweight VMs and containers [6].
- Youki: Written in Rust, providing a performance-focused alternative to Go-based runtimes [7].
- gVisor: It provides sandbox-level virtualization with user-space kernels [8], balancing performance and security.
- Quark Container: It leverages hypervisor-level virtualization for heightened security [9], particularly suited for scenarios requiring strict isolation.
- Overall Considerations:
  - Diverse Technologies: The following runtimes were selected because of the diversity of programming language used to develop runtime and virtualization level that ensures a comprehensive evaluation.
  - Relevance to Trends: As per the new technologies like fog and edge, IoT, and wasm (web assembly) runtimes like youki, docker, and quark containers were selected, thus contributing to the recent trends in the market.

Technology	**Language**	**Virtualization**
Docker	Go	OS-Level
Kata Containers	Go, Rust	Hardware-Level
Youki	Rust	OS-Level
gVisor	Go	Sandbox-Level
Quark Container	Rust	hypervisor-level

Table 1: Technologies and Virtualization

[5] `https://www.docker.com/`
[6] `https://katacontainers.io/`
[7] `https://containers.github.io/youki/`
[8] `https://gvisor.dev/`
[9] `https://github.com/QuarkContainer/Quark`

4.1.1 Benchmarking Metrics for evaluation

Focused on the following performance metrics to evaluate the container runtimes:

- Container lifecycle: Time taken for the container to build, run, and kill the running container.
- Resource Utilization: CPU, Memory, and I/O consumption during runtime.
- Overall Runtime Performance: Comprehensive assessment of container runtime performance.

4.2 Explanation of proposed algorithm

The Algorithm is designed to detect and prevent the container breakout, which occurs when a process in a container escapes from its isolation and gains access to the host resources. The algorithm has five key components which are check pid namespace, check child pids, check child pid namespace, check container pids on the host, and check child pids on the host.

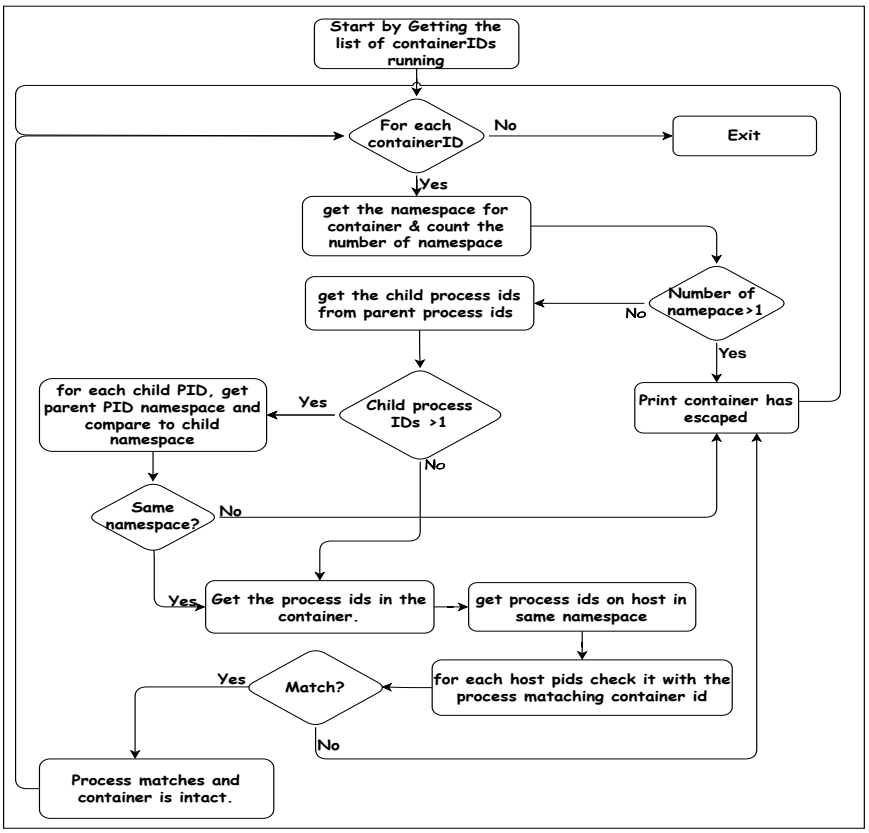


Figure 2: Flow diagram of the container detection.

4.2.1 `check_pid_namespace:`

The algorithm starts by getting the list of running container IDs and iterating through each container ID one at a time. First, the algorithm calls the function called check pid namespace which is also the driver function of the algorithm. Its task is to login

into the container and check the number of pid namespace in the container(`docker exec "$container_id" bash -c "lsns -t pid" | awk 'NR>1'`). Then it checks the condition **no of namespace > 1**. If more than one pid namespace is found in a container then it signals that the container is escaped and prints the container id which is escaped and will again to the starting point and iterate through a new container id. Otherwise will call the other function called check child pids.

4.2.2 `check_child_pids:`

This function takes two arguments called container id and pid namespace of the container. So when called the driver function supply these two arguments to the function. The main task of this function is to traverse the process tree within the container from the initial process with the help of parent PID and will extract all the child pids that were spawned and map out the entire process hierarchy that has been originating from the container start. All child pids will be checked with the parent pids by calling another function called `check_cpid_namespace`.

4.2.3 `check_cpid_namespace:`

The main objective of this function is to check for each child PID and parent PID(`docker exec "$container_id" bash -c "cat /proc/$cpid/status" | grep PPid`) should have the same namespace, so fetch the parent PID namespace that it should match and compare child process namespace to parent namespace. If mismatch, indicate the message that the process has escaped the namespace and container escape is detected.

4.2.4 `host process analysis:`

If the namespaces in the previous function are matched then the command(`docker exec $container_id bash -c ps -e -o pidns,pid | awk 'NR>1' | awk '{print $2}'`) gets a list of all valid in-container process IDs. Also, a similar command is used to retrieve process IDs running on the host that are within the same namespace context as the container. Then these host processes are checked against the known good container process list. If all the processes are matched and confirmed it means the container is intact and if there are any additional processes it indicates that a container escaped to the host.

4.3 Additional checks in container escape detection

The algorithm starts by getting the ID of the Docker container that is currently running and needs to be inspected. Once a list of container IDs is fetched it is saved in a variable called containerID. A for loop is then used to iterate through each container ID so that multiple containers can be checked.

Then using the containerID the program inspects the directories mounted from the host into the container by getting a list of mounts using the docker inspect command. This list is searched for any sensitive host directories like /etc - if found, a risk message is printed about the container having access to sensitive files.

Following that, the processes running inside the container are obtained using the docker top command which will give the list of processes. This list of processes is then searched for suspicious processes like sh, bash, and netcat utilities which could be used to get unauthorized access attempts from within the container.

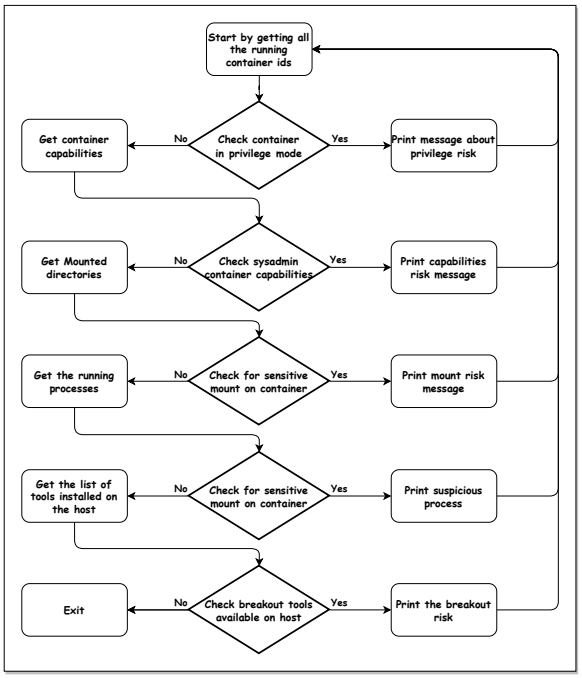


Figure 3: Flow diagram of the Additional checks in container escape detection.

Finally, the code checks if breakout tools like nsenter, chroot, and docker exec are present on the host itself. These tools could allow escape from the confinement of the container. If available, a risk message is printed about the breakout opportunity. After iterating through all the defined checks for a given container ID from the input list, the algorithm ends its execution.

Both algorithms will be tested with various scenarios, like normal container operations, child pid mismatching and matching with parent pid, multiple container operations, host process matching the container process, etc. So by experimenting with these various use cases, it can be determined the robustness and ability of the program to find the abnormal behavior that will indicate the container's escape.

5 Implementation

The Implementation involves various steps like setting up the test bed for the experiment and configuring various prerequisites and tools required for conducting research and implementing it.

5.1 Experimental Setup:

5.1.1 Hardware Configuration

For the runtime evaluation, we have utilized Amazon Web Services (AWS)[10] instances with the following hardware configurations:

- Docker, youki and gVisor (t3.xlarge[11]):

[10] `https://aws.amazon.com/`

[11] `https://aws.amazon.com/ec2/instance-types/t3/`

  - CPU: Intel Xeon Platinum 8000 series (4 vCPUs).
  - Memory: 16 GB RAM.
  - Storage: EBS-backed storage.
  - OS used: Amazon Linux for docker and Ubuntu for youki and Gvisor.

- Quark Container and Kata Containers (i3.metal[12]):
  - CPU: 2nd Gen Intel Xeon Scalable processors (Up to 72 vCPUs).
  - Memory: 512 GB RAM.
  - NVMe SSD-backed instance storage.
  - Os used: Ubuntu for both runtimes.

5.1.2 Benchmarking Tools

For benchmarking used the following tools:

- Sysbench: Used for evaluating system performance by stressing various system components
- Hyperfine: Employed for precise measurement of command-line program execution times

5.1.3 Benchmarking Procedure

The benchmarking procedure involved the following steps for each container runtime:

- Container Initialization: A simple docker file was created which used ubuntu latest image and the apt-update command is used to update all the necessary packages. Also, sysbench was installed which would be required for benchmark. Using this file docker image was created with the below command. `docker build -t ubuntu:sysbech -f Dockerfile .` To use this image in kata container runtime

```
[root@ip-172-31-36-138 sysbench]# cat Dockerfile
FROM ubuntu
RUN apt-get update && apt-get install -y sysbench
```

Figure 4: Dockerfile for sysbench.

  the build docker image is exported using command(`docker save -o ubuntuSysbench.tar ubuntu:sysbech`). Then below command is used to import that in docker.io library so that the ctr command can use that image.

```
ctr images import ubuntuSysbech.tar
```

- Metric Measurement: Sysbench and Hyperfine are the linux tools that are used to collect data on startup time, resource utilization, and overall runtime performance.
  - Sysbench: The image that is built in the previous section is used to run docker container for the below Sysbench benchmark.

[12] `https://aws.amazon.com/ec2/instance-types/i3/`

* CPU: Using sysbench the CPU performance is measured by calculating the maximum prime number upto 20000 using the below command. Note docker command is used for all other runtime except the kata container. For kata container the command used is 'ctr'.

```
docker run -it ubuntu:sysbench sysbench cpu --cpu-max-prime=20000 run
ctr run --cni --runtime io.containerd.run.kata.v2 --rm -t \
docker.io/library/ubuntu:sysbench cpu --cpu-max-prime=20000 run
```

* Memory: Using the sysbench command the memory performance is measured using sysbench which take memory block size ok 1 kilobyte and the total size of the memory buffer that sysbench will use in this case it will be 1 Gigabyte.

```
docker run -it ubuntu:sysbench sysbench memory --memory-block-size=1K \
--memory-total-size=1G run
ctr run --cni --runtime io.containerd.run.kata.v2 --rm \
-t docker.io/library/ubuntu:sysbench my-sysbench-container sysbench memory
--memory-block-size=1K --memory-total-size=1G run
```

* I/O: I/O performance evaluation is done using sysbench in three phases. In the first phase, it starts creating various files which are collectively of 10 Gigabytes and that is done by using random read-write mode. In the second phase it runs the I/O benchmarking on those files. Lastly in the third phase, the files generated in the first phase are deleted.

```
docker run -it --rm ubuntu:sysbench /bin/bash -c "sysbench fileio \
--file-total-size=10G --file-test-mode=rndrw prepare  sysbench fileio \
--file-total-size=10G --file-test-mode=rndrw run \
sysbench fileio --file-total-size=10G --file-test-mode=rndrw cleanup" \
--memory-total-size=1G run
```

– Hyperfine: It is used to gather metrics of container lifecycle that is build, run, and delete. So a separate script is created for each part and it simply has docker build, docker run and docker delete commands. This script is run sequentially by first cleaning the cache and assigning warmup time and min runs that is used by the workload to get stabilized and the sequence is run 100 times for better accuracy.

```
hyperfine \
  --prepare 'sudo sync; echo 3 | sudo tee /proc/sys/vm/drop_caches' \
  --warmup 10 \
  --min-runs 100 \
  'sudo ./build.sh && sudo ./run.sh && sudo ./delete.sh' \
  --show-output
```

- Logging: The output of commands which provide detailed logs are used for post-experiment analysis.

5.1.4 Container escape detection:

The container escape detection which investigates container isolation is implemented as follows:-

- PID Namespace Analysis:- The PID namespace analysis algorithm, implemented in Bash, and it iterates through running containers, performing PID namespace checks, and analyzing child processes to detect potential container escapes.
- Container Isolation Analysis:- Container Isolation Analysis algorithm which is implemented in Bash inspects various aspects of container isolation, highlighting potential security risks based on privileged mode, capabilities, mounted directories, container processes, and common breakout tools.

Jenkins Integration: It is seamless integration into the deployment pipeline, and the security scripts can be incorporated into Jenkins[13].

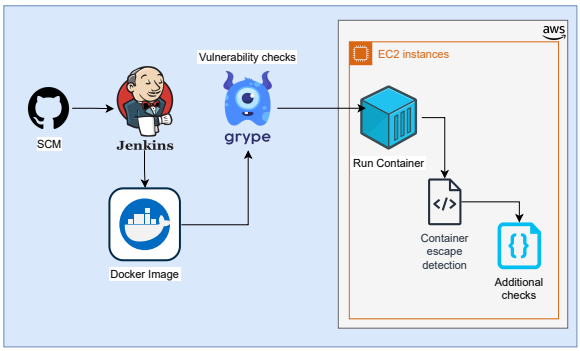


Figure 5: Dockerfile for sysbench.

- Pipeline Trigger: The pipeline can be triggered automatically when a commit is done in an SCM like GitHub[14] through webhook integration with Jenkins.
- Vulnerability scan: The container image is built using docker and scanned with the vulnerability scanner called Grype[15]. And if there are any critical vulnerabilities in the container it will stop the pipeline.
- Checks for escape: The container is run and will go through the container escape and additional checks algorithm. if the escape is detected the container id will be returned.

6 Evaluation

6.0.1 CPU Performance:

The evaluation of CPU performance across different container runtimes reveals their ability to handle CPU-intensive tasks. Among the evaluated runtimes, Quark Container demonstrated the highest CPU speed, achieving an average of 16.43 events per second with an average latency of 60.86 ms.

It is then followed by Kata [16] at 15.56 and Docker at 15.33 events per second. Others displayed competitive performances, averaging around 14.79 to 14.99 events per second.

[13] https://www.jenkins.io/
[14] https://github.com/
[15] https://github.com/anchore/grype
[16] https://github.com/kata-containers/kata-containers/releases/tag/3.2.0

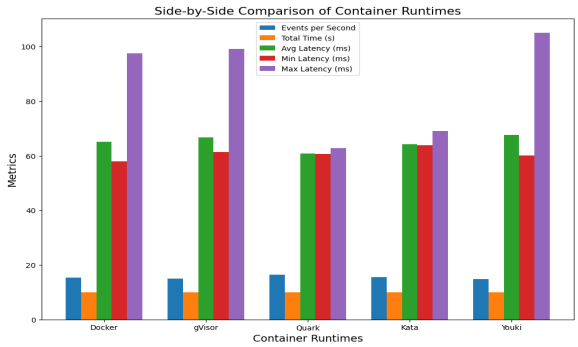
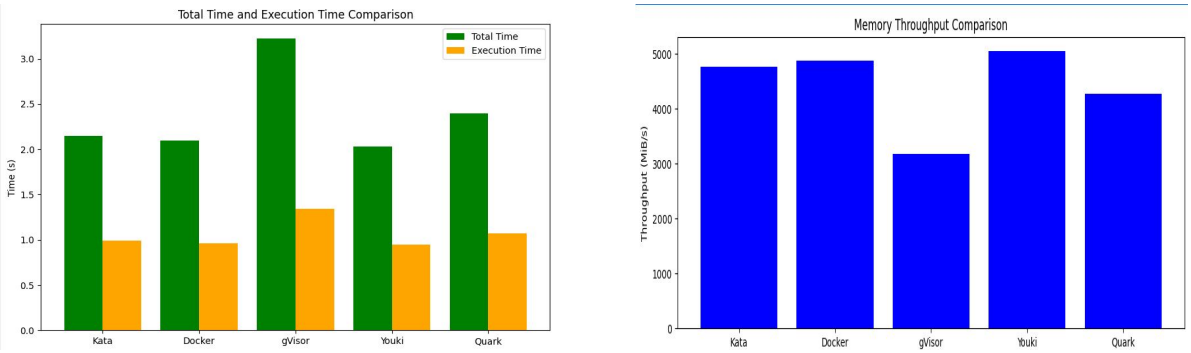


Figure 6: CPU Performance.

These results imply that if isolation and performance are considered, both quark and kata containers, which provide hypervisor-based isolation, are the best choices. Otherwise, one can go with Docker, which is highly adopted in the organization.

6.0.2 Memory Throughput:

The memory throughput analysis provides insights into how well each container runtime manages memory-intensive operations. In this benchmarking, the memory throughput was done using a sysbench tool with a block size of 1KiB and a total size of 10GiB. Out of all, Youki showcased a high throughput of 5,047.10 MiB/sec, completing the test in 2.0272 seconds with an execution time of 0.9447 seconds. It is closely followed by Kata Container and Docker at a throughput of 4,881.65 MiB/sec, an execution time of 0.9584 seconds, and a throughput of 4,881.65 MiB/sec. Execution Time: 0.9584 seconds. Quark also performed well, but Gvisor had the lowest performance.



(a) Memory Operation per seconds

(b) Memory throughput

Figure 7: Memory performance metrics.

6.0.3 I/O Operations:

The I/O performance evaluation focused on random read and write operations, providing a comprehensive view of each container runtime's ability to handle diverse storage workloads. In terms of I/O operations, gvisor and Docker showed similar read and write speeds, with 170 MiB/sec. Youki, Quark Container, and KATA displayed competitive I/O performance, with minor variations in read and write speeds. Overall execution times

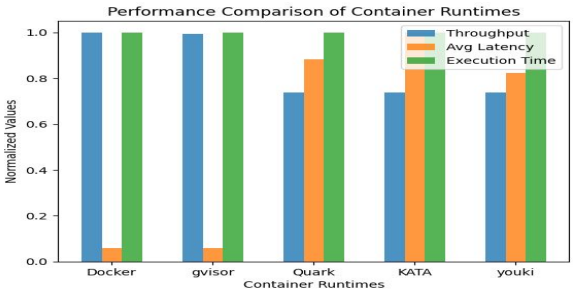
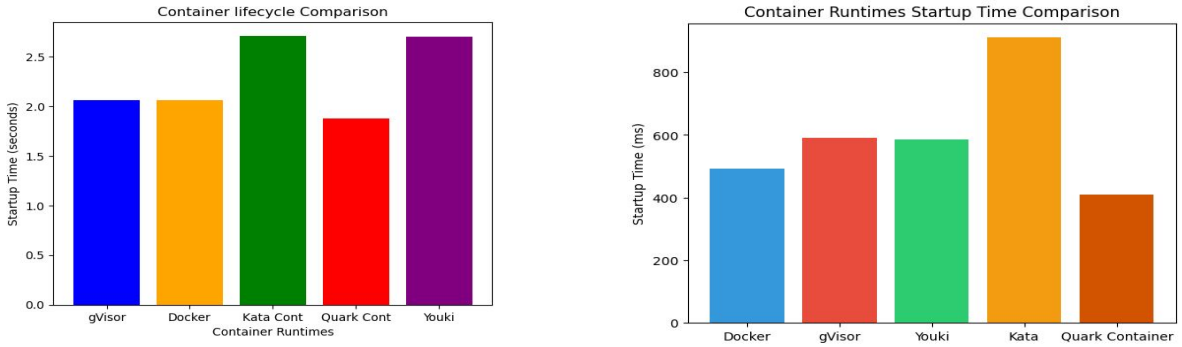


Figure 8: Memory Throughput.

vary, with Docker and Gvisor completing the tests in around 10 seconds, while Quark Container, KATA, and youki take a bit longer. Thread fairness shows consistent values across different runtimes, indicating a balanced distribution of events and execution time among threads.

6.0.4 Container Operations:

- Container lifecycle: Quark container has the fastest mean execution time(1.879s) with minimal variations(σ: 0.041s) that suggest consistency and performance. Gvisor and docker have similar times with the execution time 2.063. The rest of them have slower execution times with greater standard deviation.



(a) Container lifecycle.

(b) Container startup time.

Figure 9: Container performance metrics.

- Container startup time: From the graph, it is evident that the Quark container startup is 409.22 milliseconds, which is faster than the rest of the runtimes. The performance of Docker is also great, with 492.06 milliseconds. Following them are Youki and Gvisor at 585.3 and 591.92 milliseconds, respectively. The worst to perform is kata container i.1 911.19 milliseconds. Even though Quark Container provides hypervisor-level isolation, its performance is better than that of its competitors, like Kata Container. So if performance and isolation level security are concerns, the quark container is the obvious choice.

6.1 Container escape detection

6.1.1 Experiment 1: Normal Execution

Here the first test case scenario that is taken is that the container is fully intact which has no critical vulnerability and passes all the checks. It goes through a vulnerability check

Figure 10: Test case 1.

that checks for any critical vulnerability in the docker image and following that it passes both the algorithm stages. This was tested repeatedly just to check that the algorithms didn't provide a false positive response.

6.1.2 Experiment 2: Detecting container escape.

As shown in the figure during every deployment the same cycle will be repeated and each and every time it will go through all the stages. In this experiment, it passes all the stages until it fails at the check container escape stage in Jenkins. In this scenario, the container

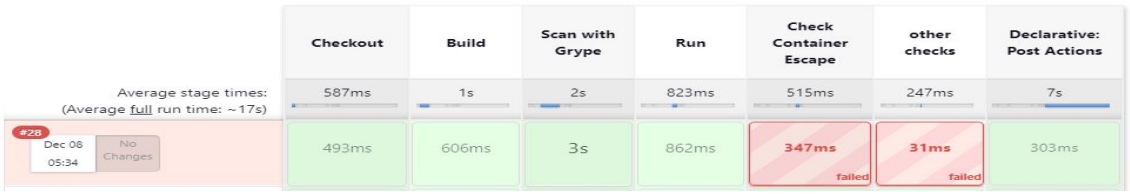


Figure 11: Test case 2.

escape is done intentionally to check whether the algorithm can detect container escape or not.

6.2 Experiment 3: Additional checks that can prevent escapes.

In the third test case intentionally a sensitive host directory was mounted on the container to check whether the algorithm detected one of the edge cases successfully or not. As

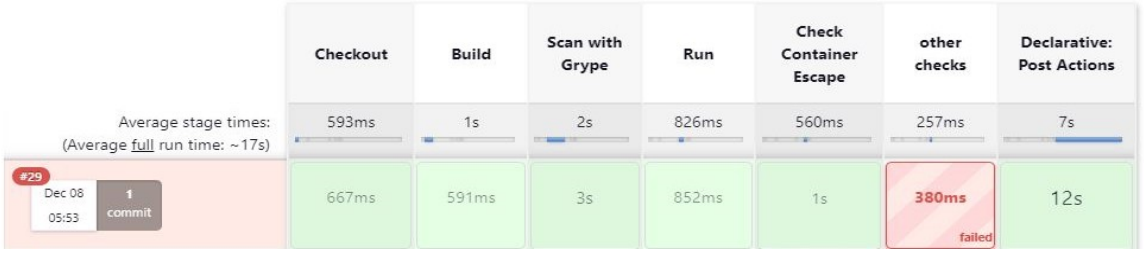


Figure 12: Test case 3.

shown in figure 12 it was detected with the message in the log printing that a sensitive host directory is mounted.

6.3 Experiment 4: Detecting Vulnerable Container Image.

In this test case, we took a vulnerable image that had a critical vulnerability. So whenever a container is built it goes through the container image scanner which checks all the layers of the container image for vulnerability based on the data it has. So here it detects that the container image that is built has one or more than one critical vulnerability.

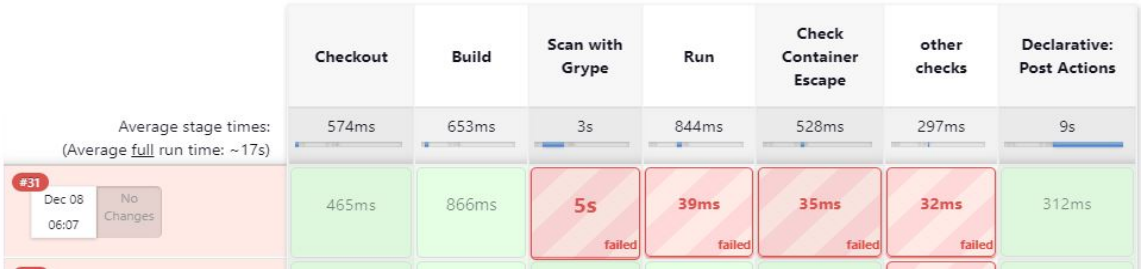


Figure 13: Test case 4.

6.4 Discussion

The research question was to evaluate container runtimes on performance metrics while implementing solutions to detect escapes. The objectives were successfully achieved by benchmarking diverse runtimes and designing a PID namespace analysis technique. Quark performed best than other runtime when CPU performance was performed, indicating efficient CPU resource utilization. Docker and Kata performed reasonably well, while the remaining runtimes could benefit from improvements in CPU Performance. For memory-intensive tasks docker and youki seem to have performed well as they had faster memory write speeds than other runtimes. Following that were kata container and gVisor which performed slightly slower. Quark's slowest performance. For I/0 performance kata and youki showed the slowest I/0 performance when compared with other container runtimes. docker, gvisor, and quark had similar I/O performance in terms of throughput and operation per second. Kata and youki showed slower performance when compared with other runtimes. For startup time quark outperformed other runtimes resulting in fastest startup time. It was followed by gVisor, Youki, Docker, and Kata.

Docker, which is a popular choice for containers, has been benchmarked for many years on various parameters like memory, I/O, and CPU. It is well-documented and has been used in production by many of the organizations. However, new runtimes like Kata Runtime v2 and Quark strike a balance between security and performance as they leverage the hypervisor layer for isolation between containers. This led to variations in performance results, but overall, Quark outperformed Kata Container Runtime. It has shown promise in terms of startup time, CPU, and I/O. So organizations can use it for certain use cases, like serverless computing, and even to run their everyday workload. However, it is important to consider that container runtimes are continuously evolving, and runtime performance can change with new optimization and performance. Also, the evaluation done should be on the real-world workload so that actual behavior can be captured.

The algorithms proposed and implemented focused on the examination of pid namespaces, which yielded noteworthy results. It was able to identify containers that escaped to the host by checking various parameters, like whether the child process belongs to the same parent process and the container has more than one namespace that escaped, etc., which indicated a potential security concern. It was also able to find various other aspects, like containers running in privilege mode, capabilities, critical host mount directories mounted on containers, container breakout tools, etc., thus providing a broad perspective on isolation security. However, there are some shortcomings, like the fact that the algorithm only uses the Docker command, thus neglecting other technologies on the market like podman, rkt, ctr, etc.; hence, they are runtime-specific. Also, it only focuses on one of the kernel namespaces, like PID, thus heavily relying on it.

7 Conclusion and Future Work

The main objective of the research was to identify key performance metrics of the leading open-source container runtime while also implementing a novel solution for detecting container escapes. The main aim of the research question was to evaluate Docker, Kata Containers, Gvisor, Quark Containers, and Youki across metrics like container lifecycle seed, resource utilization, and overall runtime throughput, while also designing methods to reveal container isolation breaches through process identifier namespace analysis. The benchmarking of various container runtimes was largely successful in generating important insights. From the experiments, the key findings showed that quark containers demonstrate excellent performance in terms of container speed and efficiency despite using the hypervisor-based isolation technique. Comparatively, Kata Containers traded some throughput for enhanced security guarantees. The PID namespace analysis technique accurately pinpointed suspicious process behaviors that could indicate violations of container boundaries and isolation controls. The conclusion shows that while finding the ideal container runtime for deploying their applications, organizations should consider security, performance, and complexity. As containers provide functionality, portability, and efficiency for microservice architectures, this research provides in-depth real-world metrics for various container runtimes and escape detection algorithms, which are important for deployment decisions. The limitation of this research was that only a few open-source runtimes were considered for container evaluation rather than proprietary options. The algorithms that were proposed were only for the runtimes that can integrate with Docker; other engines, like Podman, should also support the proposed algorithms. Also, research focuses only on one namespace, so more namespaces and Linux kernel features should also be explored. More work can be added to the current study in the future by involving more runtimes and various types of benchmarking for repeated periods. Also, the coverage for escape detection should be increased by including other Linux features and integrating them with machine learning for more automated analysis of process behaviors with containers. Also, this research can be used in developing security products or tools that cater to customer needs.

References

Abhishek, M. K. and Rajeswara Rao, D. (2021). Framework to secure docker containers, *2021 Fifth World Conference on Smart Trends in Systems Security and Sustainability*

(WorldS4), IEEE, pp. 152–156.

Brady, K., Moon, S., Nguyen, T. and Coffman, J. (2020). Docker container security in cloud computing, *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, IEEE, pp. 0975–0980.

Combe, T., Martin, A. and Di Pietro, R. (2016). To docker or not to docker: A security perspective, *IEEE Cloud Computing* **3**(5): 54–62.

Creasy, R. J. (1981). The origin of the vm/370 time-sharing system, *IBM Journal of Research and Development* **25**(5): 483–490.

Docker, I. (2020). Docker, *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker* .

Flauzac, O., Mauhourat, F. and Nolot, F. (2020). A review of native container security for running applications, *Procedia Comput. Sci.* **175**: 157–164.

German, K. and Ponomareva, O. (2023). An overview of container security in a kubernetes cluster, *2023 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, IEEE, pp. 283–285.

Jian, Z. and Chen, L. (2017). A defense method against docker escape attack, *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, ACM, New York, NY, USA.

Kumar, R. and Thangaraju, B. (2020). Performance analysis between RunC and kata container runtime, *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, IEEE, pp. 1–4.

Price, D. and Tucker, A. (2004). Solaris zones: Operating system support for consolidating commercial workloads., *LISA*, Vol. 4, pp. 241–254.

Randazzo, A. and Tinnirello, I. (2019). 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)*, IEEE, pp. 209–214.

Rangnau, T., Buijtenen, R. v., Fransen, F. and Turkmen, F. (2020). Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines, *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 145–154.

Sultan, S., Ahmad, I. and Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead, *IEEE Access* **7**: 52976–52996.

Yang, Y., Shen, W., Ruan, B., Liu, W. and Ren, K. (2021). Security challenges in the container cloud, *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, IEEE, pp. 137–145.