健康資訊交換系統中之容器安全
The Container Security in Healthcare Data Exchange System

Chih-Hsuan Yang[1], Chun-I Fan[2]
Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan
zxc25077667@protonmail.com[1]
cifan@mail.cse.nsysu.edu.tw[2]

Abstract

This research proposes a mechanism, forcing a system to call a specific policy in a container, which is deployed in runtime. This policy is designed for the FHIR healthcare data exchanging in a standard's container, which could guarantee the FHIR server to have only supported behaviors and to take almost zero overhead. Recently, many companies use containers to run their microservices since containers could make more efficient use of their hardware resources as well as the newest healthcare data exchange standard FHIR (Fast Healthcare Interoperability Resources) has been implemented in a container by IBM, Microsoft, and Firebase. The deployment of FHIR in a container is a trend in the digital world. Containers are isolated processes instead of sandboxes. Therefore, this proposed scheme could reduce the risk of attacks in healthcare data exchange system without performance overhead.

***Keywords*—Container, Linux Kernel, Healthcare Data**

1. Introduction

1.1 Container and Linux Kernel

The container techniche is a secondary product of the operating system in the past 20 years. The FreeBSD develops 'Jails' in 1999, and the Solaris develops 'Zones' in 2004. Linux also took this idea into the Linux kernel, which is named cgroups (2007), the namespace (2002), the capabilities (2003), and seccomp (2005). However, why the Linux breaks this technology into many parts? This is because they had discussed: "Why Should a System Administrator Upgrade?" in 2001. The Linux kernel almost entered the development path of "upgrade for demand" like Microsoft Windows, and deviated from the original path of "providing a mechanism but not a strategy" of the original Linux kernel.

While Linux were spreading in various server or distributed system, the Linux community got more pull requests to solved the scalability and virtualization issues [8]. However, they avoided confusion caused by multiple meanings of the term 'container' in the Linux kernel context. In kernel version 2.6.24 (2007) [1], control groups functionality was merged into the mainline, which is designed for an administrator (or administrative daemon) to organize processes into hierarchies of containers; each hierarchy is managed by a subsystem. Moreover, the cgroups was rewrote into cgroups-v2 in Linux kernel 4.5 (2015).

Containers offers "build once, run anywhere." Docker does this by bundling applications with all their dependencies into one package and isolating applications from the rest of the machine on which they're running. Therefore, this research is based on docker container to propose a scheme of healthcare data exchange system's security.

1.2 Fast Healthcare Interoperability Resources (FHIR)

FHIR is a standard for healthcare data exchange. The FHIR standard will be used in Taiwan in the near future. FHIR will be used to provide PHR (Personal Healthcare Records) in Taiwan. Therefore, we choose the most popular standard "FHIR" for the target of the healthcare data exchange system.

This paper will take IBM FHIR server as an example. There are many applications using IBM's FHIR server as the base component of the EHR (Electronic Health Records) system to communicate with the other various databases. Take it for example that the NextCloud's EHR service, Taipei Veterans General Hospital, and AWS Cloud are using the FHIR server in a container for subroutine service. Many people caring about their privacy issues distrust the FAAMG (Facebook, Amazon, Apple, Microsoft, Google), so they are using NextCloud to keep their privacy on their own. Therefore, they are eager to have a secure EHR system for their PHR [2].

The benefits of providing IBM FHIR container security in our study are providing secure protection testing, methods, and performance evaluation for FHIR services provided by a well-known international company (IBM). This research will provide an important reference for commercial projects for the health information exchange system practiced by medical institutions in Taiwan.

*本研究接受國科會編號：110-2813-C-110-046-E 研究計畫經費補助

[1] https://lwn.net/Articles/256389/
[2] Richard Stallman talks about IoT

2. Related Works

2.1 Collecting System Calls

There are several pieces of research to detect intrusions or unexpected behaviors by collecting the system calls methods in runtime. Abed, Clancy, and Levy [1] proposed a real-time host-based intrusion detection system in a container, which is based on system call monitoring. They use the 'strace' command to collect a behavior log to a system-call parser. Then use the BoSC (Bag of System Calls) [16] to classify is it a normal behavior in the database.

The BoSC technique is a frequency-based detection tip. Kang, Fuller, and Honavar [16] defined those distinct system calls in $\{c_1, c_2, \ldots, c_n\}$, For all system call s_i had been called in c_i times. And they use Naïve Bayes classification to deduce if it is unexpected behavior. Then the Abed, Clancy, and Levy give the false positive rate of around 2% in $O(S + n_k)$ epochs to the MySQL database [1]. However, the BoSC is running in user space, even though it is a background service running on the same host kernel. It might have heavy constant time costs of copying data from user to kernel and kernel to user by the 'copy_to_user()' and 'copy_from_user()' calls.

Azab, Mokhtar, Abed, *et al.* [6] and Azab, Mokhtar, Abed, *et al.* [7] showed the survival rate in Abed, Clancy, and Levy [1] model for some zero-day vulnerabilities in different types and numbers of machines. Azab, Mokhtar, Abed, *et al.* [6] and Azab, Mokhtar, Abed, *et al.* [7] concluded that an IDS could detect and avoid mobile continually-growing attacks efficiently by the 'escaping' model with collecting system calls.

2.2 Fine-Grained Permission Control

The file system access control list (ACL) was defined in POSIX, which shares a naive and robust permission model [5], [12]. But after 20 years of evolution, in the practical consideration of the Linux operating system design, it can be divided into two permission control mechanisms: (i) POSIX ACL and (ii) seccomp. Traditional permission control is mostly controlled by ACL or similar. Many Linux secure modules (LSM) also use ACLs for file access control[21]. For example, SELinux and AppArmor use such permission settings.

Han, Lee, Lee, *et al.* [14] had proposed an architecture to enforce the access control of the image's layers. Because the docker engine does not guarantee the layers could not be modified by the host environment. Therefore, if we give a container privileged permission, it could modify the layers of images. The research [14] is using the LSM's policy table to enforce the access control of the file system in the kernel.

However, the policy engine has four types of policy conflicts: (I) Parent-Child Conflict, (II) Global-Local Conflict, (III) Lack of Authority, and (IV) Environment does not meet the expectation. The initial security namespace Φ is $\emptyset$. (I, II) will route the policy to $\Phi = \Sigma(\Phi \cap P_i), i \in \mathbb{N}, i < n$. And the (III, IV) is conflicted by the capabilities of that process. Sun, Safford, Zohar, *et al.* [22] give the capabilities a higher hierarchy than policy in the policy engine. Therefore all of these conflicts will follow the capability first.

2.2.1 Capabilities

Linux provides a more detailed permission control method on the file system, which is called capability and was proposed by Karger and Herbert. We can give archives some given capabilities without giving whole root permissions when it executes specific system calls. Otherwise, it must be a privileged process that can bypass all permission checks in Linux kernel.

2.3 Recently Exploited Vulnerabilities

In this subsection, we will mention and review some 'High' or 'Critical' vulnerabilities about kernel and containers in CVSS (Common Vulnerability Scoring System). Because container is not a real virtual machine, it is an isolated process.

We ignore the CVE-2020-29389 series (CVE 306). Because those CVEs are not container or kernel's vulnerabilities, those CVEs are issue of image default passwords. Despite those CVEs got 10.0 score, those are trivil vulnerabilities.

2.3.1 Five Stages of Malware

We had been inspired by the quark engine, which is an open-source malware scoring system for Android APK files. Quark engine had been developed from the Taiwan Criminal Law's five stages: (i) Determination, (ii) Conspiracy, (iii) Preparation, (iv) Start, (v) Practice.

We also can use these five steps and category to give the malware stage to exploit the vulnerabilities. (i) Base image landing, (ii) Derived image landing, (iii) User landing, (iv) Kernel landing, (v) Escaping. The escaping category is the worst case of container security, because we want a container be a container, it must has zero leakage of capsulation.

a) Base image landing

This is the most fundamentally basic assumption or guarantee of container security. Ahmed and Pierre [2] proposed the BoSC technique must be $S = \{\emptyset\}$ in this step. By definition, for all container c is an image I in execution, that is $c = E(I) = \{\delta, \lambda\}$. E is a function to execute and give container c a description δ and a lifetime status λ.

b) Derived image landing

It is called derive image landing if some malicious items are inserted into the final layer, while developers are inserting the application(s) and some dependencies into image layers, It could be performed by a malicious base, dependencies, libraries, or binaries are inserted into the filesystem. It is often in third-party unknown source image which is integrated and republished by some crackers.

c) User landing

In this step, the cracker could control the normal service to do the unexpected behaviors as normal hacking scenarios. They can drop databases [13] , practice the local file inclusion [15], [23], etc. Take an online judge in a container as an example: People could write some program, compile it, and execute it on that machine. The cracker could write some malicious program or load some shellcode in those programs, and give the operating system to execute. This is the user landing step.

d) Kernel landing

It is the hacker could hack the kernel While the kernel copies data from the user and executes the user-provided malicious pattern or the user exploits the kernel vulnerabilities, and lets that code executed in kernel mode, that is kernel landing.

e) Escaping

This is the most critical step of these five steps because this is the final utility given by the container. Despite the kernel landing is almost control the whole machine, it is the last container insecure issue of breaking the containers. There are three types of escaping: (i) Cgroups, (ii) Namespaces, (iii) Capabilities.

(i) The cgroup escaping showed that Gao, Gu, Li, *et al.* [10] break the cgroups' limitation and affect the other container on the same host significantly, and gain some extra resources from the host. (II) Because of the limitation of lengh we skip it. The last one, (iii) capability escaping can be overridden the capability after the kernel landing and modify the 'task struct' of the process in the kernel.

2.3.2 Case Studies

This paper considered the Dirty CoW Alam, Zaman, Farah, *et al.* [3] and Lan and Wang [19] exploits, CVE-2016-8655 series, and some runC exploits.

First, the Dirty CoW is a kernel vulnerability based on race condition of mmaped memory spaces. To prevent Dirty CoW in legacy infrastructures without kernel patch, the only one solution is giving the limitation of system calls. Second, there are found many vulnerabilities in net/packet/af_packet.c with unsuited capability configurations that would help hackers gain root privileges. The last studied we had studied the runC exploits, such as CVE-2019-5736, CVE-2021-30465 etc. Those vulnerabilities were attacking the entry point that helps containers start-up, which would modify container configurations to lead hacker landing and escaping to the host machine.

2.4 Virtual Environment Performance Benchmark

There is a trend of applications are developed or deployed into microservice in a virtual environment since 2008. And the performance benchmark of applications in the virtual environment becomes more and more critical.

Therefore, there are many pieces of research shows how to evaluate the performance when using containers or the other virtual infrastructures[4], [9], [18], [24]. They are comparing the throughput, latency, and QoS for memory IO, or cryptography algorithms calculating costs.

Young, Zhu, Caraza-Harter, *et al.* [24] showed the gVisor costs: 2.2× system call overhead, 2.5× memory allocation latency, and 216× **slower** than raw system on complex file opening. And Kozhirbayev and Sinnott [18] showed that I/O times have more disadvantages of latency and throughput, which is compared to container and native machines.

3. The Proposed Scheme

It is the programmer's responsibility to write complete unit and integration tests. We extend the definition Test-Driven Development (TDD), which is not only red, green, and refactor, but also "Test Do what's Designed".

3.1 Workflow

In short, our proposal is generating a perfectly fittable mask layer which is coupled with the healthcare data exchange system in build time.

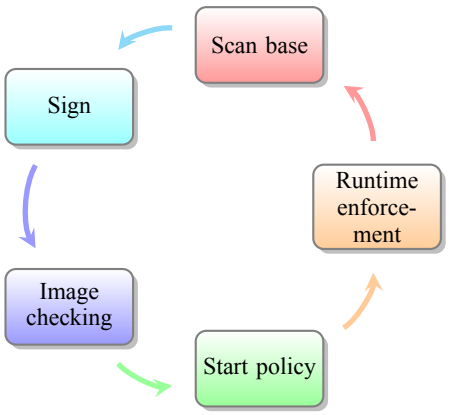

Fig. 1: **Contiguous Integration and Contiguous Deployment**

We proposed a CI/CD workflow to guarantee the runtime enforcement of policies in Fig. 1. Each block of the workflow will be described in the following subsubsection.

Because of the CI/CD workflow, we can rolling update all the features or patch vulnerabilities such that the software would be released secure eventually. Linus Torvalds said[3] "The only real solution to security is to admit that bugs happen, and then mitigate them by having multiple layers." And our layer is enforced in kernel space; therefore, there are not existing other attacks that can be inflected in the user program except for the kernel exploit.

3.1.1 Scan Base Image

We scan all the layers which construct the image of the container recursively. All containers are images in execution, that is, we can treat the container as

[3] https://www.youtube.com/watch?v=5CIL54-KKz0

an image in runtime. Therefore, the layers of image construction have to be trusted.

For a general image I_i which has been constructed in n layers L_i, $\forall i \leq n$, $n \in \mathbb{N}$, we can use the spotbugs[4] or the other bug-scanning tools to ensure that the software is a bugless program. The bugless program p_i is in the layer L_i which constructs the I_i

3.1.2 Building and Signing

We will execute the developer's unit tests and the integration test in the build time. We catch all the system calls s_i by the BoSC [16] method, and generate the $S = \{s_1, s_2, \ldots s_i \ldots s_n\}$ set from the program's n system calls, where $S \subseteq \mathbb{S}$, and $\mathbb{S}$ is the set of all the system calls that the kernel supported. We wrote a driver to parse the S into a whitelist filter of seccomp's policy P.

Through the workflow above, the L_i's security is almost surely enough. Then we sign our certificate C and the policy R to the image I_i, which is constructed by those trusted layers L_i into $\hat{I}_i$. That is $\hat{I}_i = C(P \oplus \Sigma_{\forall i} L_i)$.

3.1.3 Check Image and Policy

When we deploy the $\hat{I}_i$ into an active machine, we have to check the C of $\hat{I}_i$ is valid for signer's trusted verification server.

The verification server can check the certificate C's integrity and encrypt those checking results by the server's private key P_{VK} to the active machine. The active machine will also check the certificate C' from the verification server bidirectionally.

And we register our policy P into the active machine's kernel to limit the $\hat{I}_i$ launched by the user in runtime, that is the container.

3.1.4 Enforce the Policy

The kernel of the active machine can help us to guarantee that the policy P is enforced in kernel space. Since the container is launched by the user, the policy P has been invoked in each system calls of the container. Because the policy P is a whitelist, all of the other system calls which do not belong to the signed container's application would send a permission denied signal from the kernel.

3.2 Rolling Updates

The rolling update is a trend of software engineering products, which is also named agile software development. Eric S. Raymond formulated the Linus's law in *The Cathedral and the Bazaar* [20]. We give enough eyeballs and layers, and all bugs or vulnerabilities are shallow in our healthcare data exchange system. Therefore, the container can be secure eventually.

4. Analysis and Benchmark

The proposed scheme profiles an image of containers via cgroup and namespace and uses the seccomp mechanism to force the policy in the kernel. We will analyze how the proposed scheme protects our system when hackers land into the container, and we profile the concurrent costs of this mechanism.

4.1 Analysis

Our defense level is at the kernel level, but the virtual machine's defense level is at the instruction level. Because we do not impose any restrictions on the CPU instruction set, nor isolate the host operating system. Although the defense level at the instruction set seems to be more efficient, the virtual machine's protection consumes more time. We will show that in Fig 3.

In the health and medical information exchange system, the health and medical information we protect is specialized and fixed. For example, we do not have any attack on parsing some format string, which is an exploit of bypassing the ASLR[5].

So the proposed scheme can remove some redundant system calls that could limit the possibility of hacker exploitation. Therefore, we can protect the user's data from being hacked in the information exchange system.

4.1.1 Attacking Surface

We discuessed the five stages of malware in 2.3.1. We analyzed three possible attacking scenarios for hackers in this subsubsection.

4.1.2 Zero-day or One-day Vulnerability

Assuming that the hacker does not have system administration privileges, but exploits the vulnerability of the health information exchange system (IBM/FHIR server) to conduct malicious attacks. We can also use the same behavioral filter to filter the attack. For example, the log4j attack (CVE-2021-44228), which has been a published vulnerability while we researching this container security issue. Before our research, this vulnerability existed in IBM/FHIR container server. When a user turns the "export to parquet" feature on, it would bring in much of Apache Spark which leads to enable the vulnerable log4j.

However, unfortunately, we have to admit that the defenses we propose cannot withstand this log4j attack. The IBM/FHIR server itself can enable such a mechanism, so actions using log4j are invoked at build time. We would admit these behaviors as normal behavior at the system-call filter level.

4.1.3 Time Consuming

Kozhirbayev and Sinnott [18] showed that there is basically no statistical difference between container and host environment. This is completely in line with our perception of a container, which is said that containers are isolated processes.

According to our experiments, to do a statict that the integration tests and unit tests were executed on IBM/FHIR server 4.9.0, and the system calls, and system events we collected are shown in Fig. 2.

Fig. 2 illustrates the FHIR server's all system calls in BoSC [16] and the number of times that had been called. Among them, we can find that the most used

[4] https://spotbugs.github.io/

[5] https://lwn.net/Articles/569635/

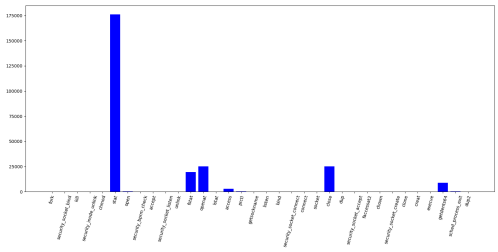

Fig. 2: **All the system calls from the FHIR Server**

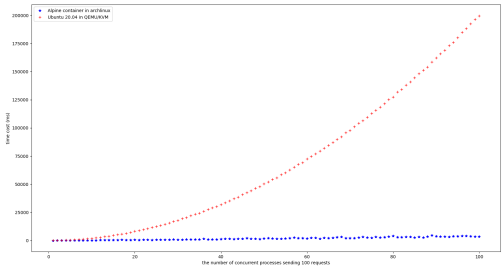

Fig. 3: **Concurrent processes transporting time**

is the 'stat' system call. To benchmark, it is found that the discussion of container performance testing is less focused on the requirement of parallel multiplexing [4], [9], [18], [24]. And it is a more important issue for the server's high multiplexing performance service client. Which result is same as our tesing results.

4.1.4 Latency

Fig. 3 is the concurrent processes transporting time difference in a container and a virtual machine. Young, Zhu, Caraza-Harter, *et al.* [24] showed that the latency of opening and closing files is no significant difference between native and runc. But there was 12 times faster than the gVisor with internal access. Although our IBM/FHIR server cannot be executed in gVisor, it is the same in native and runc with no significant difference.

Felter, Ferreira, Rajamony, *et al.* [9] showed the relation between the throughput and the concurrency, where both have the transaction's upper bound cost in MySQL. The overhead of KVM is much higher, above 40% in all measured cases. We think that there is a driver buffering bottleneck in the hypervisor of KVM in ring 0.

Hence, we compare the time lag between Ubuntu 20.04 in QEMU/KVM in Archlinux and native Alpine container in Archlinux on concurrent requests. A phenomenon we found is that the latency curve of a virtual machine seems to be different in complexity from that of a native container.

5. Conclusion

We can see that the comparison results in virtual machine and container are significantly indifferent order of time-consuming. There is no existence of the gVisor's result because the gVisor was not able to launch the IBM/FHIR server system, which is the target of our research. We also expect that the gVisor might run faster significantly than the virtual machine; however, our target cannot be launched successfully in gVisor's sandbox. We thought that there might have been some race condition bugs via JWE (JAVA Web Engine) in gVisor, such that the gVisor did not do well in supporting all system calls.

And the time complexity of the virtual machine is significantly different from the container. We propose a hypothesis of the time complexity of the virtual machine, because there are more page fault events and the throughput limitation of virtual machine device driver [9], [11].

The proposed scheme is nearly zero-overhead protection in the kernel, and the policy is auto-generated and conformed in the build time. It could help developers to deploy into a secure environment with no pain. This paper also benchmarked the protection via virtual machine and our proposed scheme in the container. It can be concluded that it is a good solution if we want low latency and security.

References

[1] A. S. Abed, C. Clancy, and D. S. Levy, "Intrusion detection system for applications using linux containers," in *Proceedings of the 11th International Workshop on Security and Trust Management - Volume 9331*, ser. STM 2015, Vienna, Austria: Springer-Verlag, 2015, pp. 123–135, ISBN: 9783319248578. DOI: 10.1007/978-3-319-24858-5_8. [Online]. Available: https://doi.org/10.1007/978-3-319-24858-5_8.

[2] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018, pp. 1–8. DOI: 10.1109/EDGE.2018.00008.

[3] D. Alam, M. Zaman, T. Farah, R. Rahman, and M. S. Hosain, "Study of the dirty copy on write, a linux kernel memory allocation vulnerability," in *2017 International Conference on Consumer Electronics and Devices (ICCED)*, 2017, pp. 40–45. DOI: 10.1109/ICCED.2017.8019988.

[4] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[5] S. Arnautov, B. Trach, F. Gregor, *et al.*, "Scone: Secure linux containers with intel sgx," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA,

USA: USENIX Association, 2016, pp. 689–703, ISBN: 9781931971331.

[6] M. Azab, B. Mokhtar, A. S. Abed, and M. Eltoweissy, "Toward smart moving target defense for linux container resiliency," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, 2016, pp. 619–622. DOI: 10.1109/LCN.2016.106.

[7] M. Azab, B. M. Mokhtar, A. S. Abed, and M. Eltoweissy, "Smart moving target defense for linux container resiliency," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, 2016, pp. 122–130. DOI: 10.1109/CIC.2016.028.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, *et al.*, "An analysis of linux scalability to many cores," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC: USENIX Association, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores.

[9] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[10] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 1073–1086, ISBN: 9781450367479. DOI: 10.1145/3319535.3354227. [Online]. Available: https://doi.org/10.1145/3319535.3354227.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96, San Jose, California: USENIX Association, 1996, p. 1.

[12] A. Grünbacher, "Posix access control lists on linux," in *USENIX Annual Technical Conference, FREENIX Track*, 2003.

[13] W. G. Halfond, J. Viegas, A. Orso, *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, IEEE, vol. 1, 2006, pp. 13–15.

[14] S.-H. Han, H.-K. Lee, S.-T. Lee, S.-J. Kim, and W.-J. Jang, "Container image access control architecture to protect applications," *IEEE Access*, vol. 8, pp. 162 012–162 021, 2020. DOI: 10.1109/ACCESS.2020.3021044.

[15] M. M. Hassan, T. Bhuyian, M. K. Sohel, M. H. Sharif, and S. Biswas, "Saisan: An automated local file inclusion vulnerability detection model," *International Journal of Engineering & Technology*, vol. 7, no. 2-3, p. 4, 2018.

[16] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, 2005, pp. 118–125. DOI: 10.1109/IAW.2005.1495942.

[17] P. A. Karger and A. J. Herbert, "An augmented capability architecture to support lattice security and traceability of access," in *1984 IEEE Symposium on Security and Privacy*, 1984, pp. 2–2. DOI: 10.1109/SP.1984.10001.

[18] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.08.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X16303041.

[19] H. Lan and X. Wang, "Research and design of concurrent web server on linux system," in *2012 International Conference on Computer Science and Service System*, 2012, pp. 734–737. DOI: 10.1109/CSSS.2012.188.

[20] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly Media, Inc., 2002, ISBN: 9780596001087.

[21] S. D. Smalley, C. Vance, and W. Salamon, "Implementing selinux as a linux security module," 2003.

[22] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making linux security frameworks available to containers," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 1423–1439, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[23] M. E. Whitman and H. J. Mattord, *Principles of information security*. Cengage learning, 2011.

[24] 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," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/young.