ILKOM Jurnal Ilmiah Vol. 16, No. 1, April 2024, pp. 12-26
Accredited 2nd by RISTEKBRIN No. 200/M/KPT/2020; E-ISSN 2548-7779 | P-ISSN 2087-1716

Quantifying of RunC, Kata and gVisor in Kubernates

Rahmat Purwoko [a,1]; Dimas Febriyan Priambodo [a,2,*]; Arbain Nur Prasetyo [a,3]
[a] *Cybersecurity Engineering, National Cyber and Crypto Polytechnic, H.Usa Street, Bogor 16120, Indonesia*
[1]*rahmat.purwoko@poltekssn.ac.id;* [2]*dimas.febriyan@poltekssn.ac.id;* [3]*arbain.nur@bssn.go.id*
** Corresponding author*

Article history: Received March 21, 2023; Revised May 14, 2023; Accepted February 12, 2024; Available online April 26, 2024.

Abstract

The advent of container technology has emerged as a pivotal solution for application developers, addressing concerns regarding the seamless execution of developed applications during the deployment process. Various low-level container runtimes, including runC, Kata Container, and gVisor, present themselves as viable options for implementation. The judicious selection of an appropriate low-level container runtime significantly contributes to enhancing the efficiency of Kubernetes cluster utilization. To ascertain the optimal choice, comprehensive testing was conducted, encompassing both performance and security evaluations of the low-level container runtimes. This empirical analysis aids developers in making informed decisions regarding the selection of low-level container runtimes for integration into a Kubernetes cluster. The performance assessments span five key parameters: CPU performance, memory utilization, disk I/O efficiency, network capabilities, and the overall performance when executing an nginx web server. Three distinct tools—sysbench, iperf3, and Apache Benchmark—were employed to conduct these performance tests. The findings of the tests reveal that runC exhibits superior performance across all five parameters evaluated. However, a nuanced consideration of security aspects is imperative. Both Kata Container and gVisor demonstrate commendable host isolation, presenting limited vulnerability to exploitation. In contrast, runC exposes potential vulnerabilities, allowing for exploits against the host (worker node), such as unauthorized directory creation and system reboots. This comprehensive analysis contributes valuable insights for developers, facilitating an informed decision-making process when selecting low-level container runtimes within a Kubernetes environment.

Keywords: Containerization; gVisor; Kata Container; Performance Analysis; RunC; Security Analysis

Introduction

The development of applications involves a series of intricate processes, including development, testing, staging, and production [1]. Each stage necessitates diverse environments, posing challenges in terms of software variations and configurations. This diversity often hampers the development process, particularly with regards to time efficiency. To address these concerns, container technology has emerged as a pivotal solution for application developers. Containers encapsulate applications with their code, runtime, system libraries, binaries, configurations, and dependencies [2]. While Virtual Machines (VMs) can mitigate similar challenges, container technology offers superior flexibility, scalability, and resource efficiency in management. Moreover, containers facilitate faster packaging and deployment compared to VMs.

Docker stands out as the foremost containerization platform [3], utilizing RunC as its default low-level container runtime. Other notable low-level container runtimes, such as Kata Container and gVisor [4]. Kata container and gVisor are low-level container runtimes that already have an added layer of security [5]. Security level in Kata containers and gVisor is also supported by the use of the Virtual Machine (VM) concept where containers will be isolated and seem separate from the host. The VM concept is of course coupled with the lightweight nature of the container [3].

In the realm of containers, the significance of Kubernetes cannot be overstated. Coordinating and managing numerous containers pose challenges, which Kubernetes addresses as an open-source cluster manager for containers, also known as container orchestration [6]. Kubernetes facilitates the deployment of multiple pods, where each pod supports multiple containers that can utilize associated service [7]. Selecting the appropriate low-level container runtimes becomes crucial in maximizing the effectiveness of a Kubernetes cluster, especially considering resource management in a multi-tenant environment [8]. Additionally, the scalability advantage achieved through auto-scaling demands careful attention, as efficient resource usage significantly impacts cluster performance and associated costs. This underscores the necessity for comprehensive knowledge regarding different low-level container runtimes in terms of both performance and security.

Research related to the performance of low-level container runtimes has basically been done. Some of the parameters measured include CPU, memory, and the performance of the I/O block [8]. We extend the scope to include a comparison of VMs with containers [3], [9], incorporating multiple quantifiable parameters. Furthermore, our focus

on testing within a Kubernetes cluster environment distinguishes this research, as previous studies often neglected this critical aspect.

To address the security aspect, previous studies [4] have touched on the security of container runtimes, but this research introduces a scenario-based approach to assess the exploitability of each container. The presence of three low-level container runtimes—RunC, Kata Containers, and gVisor—compels developers to make critical decisions for their specific hosting needs. Consequently, a thorough investigation into the performance and security aspects of these runtimes within a Kubernetes cluster becomes imperative. Our testing process will evaluate key performance metrics, including memory utilization, CPU efficiency, disk I/O throughput, and network performance. Additionally, we will conduct security tests to assess the vulnerability of these runtimes to potential exploits and their impact on the nodes or hosts responsible for running the pod. This research aims to empower developers with a robust understanding of the performance and security characteristics inherent in each runtime within a Kubernetes environment. This knowledge will facilitate informed decision-making, enabling developers to select the most suitable runtime for their specific use case, ultimately enhancing the overall security and performance of containerized applications in a dynamic computing landscape.

Literature Review

A. Containers

Containers are a lightweight, portable, and consistent way to package, distribute, and run applications and their dependencies [10], [11]. They encapsulate an application and its required components, providing isolation and ensuring that it runs consistently across various environments. Popular containerization tools include Docker and container orchestration systems like Kubernetes. Containers enhance the portability of applications across different cloud providers and even on-premises environments. They encapsulate everything needed to run an application, making it easier to move applications between different cloud platforms. From efficiency aspect, containers share the host operating system's kernel, reducing overhead and enabling the deployment of more containers on a single host. This efficiency aligns with the scalability and cost-effectiveness offered by cloud services.

B. Kubernates

Kubernetes functions as an open-source orchestrator for deploying containerized applications, providing scalability, automated deployment, and management capabilities [2]. Its roots can be traced to Google's internal Borg system, later named Omega, which was developed to oversee thousands of servers owned by Google. The increasing adoption of Kubernetes is in tandem with the transition from monolithic to microservices architecture. In contrast to the monolithic approach, microservices break down applications into smaller, specialized components that are interconnected to perform specific tasks, as illustrated in **Figure 1**, facilitating understanding for those new to the technology.

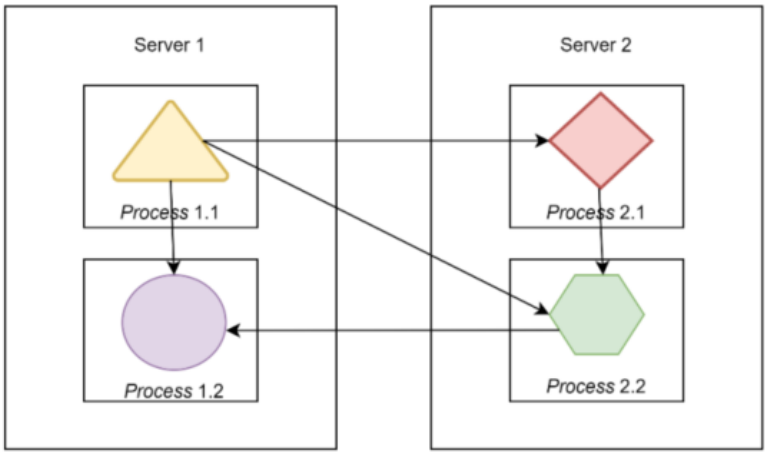


Figure 1. Microservices architecture from Kubernates in Action [12]

The surge in container usage for application deployment further fuels Kubernetes' popularity. The Kubernetes cluster architecture, illustrated in **Figure 2**, comprises master and node components [13]. The master component, featuring Kube-apiserver, acts as a liaison, communicating with etcd—a distributed data store. Kube-Controller manager oversees controllers like replication, pod, services, and endpoints. The Cloud-controller manager interacts with cloud services, while Kube-scheduler handles pod-to-node scheduling.

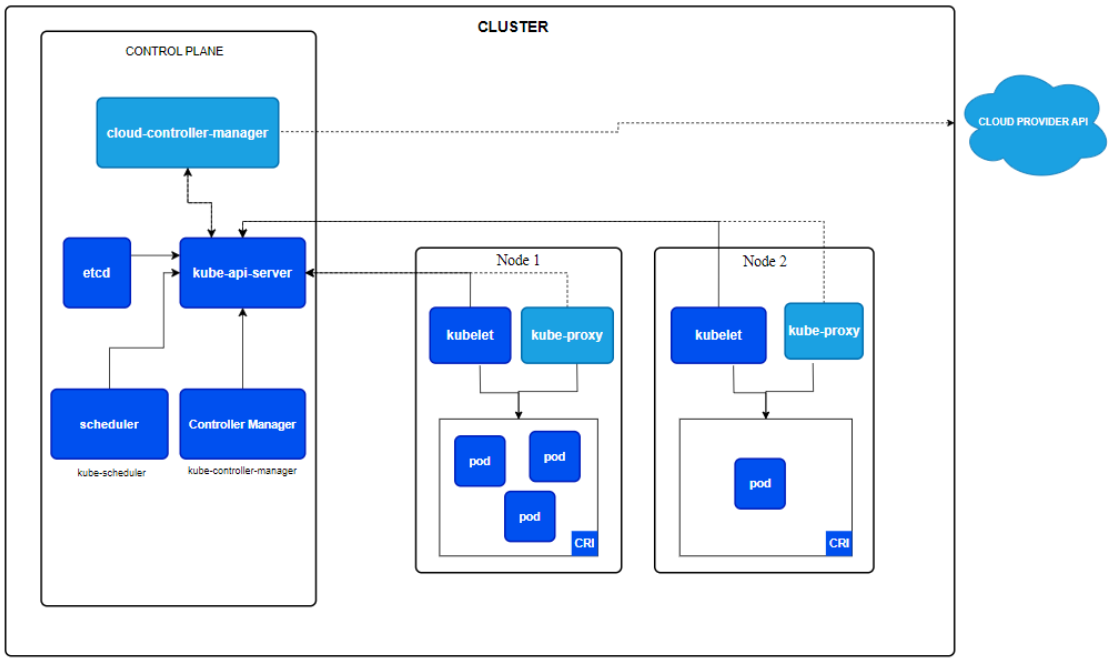


Figure 2. Kubernates cluster architecture [14]

Node components consist of Kubelets ensuring container runtime within pods, Kube-proxy managing low-level networking on nodes, and container managers (or runtimes) directly controlling containers. **Figure 3** delves into the Container Runtime Interface (CRI) architecture, serving as the link between kubelet components and container runtimes. Three CRIs—Dockershim, cri-containerd, and cri-o—offer diverse container runtimes for managing containers at the low level within the Kubernetes cluster. Understanding this architecture is crucial for effective utilization of Kubernetes in deploying and managing container-based applications.

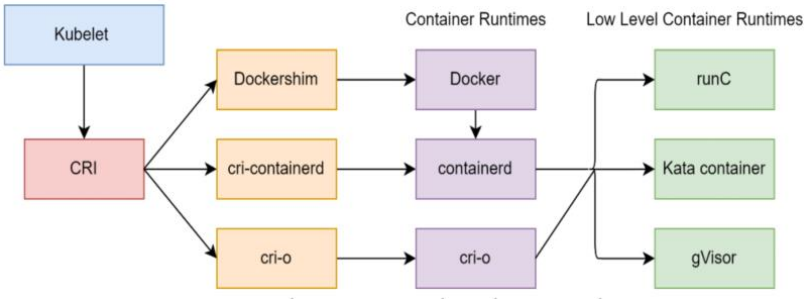


Figure 3. CRI in kubernetes cluster

C. Low level container Runtime

A Low-Level Container Runtime, often referred to simply as a "container runtime," is a fundamental component responsible for executing and managing containers on a host system. It operates at a lower level, interacting closely with the underlying infrastructure and facilitating the creation and execution of containerized applications. Several key aspects define the significance and functionality of low-level container runtimes. Low-level container runtimes play a critical role in ensuring the security of containerized applications. They implement measures to prevent unauthorized access, restrict resource usage, and maintain the integrity of the host system [15]. Understanding low-level container runtimes is essential for developers, system administrators, and anyone involved in containerized application deployment. These runtimes form the backbone of container technology, contributing to the efficiency, security, and portability of applications across various computing environments.

Low level container runtimes are container runtimes that only focus on creating containers. Low level container runtimes implement the OCI runtime and make it possible to create containers from “filesystem bundles”. Some of the low level container runtimes include RunC, gVisor, and Kata container [4]. RunC is one of the low level container runtimes which is used by default in container runtimes. Using RunC allows users to access the host more by using privileged mode. RunC is mostly used for nested containers or running containers on top of containers. RunC uses libcontainer to do the job of creating containers and works as a wrapper to abstract system calls [3]. RunC provides implementations for creating containers using namespaces, control groups, network interfaces, file access controls, security profiles, and capabilities.

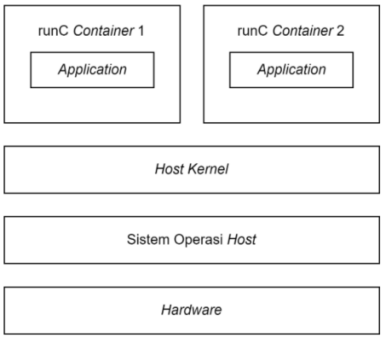


Figure 4. RunC Architecture [3]

Figure 4 illustrates the architectural details of container runtimes, particularly emphasizing the connectivity between the application and the host kernel within the RunC container. Notably, the depiction unveils that the application running on the RunC container establishes a direct connection with the host kernel. This architectural characteristic implies that RunC, as a container runtime, enjoys a higher level of access to the host environment, as it is not entirely isolated from the underlying host system. The reduced isolation suggests that RunC operates with a certain degree of proximity and interaction with the host, potentially having broader privileges compared to more isolated runtimes. This nuanced connectivity is crucial for developers and system administrators to comprehend, as it impacts the security and access control considerations when selecting RunC as the container runtime.

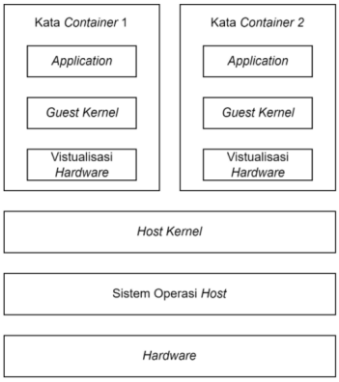


Figure 5. Kata container architecture [3]

The subsequent low-level container runtime we explore is the Kata container, designed to construct a secure and efficient container runtime by incorporating a lightweight virtual machine (VM) to enhance container isolation [16]. **Figure 5** illustrates that Kata container employs robust isolation through lightweight hardware virtualization at the container layer. Unlike RunC, Kata container utilizes a guest kernel dedicated to booting the VM, optimizing both boot time and memory usage [3].

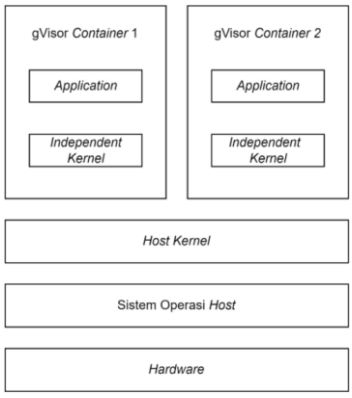


Figure 6. gVisor Architecture [17]

The final low-level container runtime in consideration is gVisor, developed by Google. gVisor distinguishes itself by utilizing an application kernel, coded in Go. Unlike other runtimes, gVisor generates its own kernel, operating independently. This characteristic enhances isolation from the host environment. In **Figure 6**, the utilization of this

independent kernel is depicted, illustrating its ability to intercept application system calls and function akin to a guest kernel. Notably, this is accomplished without necessitating translation through hardware virtualization. gVisor employs a unique approach to container technology, creating a distinct kernel that runs autonomously. The applications within the gVisor container operate on Sentry, an infrastructure that implements Linux and functions within the Linux environment [18].

D. Related Research

[19] the research focuses on comparisons to determine which virtualization technology is better to use. Comparable virtualization technology, namely between containers and VM. Tests are carried out with various benchmarking tools to measure the performance of containers and virtual machines. Several tools are used, including 7 Zip compression tests used to measure CPU performance, RAMspeed/SMP to measure memory performance, Iozone benchmark to determine disk I/O performance, Apache Benchmark used to perform load testing, and Eight Queen problem and 8 puzzle tests used to measure operation speed [19]. The results of the research conducted show that Docker containers have better performance when compared to VMs from each of the tests tested.

Research [20] focuses on performance comparisons between several virtualization technologies, namely Unikernels, containers, hypervisors, and Kata containers. Several parameters were tested, namely the size of the image, boot time, memory utilization and CPU utilization [20]. The test results show that the use of containers has an efficiency level in terms of memory and CPU consumption.

Research [3] focuses on performance comparisons between RunC and Kata container which is a low level container runtime. Testing is carried out because the two low level container runtimes represent different architectures. Testing is done by creating container environments on identical VMs. Container runtimes used, namely Docker. Several parameters were tested to determine the performance of the low level container runtime, namely the container boot test, I/O performance using the Bonnie++ tool, CPU and memory utilization, and network performance using the Python-based psutil tool [3]. The results of the performance tests conducted show that RunC has better performance when compared to Kata container.

Method

The research design used is based on Benchmarking Methodology for Network Security Device Performance draft-ietf-bmwg-ngfw-performance [21]. The methodology was released in October 2021 by the Benchmarking Technology Working Group. Benchmarking is a series of processes carried out by comparing two or more similar objects to obtain results that can be used as benchmarks for usage. One of the goals of doing benchmarking is to find out the level of performance of the object being tested [22]. Several stages that will be carried out are test objectives, test setup, test parameters, test procedures, measurement, and reporting. All the steps to be carried out can be seen in **Figure 7**.

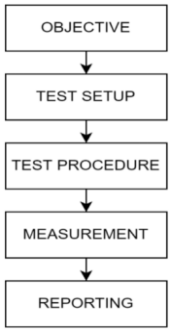


Figure 7. Benchmarking Methodology [21]

A. Objective

Some of the things that are prepared at this stage are determining the parameters to be tested, the device (complete with specifications) and the tools to be used to carry out performance and security tests. The performance parameters that will be tested in this study are CPU performance, memory performance, disk I/O performance, network performance, and pod performance when running nginx. Testing the performance of nginx running in the pod is carried out to determine the effect of the low level container runtime used on the web server in handling received requests. This needs to be done because to find out the optimal performance of the web server that is running. Problems that arise due to a non-optimal web server can cause web applications to be disrupted and even stop working. The security parameter to be tested is the level of isolation provided by each low level container runtime that is used for the worker node.

B. Test Setup

The installation process is carried out on the application in the test environment and the tools used to measure the tests carried out. All testing requirements that have been installed at this stage will be used to carry out the next stage. The need for the testing environment and the tools used can be seen in the previous stage, namely the objective stage.

C. Test Procedure

Testing was carried out on a test environment that had been prepared previously. There are two test schemes that will be carried out, that is:

- Testing of scheme 1 is done by testing the performance of CPU, memory, disk I/O, network, and nginx on pods running with three different low-level container runtimes on Kubernetes clusters.
- Scheme 2 testing is carried out by conducting several tests related to the low-level container runtimes used to determine the security in the form of the isolation level provided by each low level container runtime used. Further analysis is carried out to determine the extent to which containers can affect worker nodes. Security testing is carried out by utilizing the "SYS_ADMIN" capability which allows pods to mount [23].

Performance testing is carried out at least twice to fulfill the repeatability element [22]. In this test, it was repeated ten times. This is to ensure that the performance test results do not show a significant difference, so that the results obtained allow it to be used as a reference in analyzing the low level container runtime performance.

D. Measurement

Calculations were made on several parameters of the results of the tests carried out. Calculations are performed to determine the desired result according to predetermined parameters. The results of the calculations carried out will be presented in the form of a graph of the test results which can then be used as material in conducting a performance and safety analysis according to the specified parameters.

E. Reporting

A final report format is created based on the test results obtained from the measurement stage that has been carried out. The contents of the report made are graphs and comparison tables of the results of the overall scheme testing carried out and accompanied by an analysis of the test results. The existing graphs and comparison tables are then analyzed to find out how the performance and security of the low level container runtime used is. Knowledge of the results obtained can be used by application developers in determining which low level container runtime to use in the Kubernetes cluster for the production stage later.

Results and Discussion

A. Test Parameters

1. Performance Testing Parameters

Performance testing is carried out to find out the difference in performance of the low level container runtime used in the Kubernetes cluster. The performance testing parameters used can be seen in **Table 1**.

Table 1. Performance Testing Parameters

No	Parameter	Description
1	CPU	CPU performance testing is done using the sysbench tool. The command that is executed is a calculation of prime numbers up to the value specified with the --cpu-max-prime option. The specified value limit is 1000000 to make it easier to see the performance difference from the low level container runtimes used. The command is sysbench --test=cpu --cpu-max-prime=1000000 --num-threads=1 run.
2	Memory	Still using the sysbench tools to allocate a memory buffer and then read and write iteratively until the reserved volume (-- memory-total-size) is reached. The command used is sysbench memory --threads=1 run.
3	Disk I/O	The sysbench tool is used to evaluate several tests to compare performance related to data read and write speed, the number of searches that can be performed per second and the number of file metadata operations that can be performed per second. In the disk I/O test, several files with a total of 4 Giga Bytes were created which were divided into 128 files. The files that have been created are then read and written from each low level container runtime used. There are two commands used, namely the command to create a file and the command to run tests on files. The command used to create the file is sysbench fileio --file-totalsize=4G --file-test-mode=rndrw --time=300 --max-requests=0 prepare. The command used is sysbench fileio --file-total-size=4G --file-test-mode=rndrw -- time=300 --max-requests=0 run
4	Jaringan	Network performance testing is done using the iperf3 tool. Measurements were carried out using two pods where one pod runs as an iperf3 server and the other pod as an iperf3 client. The iperf3 -s -i2 command is executed on the server pod, while the iperf3 -c (ipadd_server) -i2 -t20 command is executed on the client pod.
5	Nginx	Pulled the image from nginx and then run it on ports 30151 (RunC), 30152 (Kata), and 30153 (gVisor). Nginx performance testing uses Apache Bench by sending 10,000 requests. The command used is ab -n 10000 -c 100 http://ip_address_pod:port_service/.

2. Security Testing Parameters

Security testing is carried out by looking for differences in the kernel used in each low level container runitmes that is run. The goal is to find out how far the isolation that is carried out at each low level container runtime can

secure the host, as well as the worker node that runs the container. Implementation is carried out by deploying pods that have added the "SYS_ADMIN" capability, so that further analysis can be carried out regarding the extent of isolation carried out by the low level container runtime to the nodes used.

3. *Test Environment Specifications*

Based on the results of the analysis of the needs of the testing system, hardware and software are needed to carry out performance and security testing of the low level container runtime used. The specifications for the computer, master node, and worker node used can be seen in **Table 2**. One VM created will act as the master node and the other VMs as worker nodes. The use of one master node and one worker node aims to fulfill the minimum requirements for creating a Kubernetes cluster.

Table 2. Test device specifications

Spesifikasi	Server	Master Node	Worker Node
Operating system	Proxmox Virtual Environment 7.2-3	Ubuntu Server 20.04	Ubuntu Server 20.04
RAM	32 GB	16 GB	8 GB
Storage	1 TB	200 GB	200 GB
Processor	Intel Xeon E-2224 (8M Cache, 3.4 GHz)	4 logic CPU (2 socket, 2 core)	2 logic CPU (1 socket, 2 core)

B. Test Setup

There is one container runtimes and several low level container runtimes installed to be able to perform performance and security testing. Some of the necessary tools are also installed at this stage. Kubernates Cluster installation and configuration begins with the installation of the Container Runtime (container). The second stage is the installation of Kubernates, which doesn't forget to pull configuration images with kubeadm to then obtain a token so that the worker node can join the master node by not forgetting to add the calico add-on.

By default, a container runtime will execute RunC immediately, therefore it is necessary to configure it so that the low level container runtime used in the Kubernetes cluster runs other runtimes such as Kata container or gVisor. Configuration is done in the /etc/containerd/config.toml file by adding Kata or gVisor as one of the runtimes that can be used.

Two images are used, one of which is built from the Dockerfile that was created and the other is pulled directly from Docker Hub. Image prsty4231/ubuntu_prase. Another image used is nginx 1.18.

C. Test Procedure

1. *Performance Test*

Tests on each parameter with a different low level container runtime were carried out ten times. After each test is completed, the pod used is immediately deleted to provide a test environment with the same resources. This is also done so that the pod used does not affect other pods in the used Kubernetes cluster.

2. *Security Test*

ubuntu-runc1.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
  annotations:
    container.apparmor.security.beta.kubernetes.io/ubuntu:
unconfined
spec:
  containers:
  - name: ubuntu
    image: prsty4231/ubuntu_prase
    command: ['sh', '-c', 'sleep 3600']
    securityContext:
      capabilities:
        add: ["SYS_ADMIN"]
```

Figure 8. Added SYS_ADMIN capability

The addition of the SYS_ADMIN capability is carried out for each YAML file used, as shown in **Figure 8**. The added capability aims to make more syscalls on the host (worker node). The security tests performed also utilized Linux cgroups as a mechanism to isolate running pods. The root user which is used by default in pods is also one of the supporters in conducting security testing. The root user is important because it will have full rights over the pods under test. The exploit requires the cgroup as the media to create a "release_agent" file and make a "release_agent" call by killing all processes in the cgroup. In simple terms, this can be done by adding a

cgroup controller and creating a child group. This can be done by creating the /tmp/cgrp directory, then adding the RDMA cgroup controller and creating a child cgroup. The technique used generally works on most cgroup controllers.

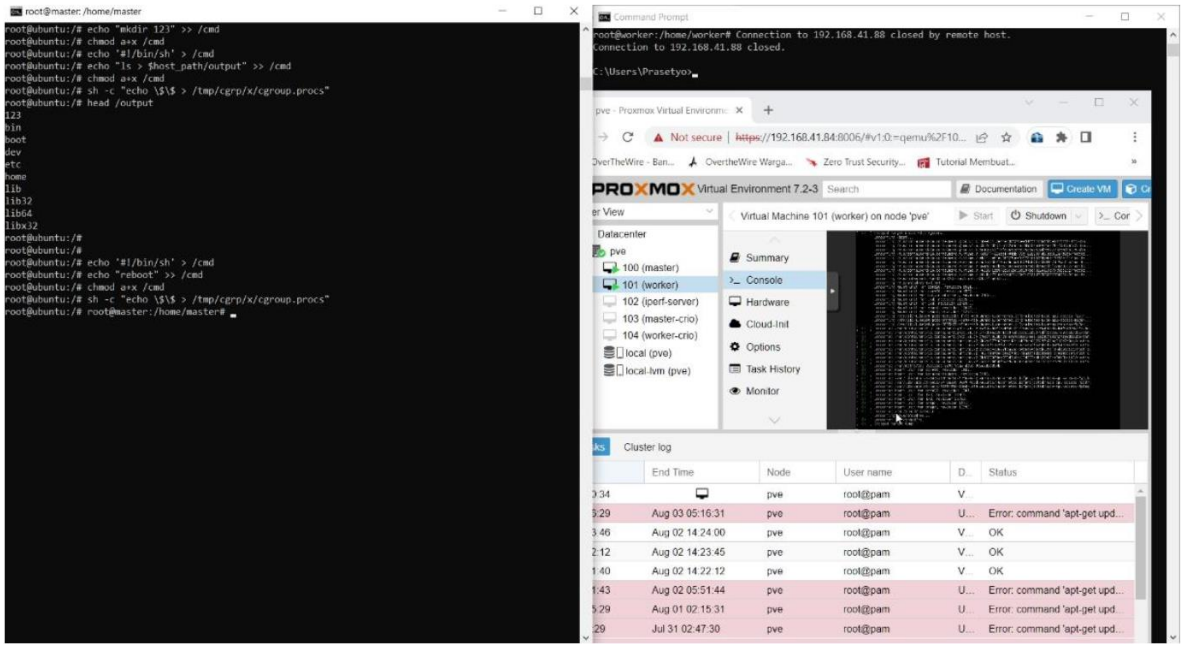


Figure 9. The worker node reboot was successful

Figure 9 shows the command to reboot the worker node from within the pod successfully executed. This proves that there is no isolation carried out by RunC which acts as a low level container runtime on the running pod. In contrast to RunC, Kata container and gVisor show an error message when the script wants to run shown in **Figure 10**. This certainly shows that both Kata container and gVisor provide a higher level of isolation to the pods being run.

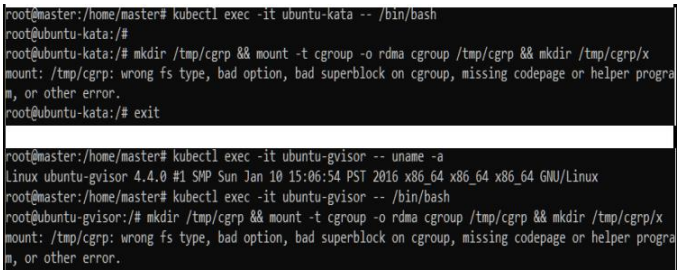
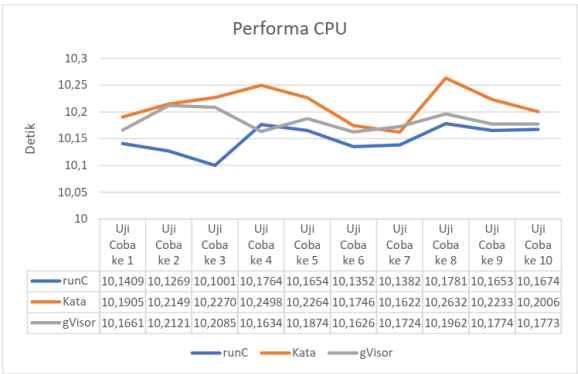


Figure 10. Error on Kata container and gVisor

D. Measurement

1. *CPU Performance*

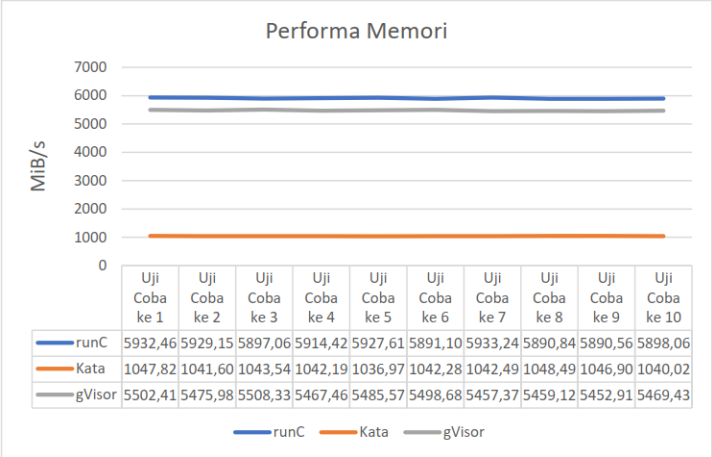


	Uji Coba ke 1	Uji Coba ke 2	Uji Coba ke 3	Uji Coba ke 4	Uji Coba ke 5	Uji Coba ke 6	Uji Coba ke 7	Uji Coba ke 8	Uji Coba ke 9	Uji Coba ke 10
runC	10,1409	10,1269	10,1001	10,1764	10,1654	10,1352	10,1382	10,1781	10,1653	10,1674
Kata	10,1905	10,2149	10,2270	10,2498	10,2264	10,1746	10,1622	10,2632	10,2233	10,2006
gVisor	10,1661	10,2121	10,2085	10,1634	10,1874	10,1626	10,1724	10,1962	10,1774	10,1773

Figure 11. CPU performance test results

Figure 11 shows the results of CPU performance tests conducted on three pods with different low level container runtimes. The results obtained are presented in graphical form which states the unit of time in seconds. The results show the performance of the CPU in running the sysbench command to calculate 1,000,000 prime numbers. Testing is carried out by limiting the use of threads, namely only using one thread for each test carried out. The faster the time required to process the CPU test being carried out, the better the performance. Tests carried out ten times on each pod showed different results. The average test is carried out for 10 seconds. **Figure 11** also shows that pods with RunC as the low level container runtime require the fastest time compared to Kata container and gVisor. The shortest time obtained from the results of the tests carried out was 10.1001 seconds using RunC, while the slowest time obtained was 10.1962 seconds using the Kata container.

2. *Memory Performance*

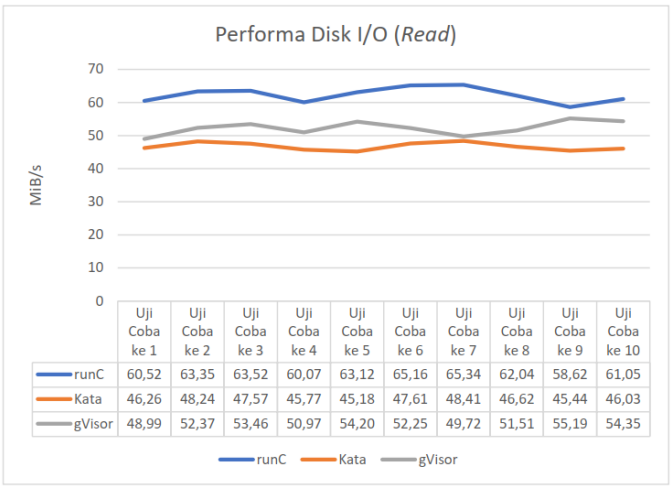


	Uji Coba ke 1	Uji Coba ke 2	Uji Coba ke 3	Uji Coba ke 4	Uji Coba ke 5	Uji Coba ke 6	Uji Coba ke 7	Uji Coba ke 8	Uji Coba ke 9	Uji Coba ke 10
runC	5932,46	5929,15	5897,06	5914,42	5927,61	5891,10	5933,24	5890,84	5890,56	5898,06
Kata	1047,82	1041,60	1043,54	1042,19	1036,97	1042,28	1042,49	1048,49	1046,90	1040,02
gVisor	5502,41	5475,98	5508,33	5467,46	5485,57	5498,68	5457,37	5459,12	5452,91	5469,43

Figure 12. Memory performance test results

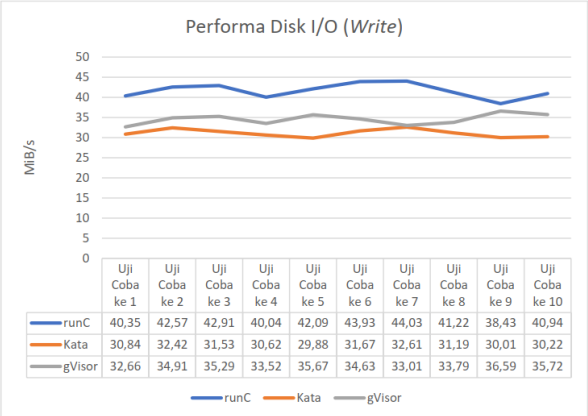
Figure 12 shows the results of memory performance tests conducted on three pods with different low level container runtimes. The results obtained are presented in graphical form stating Mebibyte/second (MiB/s). The difference between Mebibyte (MiB) and Megabyte (MB) lies in the number of bytes they have. One MiB has a total of 1,048,576 bytes, while one MB only has 1,000,000 bytes. This of course shows the difference between the use of MiB and MB, where MiB represents a larger unit of bytes when compared to MB. The results show the performance of the memory in running the sysbench command to provide the workload for the memory used. Sysbench will allocate a buffer, which by default is provided at 1 Kibi Byte (KiB). The number of sizes processed in this test is 102400 MiB. Executed commands will read and write to memory on each execution that is carried out randomly or sequentially. The more processing that can be done in seconds, the better the performance. Tests carried out ten times on each pod showed different results. **Figure 11** also shows that pods with RunC as the low level container runtime tend to be the fastest when compared to Kata container and gVisor. The highest value obtained is 5933.24 MiB/s using RunC, while the lowest value obtained is 1036.97 using the Kata container.

3. *Disk I/O Performance*



	Uji Coba ke 1	Uji Coba ke 2	Uji Coba ke 3	Uji Coba ke 4	Uji Coba ke 5	Uji Coba ke 6	Uji Coba ke 7	Uji Coba ke 8	Uji Coba ke 9	Uji Coba ke 10
runC	60,52	63,35	63,52	60,07	63,12	65,16	65,34	62,04	58,62	61,05
Kata	46,26	48,24	47,57	45,77	45,18	47,61	48,41	46,62	45,44	46,03
gVisor	48,99	52,37	53,46	50,97	54,20	52,25	49,72	51,51	55,19	54,35

Figure 13. Disk I/O performance test results (read)



	Uji Coba ke 1	Uji Coba ke 2	Uji Coba ke 3	Uji Coba ke 4	Uji Coba ke 5	Uji Coba ke 6	Uji Coba ke 7	Uji Coba ke 8	Uji Coba ke 9	Uji Coba ke 10
runC	40,35	42,57	42,91	40,04	42,09	43,93	44,03	41,22	38,43	40,94
Kata	30,84	32,42	31,53	30,62	29,88	31,67	32,61	31,19	30,01	30,22
gVisor	32,66	34,91	35,29	33,52	35,67	34,63	33,01	33,79	36,59	35,72

Figure 14. Disk I/O performance test results (write)

Figures 13 and **14** show the results of disk I/O performance tests performed on three pods with different low level container runtimes. The results obtained are presented in graphical form stating Mebibyte/second (MiB/s). There are two test results obtained, namely the condition when the disk reads and writes data. There are a total of 4 GB of files processed, where the files are divided into 128 files as shown in **Figure 15**.

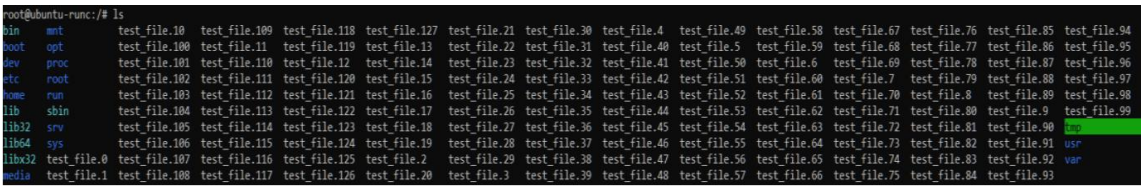
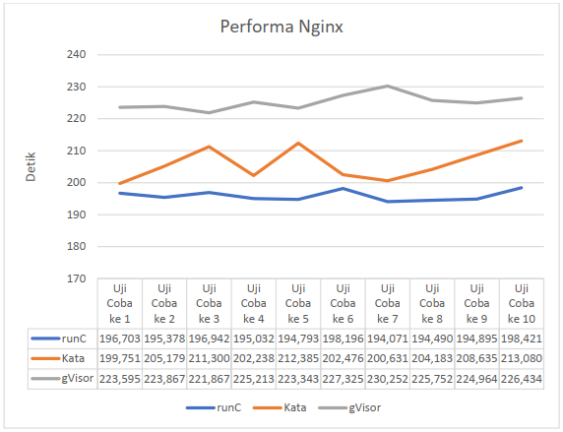


Figure 15. 128 test files

4. Performa Nginx



	Uji Coba ke 1	Uji Coba ke 2	Uji Coba ke 3	Uji Coba ke 4	Uji Coba ke 5	Uji Coba ke 6	Uji Coba ke 7	Uji Coba ke 8	Uji Coba ke 9	Uji Coba ke 10
runC	196,703	195,378	196,942	195,032	194,793	198,196	194,071	194,490	194,895	198,421
Kata	199,751	205,179	211,300	202,238	212,385	202,476	200,631	204,183	208,635	213,080
gVisor	223,595	223,867	221,867	225,213	223,343	227,325	230,252	225,752	224,964	226,434

Figure 16. Nginx performance test results

Figure 16 shows the results of the nginx performance test conducted on three pods with different low level container runtimes. The results obtained are presented in graphical form which states the time needed by nginx as a web server to process requests received. Apache Benhmark is used in the nginx test which is done by using the "-n" parameter to determine how many times the web server is accessed. Testing is done by adding parameters 10,000 to "-n" and 100 to "-c" which indicate the number of URLs accessed at the same time. The results obtained from the tests carried out show the performance of nginx, namely the time needed to serve requests. The faster the time needed by nginx to process requests, the better the performance. Tests that were carried out ten times for each pod showed that pods with RunC as the low level container runtime tended to be the fastest in processing requests. The shortest time obtained was 194.071 seconds using RunC, while the slowest time was 230.252 seconds using gVisor.

E. Reporting

1. Performace Analysis

From the results of tests carried out on several parameters to determine the performance and security of pods running with different low level container runtimes, it was found that the results obtained were different. On

CPU performance after ten times of testing, the average CPU performance is 10.1493 seconds (RunC); 10.2132 seconds (Kata); and 10.1823 seconds (gVisor), as shown in **Figure 17**. The average results of tests conducted on CPU performance show that RunC has the best performance, when compared to Kata container and gVisor. This shows that the additional layer of isolation performed on the Kata container decreases CPU performance, where the CPU works harder when compared to RunC which runs on the same kernel as the host. Isolation is done on Kata, namely by virtualizing the CPU, so that it seems to be a new, independent system to be used.

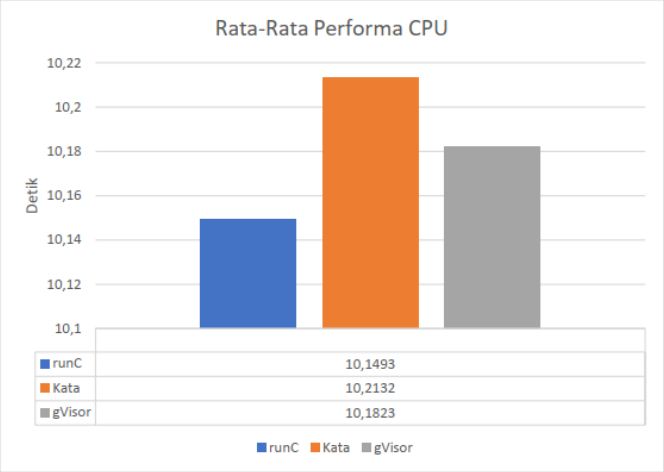


Figure 17. Average CPU performance test results

Ten times testing on memory performance, obtained an average memory performance of 5910.45 MiB/s (RunC); 1043.23 MiB/s (Kata); and 5477.72 MiB/s (gVisor), as shown in **Figure 18**. The average results of tests conducted on memory performance show that RunC has the best performance, when compared to Kata container and gVisor. Virtualization performed by Kata container shows the lowest memory performance. This is different from gVisor which, although it applies isolation to running pods, can still show good memory performance, where the results are not far from those obtained by RunC.

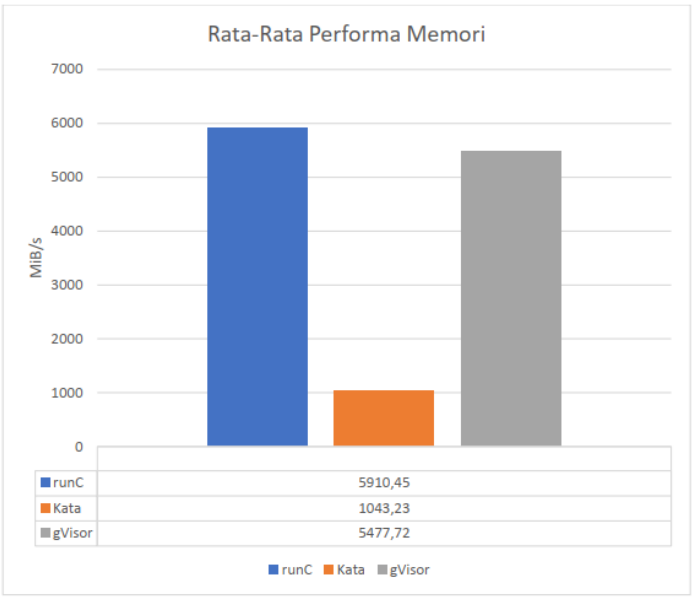


Figure 18. Average Memory performance test results

After ten tests on disk I/O performance, an average read performance of 62.28 MiB/s (RunC) was obtained; 46.71 MiB/s (Kata); and 52.3 MiB/s (gVisor), as shown in Figure 19. On write performance, the average yield is 41.65 MiB/s (RunC); 31.1 MiB/s (Word); and 34.58 MiB/s (gVisor), as shown in Figure 20. The average results of tests conducted on disk I/O performance both read and write show that RunC has the best performance, when compared to Said container and gVisor. The virtualization performed by the Kata container results in the highest reduction in memory performance. This can happen because a pod that uses Kata container will create a separate storage layer between the guest and the host. There is also an issue with the buffer cache used by Kata container and gVisor due to their state applying additional isolation.

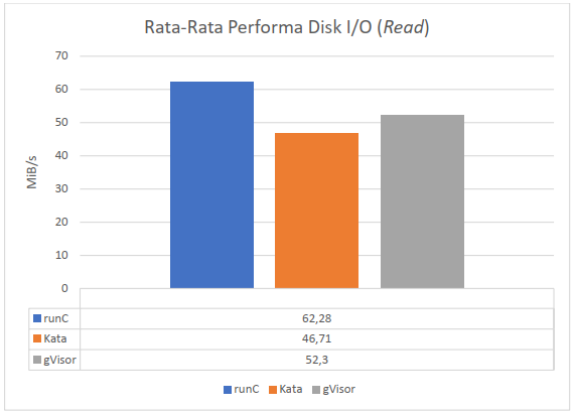


Figure 19. Average disk I/O performance (read)

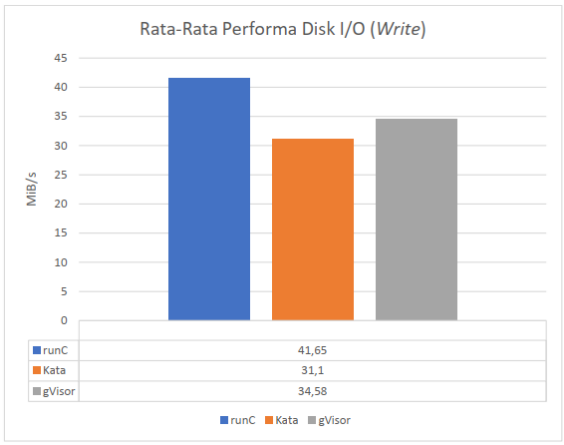


Figure 20. Average disk I/O performance (write)

On network performance after ten times of testing, the average memory performance is 5910.45 MiB/s (RunC); 1043.23 MiB/s (Kata); and 5477.72 MiB/s (gVisor), as shown in **Figure 21**. The average results of tests conducted on network performance show that RunC has the best performance, when compared to Kata container and gVisor. The isolation mechanism implemented by Kata container and gVisor has a significant impact on the network performance obtained. This is a performance issue that needs to be adjusted so that the difference in performance is not too great. One of the adjustments can be made by replacing the CNI used.

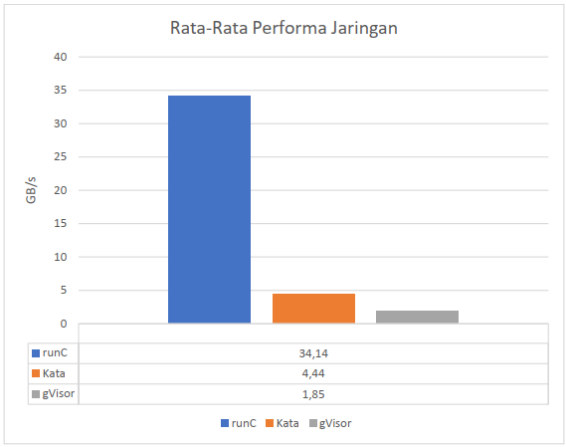


Figure 21. Average network performance test results

In ten times of testing conducted on the nginx web server on each pod running with a different low level container runtime, the average performance was 195.892 seconds (RunC); 205.986 seconds (Kata); and 225.261 seconds (gVisor) which can be seen in **Figure 22**. The average results of the tests conducted show that RunC has the best performance, when compared to Kata container and gVisor. Pods with RunC require the least average

time to receive and process 10,000 requests which are parameters to the tests performed. The difference in the results obtained is very closely related to the network architecture provided by each low level container runtime used. It can be seen that gVisor which uses netstack which is handled by Sentry has the lowest ability to receive and process requests in the tests carried out. The use of this additional layer causes a bottleneck to occur on the network used by gVisor.

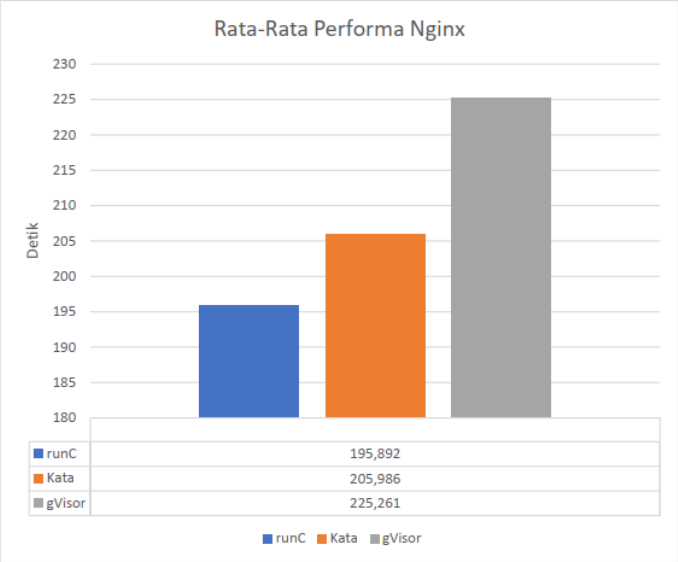


Figure 22. Average nginx performance test results

2. Security Analysis

The security tests that have been carried out produce data related to the differences in kernels used on hosts and pods running on the Kubernetes cluster. **Table 3** shows that the kernel used by the host (worker node) is the same as that used by the pod running using RunC. This is of course dangerous, if a pod that uses RunC as a low level container runtimes runs in root mode because there is a loophole to exploit the host where a pod is running.

Table 3. List of kernels

No	Type	Kernel
1	*host* (worker *node*)	Linux master 5.4.0-121-generic #137-Ubuntu SMP Wed Jun 15 13:33:07 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
2	RunC	Linux master 5.4.0-121-generic #137-Ubuntu SMP Wed Jun 15 13:33:07 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
3	Kata	Linux ubuntu-kata 5.15.26.*container* #1 SMP Wed Jun 8 16:46:19 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
4	gVisor	Linux ubuntu-gvisor 4.4.0 #1 SMP Sun Jan 10 15:06:54 PST 2016 x86_64 x86_64 x86_64 GNU/Linux

Further security testing was carried out to prove the existing loophole. The goal is to create a new directory on the host via the running pod. Another test was conducted by attempting to run the reboot host command from a pod running on a Kubernetes cluster. The results obtained can be seen in **Table 4** where the vulnerabilities found in pods running using RunC show vulnerabilities that are dangerous when exploited. An attacker can execute commands on the host via the currently running pod. Some examples of commands that can be executed are creating a new directory, viewing running processes, rebooting, and so on. In fact, the worst thing that can happen is that an attacker can shut down a host that acts as a worker node. This of course will reduce the performance of the Kubernetes cluster in running its services due to the reduced number of worker nodes used. The entire Kubernetes cluster may also not work, if the exploit successfully paralyzes all worker nodes used to run the service.

Table 4. Security test results

No	*Container* Runtime	Isolation	Description
1	RunC	No	Pod successfully modified *host*
2	Kata	Lightweight VM	Pod not successfully modified *host*
3	gVisor	Sandboxing	Pod not successfully modified *host*

Conclusion

The performance comparison between the implementations of RunC, Kata Containers, and gVisor reveals significant differences, particularly in terms of CPU, memory, disk I/O, network, and nginx performance, with pods utilizing RunC as the low-level container runtime demonstrating the best overall performance. A notable distinction is observed in network performance, where the maximum speed capacity achievable by Kata Containers and gVisor is considerably smaller compared to RunC. However, it is crucial to note that while RunC exhibits superior performance, it is the only low-level container runtime tested that has shown vulnerabilities from a security standpoint. This vulnerability arises due to the lack of isolation from the host and the utilization of the same kernel as the host. Some exploitations that can be carried out include executing commands such as mkdir, ls, and reboot.

Acknowledgement

Thanks to National Cyber and Crypto Polytechnic for support of this research and all author for great colaboration.

References

[1] C. Itron, S. Release, C. Delivery, C. D. Benefi, A. Software, and D. Methodology, "Continuous Delivery Software Release Methodology."

[2] K. H. Brendan Burns, Joe Beda, *Kubernetes: Up and Running*, 2nd Editio. Calfornia: O'Reilly Media, Inc., 2019.

[3] R. Kumar and B. Thangaraju, "Performance Analysis Between RunC and Kata Container Runtime," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–4. doi: 10.1109/CONECCT50063.2020.9198653.

[4] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia Comput. Sci.*, vol. 175, no. 2019, pp. 157–164, 2020, doi: 10.1016/j.procs.2020.07.025.

[5] A. Randazzo and I. Tinnirello, "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 209–214. doi: 10.1109/IOTSMS48152.2019.8939164.

[6] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, 2014, doi: 10.1109/MCC.2014.51.

[7] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Modelling Performance & Resource Management in Kubernetes," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, 2016, pp. 257–262.

[8] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, "Performance evaluation of container runtimes," *CLOSER 2020 - Proc. 10th Int. Conf. Cloud Comput. Serv. Sci.*, no. Closer, pp. 273–281, 2020, doi: 10.5220/0009340402730281.

[9] T. V Doan *et al.*, "Containers vs Virtual Machines: Choosing the Right Virtualization Technology for Mobile Edge Cloud," in *2019 IEEE 2nd 5G World Forum (5GWF)*, 2019, pp. 46–52. doi: 10.1109/5GWF.2019.8911715.

[10] T. Siddiqui, S. A. Siddiqui, and N. A. Khan, "Comprehensive Analysis of Container Technology," in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, 2019, pp. 218–223. doi: 10.1109/ISCON47742.2019.9036238.

[11] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: A State-of-the-Art Review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, 2019, doi: 10.1109/TCC.2017.2702586.

[12] M. Luksa, *Kubernetes in Action*. Manning Publications, 2018.

[13] G. Sayfan, *Mastering Kubernetes: Large scale container deployment and management*. Birmingham: Packt Publishing, 2017.

[14] "Cluster Architecture | Kubernetes." https://kubernetes.io/docs/concepts/architecture/ (accessed Jan. 22, 2024).

[15] K. Lee, J. Kim, I.-H. Kwon, H. Park, and C.-H. Hong, "Impact of Secure Container Runtimes on File I/O Performance in Edge Computing," *Appl. Sci.*, vol. 13, no. 24, p. 13329, 2023, doi: 10.3390/app132413329.

[16] Z. Yu, "The Application of Kata Containers in Baidu AI Cloud," no. October, 2019.

[17] "What is gVisor? - gVisor." https://gvisor.dev/docs/ (accessed Jan. 22, 2024).

[18] E. G. Young, P. Zhu, T. Caraza-harter, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "The True Cost of Containing : A gVisor Case Study".

[19] A. M. Potdar, D. G. Narayan, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," *Procedia Comput. Sci.*, vol. 171, no. 2019, pp. 1419–1428, 2020, doi: 10.1016/j.procs.2020.04.152.

[20] V. Aggarwal and B. Thangaraju, "Performance Analysis of Virtualisation Technologies in NFV and Edge Deployments," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–5. doi: 10.1109/CONECCT50063.2020.9198367.

[21] B. Balarajah; C. Rossenhoevel; B. Monkman, "Benchmarking Methodology for Network Security Device Performance," 2022.

[22] A. Akinshin, *Pro .NET Benchmarking: The Art of Performance Measurement*, 1st ed. 2019.

[23] M. Reeves, "Investigating Escape Vulnerabilities in Container Runtimes," Purdue University Graduate School, 2021.