The Cost of Container Runtime Security

Author: Luke Stigdon, contact@lukestigdon.com
Advisor: *Dr. Johannes Ullrich*

Accepted: 5/13/2024

Abstract

Containerization has fundamentally changed how applications are developed, deployed, and managed. Containers provide a lightweight, portable alternative to traditional virtual machines and their associated infrastructure. Unfortunately, containers provide less isolation than virtual machines, which has led to security trade-offs that organizations must consider. Various runtimes and tools have emerged to bring added layers of security to containerized environments. This paper aims to analyze the performance cost these tools incur. Through an analysis of existing methods and a comprehensive set of benchmarks and metrics, this paper will explore the benefits – and drawbacks – of runtime security tools and container sandboxes. By analyzing and understanding the inner workings of the available tools, administrators and developers can make informed decisions about how they build their infrastructure. Additionally, this paper will highlight areas of future research and discuss evolved challenges in container security.

1. Introduction

Virtual machines (VMs) are the fundamental building blocks that power most applications today. VMs are powered by a program known as a hypervisor that can expose the hardware of a physical system to multiple virtual machines. Each virtual machine runs an operating system entirely separate from the host system and any other virtual machines running on the same hardware. The overhead of running an entire operating system nested inside of another operating system does have performance implications. Still, those drawbacks are usually greatly outweighed by the scaling, operational, and security benefits that VMs provide.

Despite the numerous benefits VMs offer, they are a cumbersome level of abstraction for many workloads. The overhead incurred by running a VM is high relative to the resources many applications require. Many small services can run on a single virtual machine, but for security purposes and management reasons, it's often better to keep them separate. Before containers, as known today, many technologies were developed to provide application isolation at the OS level and allow administrators to prevent unwanted interference – malicious or otherwise – between applications running on the same host.

One of the earliest examples of OS-level virtualization was the `chroot`[1] system call, developed by Bell Laboratories and introduced in Unix Version 7. `Chroot` allowed a process to set a specific starting point for paths beginning with "`/`." There is little documentation regarding the original use of `chroot`. Still, in 1994, the Computer Systems Research Group at the University of California, Berkley added the `chroot`[2] command line tool to the 4.4BSD operating system. In an accompanying book, the authors state that the syscall was primarily used to set up restricted access to a system (Bostic et al., 1996, p. 38). For example, if `chroot` has been called with the directory `/var/application` and the application tries to read `/etc/shadow`, it would automatically translate to `/var/application/etc/shadow`. In 2000, FreeBSD 4.0 – a

[1] /usr/man/man2/chdir.2 – *Unix Version 7*

[2] /usr/share/man/cat8/chroot.0 – *Berkley Software Distribution 4.4*

derivative of BSD (Lucas, 2019, p. xxxvi) – introduced the `jail`[3] system call, which is built on top of `chroot` and provides additional security guarantees. Similarly, the Linux operating system developed its own OS-level virtualization. Most notably, Linux Containers and Docker (Hildred, 2015).

Large organizations with complex systems and sprawling infrastructure found that neither VMs nor containers provided the right level of abstraction for their workloads[4,5,6]. Because of this, many companies have developed custom orchestration systems to automatically provision, schedule, and manage workloads using a combination of VMs and containers (Verma et al., 2015, p. 12). One of the most influential orchestration systems was an internally developed tool at Google called Borg. It allowed users to define 'jobs' using a declarative syntax, and Borg would take care of scheduling, execution, availability, and monitoring (Verma et al., 2015). Borg directly inspired the open-source container orchestrator Kubernetes.

Containers aren't inherently insecure; they simply lack the same level of security that virtual machines can provide. The deployment speed and scale container orchestrators enable is a double-edged sword for modern security teams. Kubernetes is a complex service that requires careful implementation to be effective *and* secure without considering the security of applications that developers will run inside a deployed cluster. Additionally, beyond standard security hygiene and best practices, many workloads require an added level of security and isolation due to specific risks they entail, such as multi-tenant environments.

1.1. CI/CD Pipelines

One of the most commonly containerized workloads is CI/CD pipelines. Companies like GitHub and GitLab that offer pipelines as a service to users and

[3] https://man.freebsd.org/cgi/man.cgi?query=jail&sektion=2&manpath=FreeBSD+4.0-RELEASE

[4] https://kubernetes.io/case-studies/spotify

[5] https://kubernetes.io/case-studies/squarespace

[6] https://kubernetes.io/case-studies/box

organizations need more control over what code executes, and the sheer volume of pipelines makes it impossible for them to review every individual execution. To improve the security posture of their pipelines, they use temporary containers or virtual machines spawned for each job in each pipeline. CI/CD pipelines are generally not time or resource-sensitive, making them good candidates for adding additional runtime security. The small amount of overhead incurred by many tools would easily go unnoticed. Additionally, adding observability and security to CI/CD processes has become increasingly important as threat actors target software supply chains.

1.2. "Serverless" Functions

Unlike their name suggests, serverless functions do run on a server. To the end user, they are considered "serverless" because instead of the typical application model that relies on a long-running process that is deployed to a server or set of servers, "serverless" functions are snippets of code that are run on-demand for short periods in response to a predefined trigger. The trigger could be a user uploading a profile picture, which could trigger a function to compress and pre-convert the image to the various sizes required by the application. Serverless functions can also be scheduled to run at specific intervals or in response to certain events. Developers typically only pay for the time a function is running, but platforms and providers still need to run a fleet of servers always ready to dispatch those functions. For cost and complexity reasons, providers don't let users decide *where* their function runs, only that it will run. Hundreds of arbitrary scripts from developers anywhere in the world could execute on the same server simultaneously. Security controls, specifically sandboxing, enable providers to guarantee a level of isolation that will prevent any developer's code – even malicious code – from interfering or tampering with the other functions running on the same host. Google Cloud Platform specifically developed a technology that will be discussed later – gVisor – to secure their serverless environments (Manor, 2018).

1.3. Data Science and Machine Learning

Data science teams rely heavily on Python and Jupyter notebooks, which allows them to easily execute, organize, and visualize the results of their code. While notebooks can be locally run and shared via version control, many teams leverage an additional tool

called Jupyter Hub, which is a browser-based interface that allows them to collaborate and execute code in a shared environment. These shared environments are commonly run within containers or VMs and provide a way to give users a common pre-configured environment. While administrators can tightly control the libraries that are available since they control the images on the backend, sandboxing provides an added layer of security if a zero-day vulnerability is found within the container runtime or malicious code happens to have been found in a previously approved library (Hafner et al., 2021; Kaplan & Qian, 2021; Schlueter, 2016; Zahan et al., 2022).

1.4. Learning

Many languages today offer the ability to test your code in a browser via "playgrounds."[7,8] These environments will upload code to be compiled and run on a remote server and then return the output to the user's browser. Sandboxes add an additional layer of security to the servers handling the code. There are also learning services such as HackerRank and Codewars, which offer hundreds of interactive exercises for dozens of languages. These services work similarly, allowing users to upload the code they wrote, which is compiled and executed. Without proper sandboxing, both examples would be incredibly risky for any organization since they would purposefully allow arbitrary remote code execution, which can be incredibly dangerous.

2. Runtime security

The key to running code safely is controlling exactly what resources the environment has access to. Improperly configured containers can allow the processes running inside them to execute with higher permissions than they should normally have and glean additional information about the underlying host. In many cases, this is easier said than done. Even trivial applications can require access to hundreds of files and directories to load libraries, read and write data, and otherwise interact with a system. Access to the Internet or other network resources is another common requirement.

[7] https://play.rust-lang.org

[8] https://go.dev/play

Logging, auditing, and controlling these interactions – especially at scale – is a difficult problem, but it is vital to maintaining a robust security posture.

Security teams must often make trade-offs between following security best practices and meeting stakeholders' requirements. There are sometimes dozens of products to evaluate when looking for a solution to a specific problem. This level of choice gives administrators the flexibility they need to meet the requirements specific to their organization. However, it also makes reviewing and understanding every available option difficult. This paper aims to make understanding the ever-changing landscape of choices easier by focusing on the most common underlying technologies that power runtime security tools.

2.1. Hardened Runtimes

One of the most straightforward methods to increase the security and isolation of containers is to use a security-focused runtime. The container runtime software is responsible for managing the lifecycle of container processes. It handles every event during a container's lifetime, from its initial creation to its final termination (Open Container Initiative, 2024, p. 6). This research will analyze two popular runtimes: gVisor and Kata Containers. Both projects are open source and implement the Open Container Initiative's (OCI) runtime specification. They also aim to provide an additional layer of security while providing efficiency levels similar to a standard container runtime.

gVisor is an open-source project, initially developed at Google (Lacasse, 2018). It adds an extra layer of isolation between containers and the host operating system through its OCI-compliant runtime `runsc`. Every container launched by `runsc` runs within a sandbox that has two additional processes[9]. A *Sentry* that intercepts and responds to system calls from the container, and a *Gopher* which provides restricted filesystem access. This model allows gVisor to effectively isolate the container as a VM while not requiring the same amount of overhead as a full virtual machine. The application itself is written in Go and is very memory and CPU-efficient; however, because it must translate

[9] https://gvisor.dev/docs

system calls before passing them to the host's kernel, certain tasks such as filesystem I/O and maintaining network connections have a noticeable overhead.

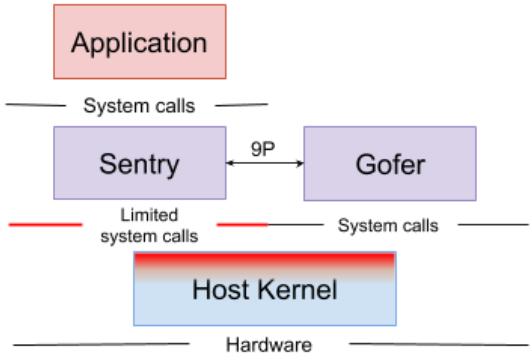

Figure 1 gVisor Architecture

Kata containers have the same goal as gVisor, but instead of acting as a shim, the kata runtime launches the container in a "lightweight" virtual machine[10]. Kata's OCI-compliant runtime is exposed through the `containerd-shim-kata-v2` command line tool. The runtime interacts with a hypervisor to launch a dedicated virtual machine for each container. Because Kata launches a VM for each container, it can guarantee the same level of isolation that VMs offer. The virtual machine images and hypervisors[11] that the Kata supports are highly optimized for containerized workloads. Still, it does incur a performance penalty due to the added overhead. By default, it uses QEMU, a type 2 KVM hypervisor, but it also supports newer, container-optimized Virtual Machine Monitors (VMM) such as Cloud Hypervisor and Firecracker. While a full analysis is out of scope for this paper, Firecracker was specifically developed by AWS for their Lambda and Fargate services to provide faster function execution and higher isolation via a purpose-built hypervisor (Gupta & Lian, 2018).

2.2. Linux Kernel Security Modules

Linux Kernel Security Modules, or simply Linux Security Modules (LSMs), are kernel modules loaded to enforce additional security controls on a system. The two most

[10] https://github.com/kata-containers/kata-containers/tree/3.3.0/docs/design/architecture

[11] https://github.com/kata-containers/kata-containers/blob/3.3.0/docs/hypervisors.md

prevalent security modules are AppArmor and SELinux, which implement forms of mandatory access control (MAC). On a standard Linux system, files and processes are associated with a user, group, and a "mode" that determines the permissions given to the owner, group, and any other users on the system[12]. The "mode" is expressed as a three-digit octal number where the first, second, and third digits correspond to the owner, group, and 'others,' respectively. Each number represents the permissions – read, write, or execute – that can be given to the specific entity in that position. Each permission has a numeric value of 1 (execute), 2 (write), or 4 (read). Adding them together can express multiple permissions. For example, 'read' and 'execute' (1+4) would be 5. A complete example of a real file might look like this:

```
-rw-r----- 1 root shadow 903 Mar 18 01:04 /etc/shadow
```

The permissions in octal would be 640, meaning the owner, `root`, can read and write to `/etc/shadow`, any user belonging to the group `shadow` can read the file, and everyone else would get permission denied. This system is known as Discretionary Access Control (DAC), and it is a straightforward and robust framework for administrators to control access to files and processes.

Mandatory Access Control adds an additional level of control by defining extra rules – sometimes called policies – for valid interactions between a *subject* and an *object* (Belim & Belim, 2018). LSMs implement this by processing additional rules and security attributes attached to files, processes, and users to make more informed permission decisions. AppArmor allows a 'profile' to be applied to a running process. This profile explicitly defines all the directories and files a process should be able to access. Because the permissions are explicitly defined, even if a user could escape a running process, e.g., a container, they would still be confined by the rules specified in the profile, and their access would remain limited. SELinux offers similar protections via security labels – called contexts – applied to every file and process on the system. The context of a process defines what it can and cannot access. For example, if a web server tried to read files that belonged to a database, SELinux would automatically block the web server's attempts unless a rule was added to allow the behavior. While these protections offered by LSMs

[12] https://tldp.org/LDP/intro-linux/html/sect_03_04.html

aren't as strong as the isolation of a virtual machine, they can still be an incredibly effective security measure if properly applied.

2.3. eBPF

Extended Berkley Packet Filters (eBPF) is another technology inside the Linux kernel that provides powerful networking, observability, and security features. BPF was initially introduced as the "BSD Packet Filter" for the BSD operating system in 1992 (McCanne & Jacobson, 1993). It provided a way for users to efficiently capture and filter network packets by implementing a BPF virtual machine within the kernel that could execute user-provided instructions via an interpreter. This novel approach had two benefits. First, it was highly efficient because it avoided the expensive operation of copying every network packet to a user-space program and instead executed the instructions directly in the kernel – only returning the packets matched by the filter. The second benefit was the safety added by an additional layer of verification the BPF virtual

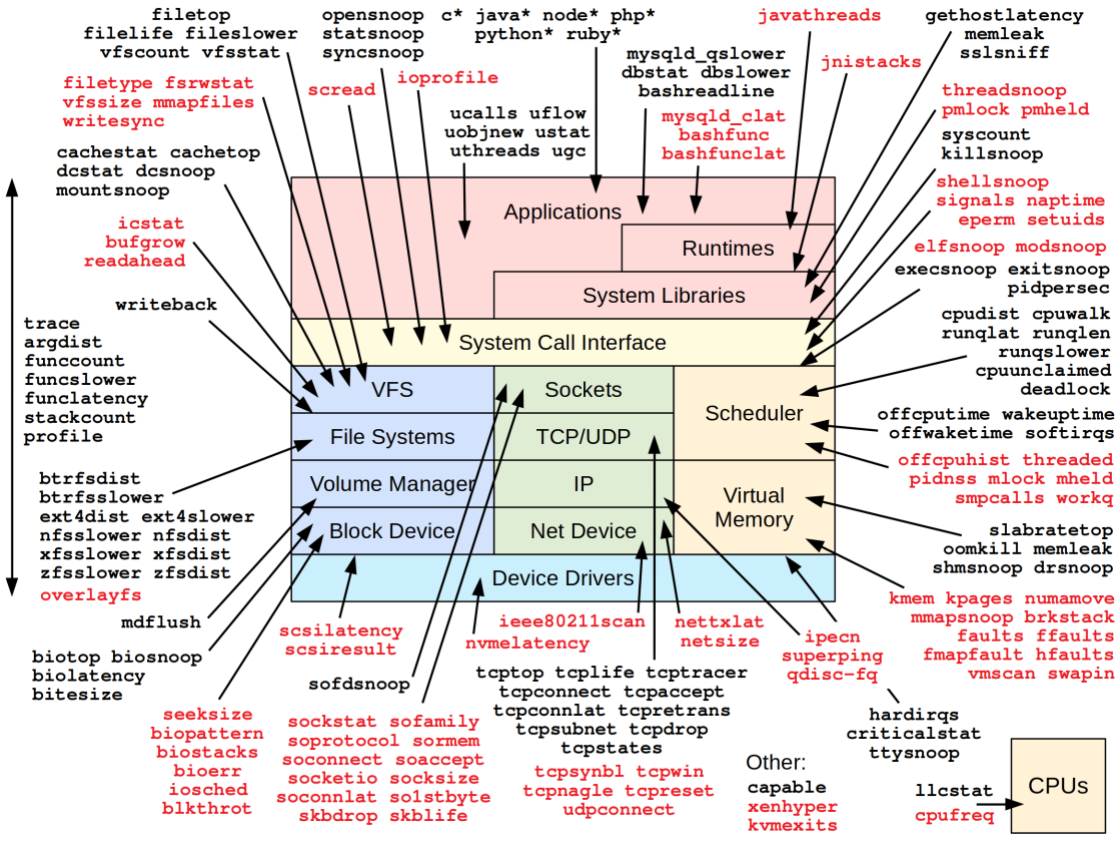

Figure 2 BPF Performance Tools

machine added. Before executing the user's instructions, the interpreter validates that the instructions are valid and safe.

BPF, as it exists today, is nearly indistinguishable from its humble beginnings as a simple packet filter. The technology has evolved into a general-purpose execution engine that allows users to write programs that can monitor virtually every aspect of the Linux operating system. The book "BPF Performance Tools: Linux System and Application Observability" by Brendan Gregg highlights the multitude of monitoring tools now powered by BPF (Gregg, 2019).

Despite its differences from "classic BPF" (the original implementation), eBPF maintains the same performance and safety guarantees as its predecessor. These guarantees are a large part of the success it has seen today. When faced with the choice of implementing a kernel module or a BPF program, BPF is usually the much better choice. Some of the most popular Kubernetes runtime security tools today are all powered by eBPF: Tracee[13], Falco[14], and Tetragon[15]. The way that these runtimes work is similar to gVisor. However, instead of inserting a process between the container and the host that intercepts system calls before they reach the kernel, the runtimes inject eBPF programs directly into the kernel that monitor for specific events and syscalls. This leads to much better performance with the added flexibility of being able to write custom rules similar to the "profiles" that LSMs use to deny or allow actions. Another benefit is the increased visibility eBPF provides. Rather than intercepting and blocking certain actions, these runtimes can feed events to external monitoring system which allows administrators to collect details metrics, improve incident response, and proactively respond to unexpected behaviors.

[13] https://github.com/aquasecurity/tracee

[14] https://github.com/falcosecurity/falco

[15] https://github.com/cilium/tetragon

3. Research Method

Each technology will be analyzed based on the isolation level it provides from the infrastructure it runs within and the performance cost that isolation incurs. Generally speaking, the higher the level of isolation, the higher the performance cost; however, due to the diverse nature of workloads and various optimizations that exist, e.g., GPU-optimized workloads may be less impacted by higher levels of isolation from the host since the calculations are offloaded to the GPU and certain tasks may perform actions that are more sensitive to network latency as opposed to system call bottlenecks.

A set of tests was chosen from the Phoronix Test Suite, a set of standardized benchmarks covering CPU, memory, and I/O intensive workloads.

3.1. Hardware and Software

The hardware selected for this test comprises four Turing RK1 Compute modules, each with 16 GB of memory and an 8-core 64-bit ARM processor[16]. The nodes will run a v1.29.2 Kubernetes cluster, and scheduling will be configured so that only one pod can run a node at a given time. Pods will be limited to consuming at most 4 CPU cores and 8 GB of memory. Each node is running the following software and versions:

Software	**Version**
Host Operating System	Ubuntu 22.04.4 LTS
Kernel	5.10.160-rockchip aarch64
AppArmor	3.0.4-2ubuntu2.3
containerd	1.7.13
runc	1.1.12
gVisor (runsc)	release-20240311.0
kata-runtime	3.3.0
qemu-system-aarch64	7.2.0
Tetragon	1.0.3

[16] https://docs.turingpi.com/docs/turing-rk1-specs-and-io-ports

The testing and result collection will be orchestrated via the Phoronix Test Suite, which is the tool developed by Phoronix Media to power OpenBenchmarking.org. The Pronix Test Suite contains hundreds of open-source benchmarks that cover dozens of different applications and performance tools. Leveraging an existing and widely deployed benchmarking tool will help reduce the chances of introducing testing biases and provide results that can be objectively compared against the millions of existing results hosted by OpenBenchmarking.org. The following tests will be executed as part of the test suite for this research:

Test	**Parameters**
pts/nginx-3.0.1	20 and 100 active connections
pts/redis-1.4.0	SET, GET, LPUSH, LPOP, and SADD with 50 parallel connections
pts/unpack-linux-1.2.0	N/A
pts/compress-zstd-1.6.0	Compression Level 3

4. Findings and Discussion

4.1.1. NGINX Requests per Second

This test was performed with two configurations, one with 20 active connections and another with 100. The goal was to measure the number of requests per second that a nginx server could handle. A higher number of requests per second is desirable. This test is primarily CPU-bound but involves lower-level system calls for establishing and maintaining TCP connections. Immediately, the impacts on performance that running containers within a sandbox become apparent. *Figure 2* below, shows the average number of requests per second the containers could handle. The requests per second were not significantly impacted by the number of concurrent connections tested. Still, the sandbox that the containers were running under caused a drastic change in the number of connections.

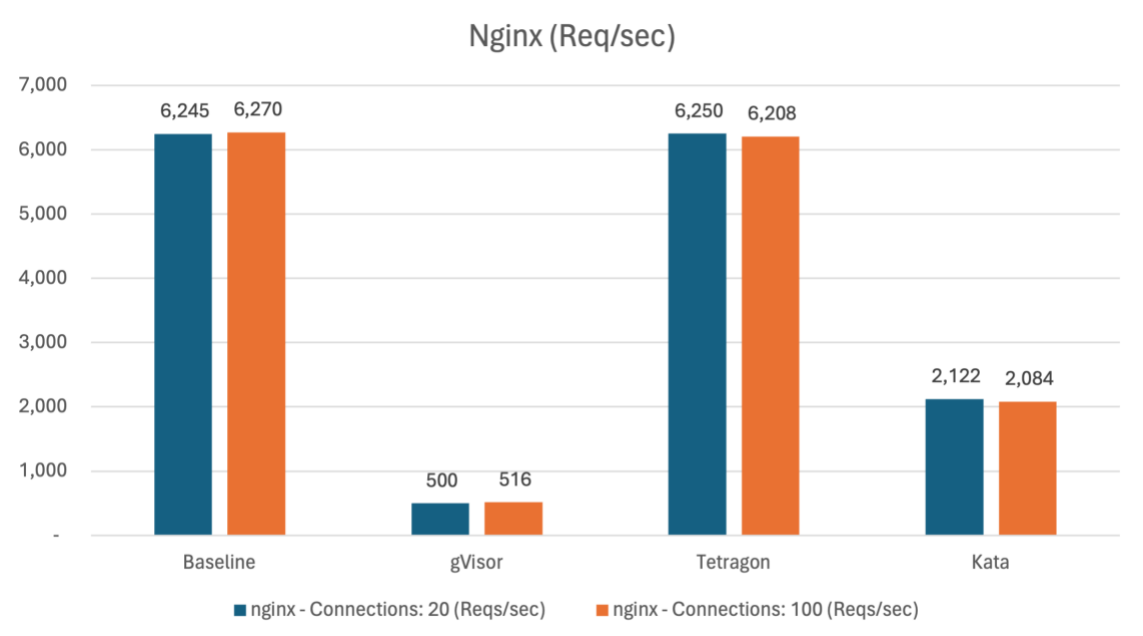

Figure 3 NGINX requests per second

Unfortunately for gVisor and Kata containers, the first test highlights their biggest weakness, handling network connections. The drastic difference here is the overhead added to the syscalls required for every request. In the case of the kata container, the inefficiencies of running a network stack within a virtual machine have a big impact on performance. Still, they are not nearly as large as the impact that gVisor imposes. gVisor suffers from the overhead the Sentry adds by needing to intercept and translate the

required syscalls for every request it handles. Tetragon performed similarly to a standard container, which is to be expected since the underlying mechanism – an eBPF program – is running directly inside the kernel. There does appear to be a slight drop in capacity for the 100 req/sec test, but the difference is negligible.

4.1.2. Zstandard Compression

Zstandard (zstd) is an open-source lossless compression algorithm developed at Facebook[17]. It was created as a fast and efficient compression algorithm focusing on real-time compression scenarios. This workload is CPU intensive and is an example of something a sandboxed application might have to deal with regularly. For example, a serverless function might be triggered to fetch compressed data from an object store. Serverless functions are typically short-lived and billed by the amount of time they run. They are also commonly executed within a sandbox to maintain security while achieving scale in multi-tenant environments.

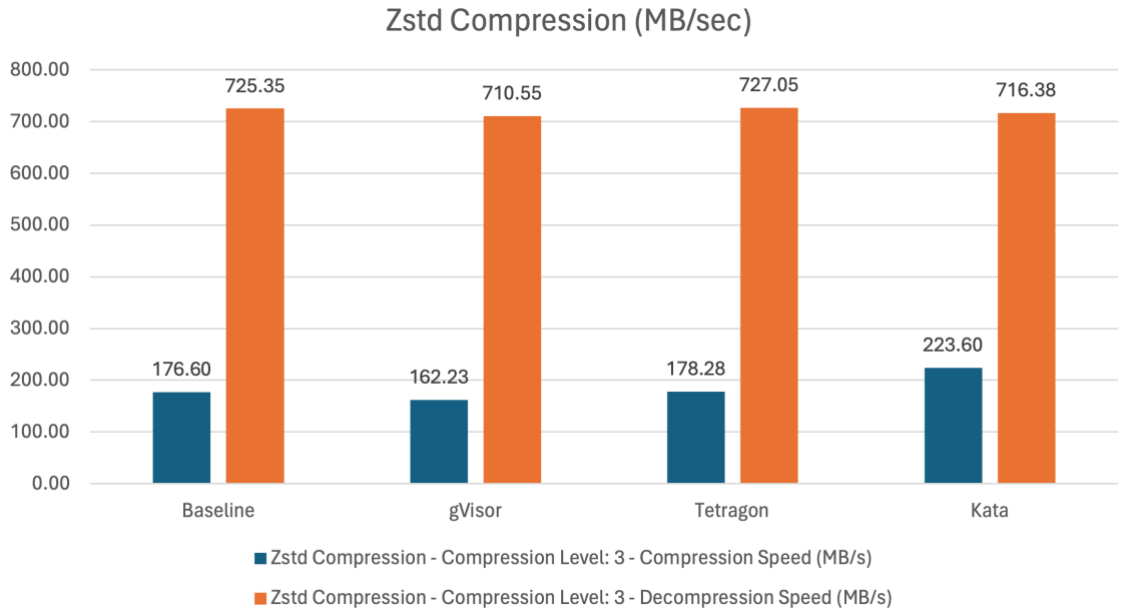

Figure 4 Zstandard Compression Throughput

Compared to the previous NGINX example, the results between the containers were practically indistinguishable. Kata containers even showed a slight improvement in

[17] https://github.com/facebook/zstd

compression speed. The cause of this improvement was unclear at the time of testing but may have been related to the fact that the QEMU virtual machines launched by the kata-runtime were using a more recent Linux kernel (6.1.62-126), which could have contained patches and optimizations that the host kernel did not have.

4.1.3. Redis

Redis is an in-memory key-value store used by many applications. It is favored for its speed and scalability, which enable it to handle millions of requests efficiently. Even in this small test environment, the number of requests the benchmark tools were able to generate was impressive. Unfortunately, this test highlighted the weakness of kata containers and gVisor, which is similar to the NGINX benchmark.

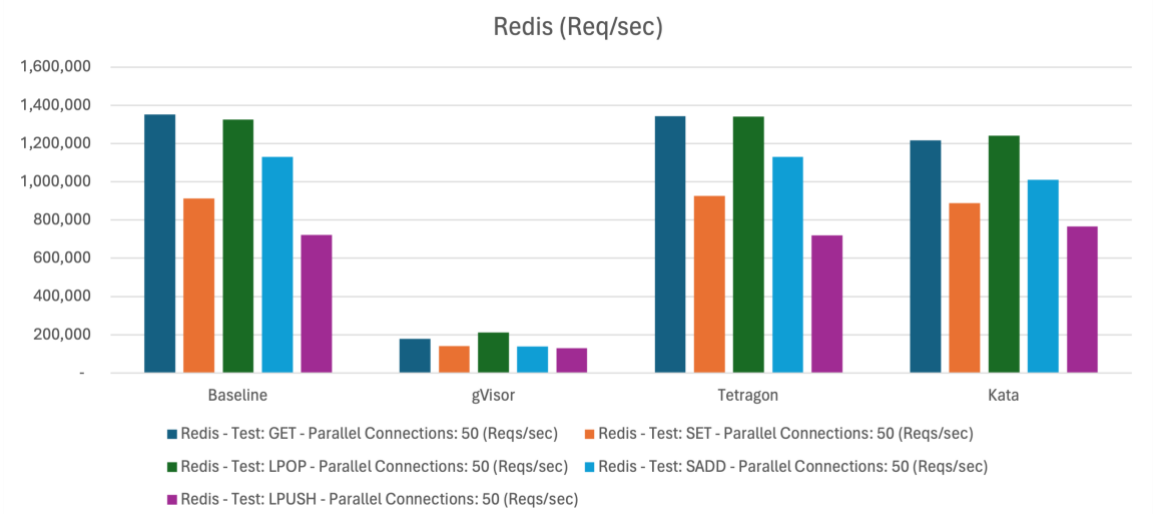

Figure 5 Redis Requests per Second

gVisor took another large performance hit in this scenario due to the overhead introduced by its syscall filtering. Interestingly, kata containers did not appear to take as much of a performance hit. They performed marginally lower than the Baseline and Tetragon benchmark runs but not as significantly as the NGINX benchmark. As was mentioned previously, this is not something that would typically need to run in a sandbox since Redis is a "trusted" application and doesn't run any arbitrary code, but it does highlight the importance of understanding an organization's applications and testing for its specific environment. It would have been easy to assume that Kata might have suffered a similar penalty in this test based on the NGINX benchmark. However, this is not the case.

4.1.4. Unpacking the Linux Kernel

The final benchmark focused on filesystem performance. While extracting a large tarball isn't a common occurrence, it is a good way to get a general idea of the performance of a filesystem. The compressed source code for the Linux kernel is hundreds of megabytes and contains thousands of files. Extracting those contents is a quick way to identify file operation bottlenecks. This benchmark measures the amount of time it takes (in seconds) to extract the source tarball, lower values are considered better.

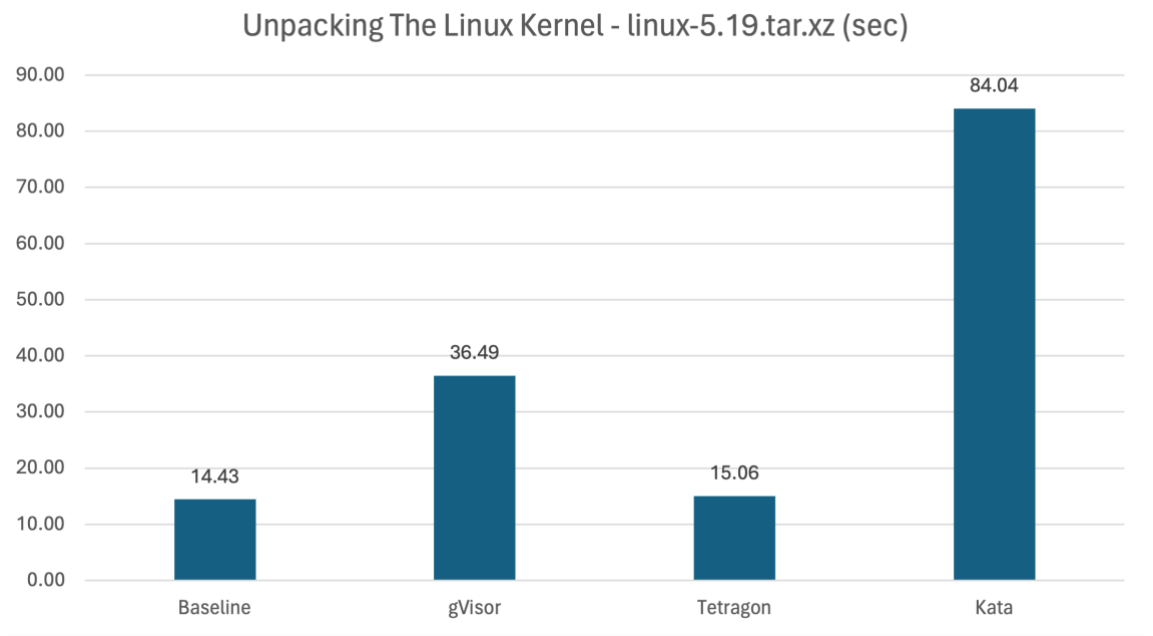

Figure 6 Unpacking the Linux Kernel

The results of this test were initially surprising. The benchmark runs with no sandboxing and the runs with Tetragon performed similarly – and had the best performance – but the gVisor and Kata runs showed that file operations were significantly impacted. Kata containers also had a large variance between runs, with the slowest taking 99 seconds and the fastest run still taking 66 seconds. None of the other runtimes had such wide variances. Upon further investigation, the performance impact seemed to derive from QEMU's implementation of shared host paths.

To avoid any bottlenecks or testing complexities that an external storage provider could introduce, the tests were performed in an `emtpyDir`[18] volume, which mounts a temporary directory from the host directly into the container. The directory on the host is ephemeral, so the data does not persist when a pod restarts. However, there should not be any filesystem overhead other than the overhead introduced by the sandbox. Additionally, the ephemeral nature of the directory ensured that each test run had a clean environment. Unfortunately, since kata containers are run inside a virtual machine the way that QEMU and, by extension, the kata runtime mounts directories from the host into the VM by setting up a virtual network and using the 9P network protocol to communicate between the guest and the host. The 9P protocol is simple and effective but has performance limitations exposed by this test. Switching to `/tmp` within the container improved consistency and performance, but the average runtime was still 73 seconds, which was considerably slower than even gVisor.

5. Recommendations and Implications

As this research has demonstrated, there are many factors to consider when analyzing what solutions might work for your situation. It is also important to remember that the solutions presented here are not a silver bullet and only represent individual layers that should be included as part of a broader security strategy for an organization's environment. Hardened runtimes such as gVisor and Kata Containers – while costly for specific workloads – should be seriously considered in multi-tenant environments or where high isolation levels are required. While the tests today exposed some of their weaknesses, they have continued to improve over time and will only improve from now on. For Kata Containers specifically, there was no time to delve into further optimizations, but the runtime itself is highly configurable and can most likely fit many workloads not addressed here.

One of the most promising developments discovered during the research for this paper is the progress and capabilities shown by BPF. While hardened runtimes are

[18] https://kubernetes.io/docs/concepts/storage/volumes#emptydir

necessary in some scenarios, the added security and observability provided by BPF-based runtime security tools are hard to overlook. They should be considered a standard addition to any new environment. Tetragon specifically positioned itself as a complement to existing LSMs and provides powerful capabilities that enable administrators to respond proactively to emerging threats.

6. Conclusion

This paper has shown that there are many available tools to choose from when administrators need to add additional layers of defense to their infrastructure. The container runtime security landscape constantly evolves, and each new technology aims to be better than the last but inevitably involves considering some trade-offs. Nevertheless, this paper's examination found that because of that constant improvement, developers and administrators should be able to find the right balance of security and performance for their needs. Additionally, our research identified eBPF as a major area of future growth and research. The observability, control, and performance offered by BPF can allow an administrator to monitor and secure their infrastructure through a common interface and allow them to react to emerging threats in a way that was not possible previously.

References

Belim, S. V., & Belim, S. Yu. (2018). Implementation of Mandatory Access Control in Distributed Systems. *Automatic Control and Computer Sciences*, *52*(8), 1124–1126. https://doi.org/10.3103/S0146411618080357

Bell Laboratories. (1979). *Unix* (Version 7) [Computer software]. https://www.tuhs.org/cgi-bin/utree.pl?file=V7

Bostic, K., Karels, M. J., & Quarterman, J. S. (1996). *The design and implementation of the 4.4BSD operating system* (M. K. McKusick, Ed.). Addison-Wesley.

Computer Systems Research Group. (1994). *Berkley Software Distribution* (4.4) [Computer software]. https://www.tuhs.org/cgi-bin/utree.pl?file=4.4BSD

Gregg, B. (2019, July 15). BPF Performance Tools: Linux System and Application Observability. *Brendan Gregg's Blog*. https://www.brendangregg.com/blog/2019-07-15/bpf-performance-tools-book.html

Gupta, A., & Lian, L. (2018, November 27). Announcing the Firecracker Open Source Technology: Secure and Fast microVM for Serverless Computing. *AWS Open Source Blog*. https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/

Hafner, A., Mur, A., & Bernard, J. (2021). *Node package manager's dependency network robustness* (arXiv:2110.11695). arXiv. http://arxiv.org/abs/2110.11695

Hildred, T. (2015, August 28). The History of Containers. *Red Hat Blog*. https://www.redhat.com/en/blog/history-containers

Kaplan, B., & Qian, J. (2021). *A Survey on Common Threats in npm and PyPi Registries* (arXiv:2108.09576). arXiv. http://arxiv.org/abs/2108.09576

Lacasse, N. (2018, May 2). Open-sourcing gVisor, a sandboxed container runtime. *Google Cloud Platform Blog*. https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime

Lucas, M. W. (2019). *Absolute FreeBSD: The complete guide to FreeBSD* (3rd edition). No Starch Press.

Manor, E. (2018, July 24). Bringing the best of serverless to you. *Google Cloud Platform Blog*. https://cloudplatform.googleblog.com/2018/07/bringing-the-best-of-serverless-to-you.html

McCanne, S., & Jacobson, V. (1993, January). The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

Open Container Initiative. (2024). *Open Container Initiative Runtime Specification* (1.2.0). https://github.com/opencontainers/runtime-spec/releases/tag/v1.2.0

Schlueter, I. (2016, March 26). Kik, left-pad, and npm. *Npm Blog*. https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm

The Kata Authors. (n.d.). *Kata Containers* (3.3.0) [Computer software]. OpenInfra Foundation. https://github.com/kata-containers/kata-containers

The Kubernetes Authors. (n.d.-a). Case study: Box. *Kubernetes User Case Studies*. Retrieved May 11, 2024, from https://kubernetes.io/case-studies/box/

The Kubernetes Authors. (n.d.-b). Case study: Spotify. *Kubernetes User Case Studies*. Retrieved May 11, 2024, from https://kubernetes.io/case-studies/spotify/

The Kubernetes Authors. (n.d.-c). Case study: Squarespace. *Kubernetes User Case Studies*. Retrieved May 11, 2024, from https://kubernetes.io/case-studies/squarespace/

The Phoronix Test Suite Authors. (n.d.). *Phoronix Test Suite* (10.8.4) [Computer software]. Phoronix. https://github.com/phoronix-test-suite/phoronix-test-suite

The Tetragon Authors. (n.d.). *Tetragon* (1.0.3) [Computer software]. Cloud Native Computing Foundation. https://github.com/cilium/tetragon

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. *Proceedings of the Tenth European Conference on Computer Systems*, 1–17. https://doi.org/10.1145/2741948.2741964

Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., & Williams, L. (2022). What are Weak Links in the npm Supply Chain? *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 331–340. https://doi.org/10.1145/3510457.3513044

Appendix

Results

Baseline - Test Results by Node	kube01	kube02	kube03	kube04	Average
Unpacking The Linux Kernel - linux-5.19.tar.xz (sec)	15.28	14.49	13.95	14.00	**14.43**
Zstd Compression - Compression Level: 3 - Compression Speed (MB/s)	177.80	176.20	176.10	176.30	**176.60**
Zstd Compression - Compression Level: 3 - Decompression Speed (MB/s)	721.30	715.40	724.90	739.80	**725.35**
Redis - Test: GET - Parallel Connections: 50 (Reqs/sec)	1,347,345	1,325,022	1,367,589	1,370,072	**1,352,507**
Redis - Test: SET - Parallel Connections: 50 (Reqs/sec)	921,270	893,255	912,752	924,104	**912,846**
Redis - Test: LPOP - Parallel Connections: 50 (Reqs/sec)	1,315,050	1,313,340	1,324,483	1,345,346	**1,324,555**
Redis - Test: SADD - Parallel Connections: 50 (Reqs/sec)	1,115,189	1,122,310	1,122,221	1,157,980	**1,129,425**
Redis - Test: LPUSH - Parallel Connections: 50 (Reqs/sec)	722,436	715,132	715,621	735,950	**722,285**
nginx - Connections: 20 (Reqs/sec)	6,122	6,236	6,186	6,437	**6,245**
nginx - Connections: 100 (Reqs/sec)	6,080	6,359	6,174	6,467	**6,270**

gVisor - Test Results by Node	kube01	kube02	kube03	kube04	Average
Unpacking The Linux Kernel - linux-5.19.tar.xz (sec)	36.01	39.07	33.66	37.22	**36.49**
Zstd Compression - Compression Level: 3 - Compression Speed (MB/s)	159.80	162.40	161.20	165.50	**162.23**
Zstd Compression - Compression Level: 3 - Decompression Speed (MB/s)	711.60	703.00	708.40	719.20	**710.55**
Redis - Test: GET - Parallel Connections: 50 (Reqs/sec)	175,777	176,882	177,973	181,704	**178,084**
Redis - Test: SET - Parallel Connections: 50 (Reqs/sec)	140,218	137,737	141,552	140,361	**139,967**
Redis - Test: LPOP - Parallel Connections: 50 (Reqs/sec)	209,774	209,634	212,602	210,840	**210,712**
Redis - Test: SADD - Parallel Connections: 50 (Reqs/sec)	137,785	136,588	140,729	137,868	**138,242**
Redis - Test: LPUSH - Parallel Connections: 50 (Reqs/sec)	128,942	129,173	130,643	131,112	**129,967**
nginx - Connections: 20 (Reqs/sec)	502	497	500	502	**500**
nginx - Connections: 100 (Reqs/sec)	515	514	511	525	**516**

Tetragon - Test Results by Node	kube01	kube02	kube03	kube04	Average
Unpacking The Linux Kernel - linux-5.19.tar.xz (sec)	15.98	16.19	13.94	14.12	**15.06**
Zstd Compression - Compression Level: 3 - Compression Speed (MB/s)	179.00	176.30	177.80	180.00	**178.28**
Zstd Compression - Compression Level: 3 - Decompression Speed (MB/s)	728.20	715.50	725.20	739.30	**727.05**
Redis - Test: GET - Parallel Connections: 50 (Reqs/sec)	1,335,899	1,313,252	1,363,654	1,358,134	**1,342,735**
Redis - Test: SET - Parallel Connections: 50 (Reqs/sec)	927,936	902,205	928,540	948,738	**926,855**
Redis - Test: LPOP - Parallel Connections: 50 (Reqs/sec)	1,352,879	1,312,276	1,346,741	1,353,217	**1,341,278**
Redis - Test: SADD - Parallel Connections: 50 (Reqs/sec)	1,127,551	1,109,052	1,130,203	1,148,415	**1,128,805**
Redis - Test: LPUSH - Parallel Connections: 50 (Reqs/sec)	713,028	723,020	714,703	726,849	**719,400**
nginx - Connections: 20 (Reqs/sec)	6,189	6,227	6,156	6,426	**6,250**
nginx - Connections: 100 (Reqs/sec)	6,141	6,303	6,098	6,289	**6,208**

Kata Containers - Test Results by Node	kube01	kube02	kube03	kube04	Average
Unpacking The Linux Kernel - linux-5.19.tar.xz (sec)	66.99	99.08	78.58	91.52	**84.04**
Unpacking The Linux Kernel - linux-5.19.tar.xz - virtio-fs (sec)	71.775	81.059	79.302	62.676	**73.70**
Zstd Compression - Compression Level: 3 - Compression Speed (MB/s)	268.80	202.70	212.50	210.40	**223.60**
Zstd Compression - Compression Level: 3 - Decompression Speed (MB/s)	718.10	704.70	717.00	725.70	**716.38**
Redis - Test: GET - Parallel Connections: 50 (Reqs/sec)	1,180,419	1,212,683	1,226,434	1,243,539	**1,215,769**
Redis - Test: SET - Parallel Connections: 50 (Reqs/sec)	869,500	883,480	895,624	904,463	**888,267**
Redis - Test: LPOP - Parallel Connections: 50 (Reqs/sec)	1,223,896	1,227,998	1,251,235	1,264,323	**1,241,863**
Redis - Test: SADD - Parallel Connections: 50 (Reqs/sec)	988,015	1,001,220	1,018,159	1,028,757	**1,009,038**
Redis - Test: LPUSH - Parallel Connections: 50 (Reqs/sec)	732,325	771,398	767,261	792,585	**765,892**
nginx - Connections: 20 (Reqs/sec)	2,956	1,815	1,858	1,860	**2,122**
nginx - Connections: 100 (Reqs/sec)	2,905	1,735	1,830	1,865	**2,084**

Test Result Averages Grouped by Sandbox	Baseline	gVisor	Tetragon	Kata
Unpacking The Linux Kernel - linux-5.19.tar.xz (sec)	14.43	36.49	15.06	84.04
Zstd Compression - Compression Level: 3 - Compression Speed (MB/s)	176.60	162.23	178.28	223.60
Zstd Compression - Compression Level: 3 - Decompression Speed (MB/s)	725.35	710.55	727.05	716.38
Redis - Test: GET - Parallel Connections: 50 (Reqs/sec)	1,352,507	178,084	1,342,735	1,215,769
Redis - Test: SET - Parallel Connections: 50 (Reqs/sec)	912,846	139,967	926,855	888,267
Redis - Test: LPOP - Parallel Connections: 50 (Reqs/sec)	1,324,555	210,712	1,341,278	1,241,863
Redis - Test: SADD - Parallel Connections: 50 (Reqs/sec)	1,129,425	138,242	1,128,805	1,009,038
Redis - Test: LPUSH - Parallel Connections: 50 (Reqs/sec)	722,285	129,967	719,400	765,892
nginx - Connections: 20 (Reqs/sec)	6,245	500	6,250	2,122
nginx - Connections: 100 (Reqs/sec)	6,270	516	6,208	2,084