Taking a Peek: An Evaluation of Anomaly Detection Using System calls for Containers

Gabriel R. Castanhel, Tiago Heinrich, Fabrício Ceschin, Carlos Maziero
Computer Science Graduate Program
Federal University of Paraná State
Curitiba, Brazil, 81530–015
E-mail: {grc15,theinrich,fjoceschin,maziero}@inf.ufpr.br

***Abstract*—The growth in the use of virtualization in the last ten years has contributed to the improvement of this technology. The practice of implementing and managing this type of isolated environment raises doubts about the security of such systems. Considering the host's proximity to a container, approaches that use anomaly detection systems attempt to monitor and detect unexpected behavior. Our work aims to use system calls to identify threats within a container environment, using machine learning based strategies to distinguish between expected and unexpected behaviors (possible threats).**

***Index Terms*—Intrusion Detection, Computer Security, Containers**

I. Introduction

Operating system-level virtualization, also known as containerization, is a kind of virtualization in which multiple isolated user spaces are provided by a single kernel. Each user space, named a *container*, runs an application with its dependencies and resources, isolated from the other applications running in the same physical host. This allows multiple containers with different structures and applications to run side by side, sharing the same operating system kernel.

Nowadays, the popularity of this type of virtualization raises security concerns, due to the proximity between the applications running in a container and the host, because the containerization layer is much thinner than a full virtualization stack. Several attacks have been found and exploited in recent years, such as remote code execution, privilege escalation, tampering, among others [1].

Intrusion detection is a way to identify and prevent malicious activity in a system. Resources used for such activities are Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) [2]. Specifically, an IDS performs intrusion identification using different techniques that can be signature-based, which performs the comparison of signatures with a known base of threats, or anomaly-based, in which the normal behavior of the system is previously known and deviations from it are classified as threats [3].

Looking only at intrusion detection targeting containers, some differences could be pointed out. Containers are more lightweight than virtual machines and thus easier and faster to manage. In addition, the semantic gap between the virtual environment and the host is very small in containers, in comparison to traditional virtual machines [4]. This allows an external observer to gather very detailed information about the applications running inside a container.

An attacker could exploit different techniques to compromise, gain access, or even run code in a container environment. Such attacks can be carried out from a compromised image, running applications with unnecessary permissions (e.g. as root), exploiting a vulnerability in the application running the container, or an application with a misconfigured environment [1].

This work studies and compares the effectiveness of a set of methods in identifying malicious activities in a container. An observer external to the container gathers detailed information about its execution, in the form of a sequence of system call issued by the application. This data then is used to train classifiers to build a "normal behavior" model of each container, and consequently allowing to identify anomalies.

In summary, this paper makes the following contributions:

- A discussion about the use of intrusion detection using system calls in containers, and how it was implemented;
- A methodology for the identification of attacks inside a container using machine learning, achieving high accuracy and precision rates for both strategies presented;
- We explore the impact of window size and system call filtering for an intrusion detection system in a container;
- A dataset that enables the evaluation of the use of system calls to detect threats inside a container.

The reminder of this paper is structured as follows: Section II presents the related work; Section III presents the background; Section IV discusses the paper proposal; Section V presents the evaluation, and Section VI concludes the paper.

II. Related Work

From an attacker point of view, there are several possibilities of attack to a virtualization system, such as exploiting virtualization to extract private information from users, launching Distributed Denial of Service (DDoS) attacks or escalating the intrusion to multiple VM instances. The literature highlights studies aimed at monitoring virtualization at different levels, in order to identify such malicious actions [5, 6].

Intrusion detection in virtualization environments brings some challenges. If the IDS is put inside the guest, it has a rich view of the monitored application, but is also exposed to attackers targeting it. On the other hand, if the IDS resides

978-1-6654-2744-9/21/$31.00

outside the monitored guest, is is protected against such attacks, but its view of the guest application execution is much poorer, due to the semantic gap [4].

However, when using container-based virtualization, the semantic gap is greatly reduced, because all containers share the same kernel. Thus, an IDS running directly on the host OS is able to fully observe the behavior of processes running in a container [4].

One of the first studies that explored the field of system call based intrusion detection was presented by [7]. It proposed a method that was inspired by the mechanisms and algorithms used by natural immune systems. The study observed that short sequences of system calls appeared to maintain a remarkable consistency among the many possible sets of system calls of the possible execution paths of a program. This inspired the use of short sequences of system calls to define the normal system behavior and presented a simple and efficient way to detect anomalies, with possible applications to real time scenarios.

For approaches based on container virtualization, it is possible to highlight [8], which presents a model for identifying anomalies in applications running within Docker. The strategy is to use n-grams to identify the probability of an event occurring. The experiment achieves an accuracy of up to 97% for the UNM *dataset* [9]. However, such dataset is very old and is not representative of current applications and virtual environments.

Still in the context of containers, [10] presents a preliminary feasibility analysis for anomaly-based intrusion detection, focusing on Docker and LXC technologies. The article proposes an analysis and capture architecture for *system calls*, the application of the Sequence Time-Delay Embedding (STIDE) and Bag of System Calls (BoSC) algorithms, and studies the training process in different cases. The study trained both algorithms with window sizes ranging from 3 to 6 *system calls*, and calculated the slope of the growth curve, which means the rate of new windows added to each classifier's normal behavior base after a period of time. The results highlight a stable learning state for STIDE with windows of size 3 and 4 and from 3 to 6 for BoSC, which means that the best configuration obtained was using windows of size 3 and 4. The database used for validation and experimentation was developed for the study but is not publicly available.

The study of [11], focuses on the detection of intrusion by anomalies in container environments, applying a technique that combines BoSC with the technique of STIDE. The analysis of the container behavior is made after its closure, with the aid of a table containing all the distinct *system calls* with the respective total number of occurrences. The method reads the flow of *system calls* by *epochs*, and slides a window of size 10 through each *epoch* producing a BoSC for each window, which is used to detect anomalies, which in turn is declared if the number of disparities in the normal behavior base exceeds a defined *threshold*. The classifier achieved a detection rate of 100% and a false positive rate of 0.58% for *epoch* of size 5,000 and *threshold* of detection of 10% the size of *epoch*. The experiment database is not available.

III. Background

This Section presents the main concepts for understanding the work, discussing points such as anomaly detection, *system calls* for IDS and containerized virtualization.

A. System calls

System calls are mechanism available for the interaction between an application (or process) and the operating system kernel. When a program needs to perform privileged operations, requests are made via *system calls*, usually because user-level processes are not allowed to perform such operations. This occurs for a variety of tasks such as interacting with hardware resources, memory management, input/output processing, network operations, file operations, and others [12].

Some of such actions should not be available to all user-level processes, like shutdown, access to other processes' memory areas, changing user IDs, and bypassing access control policies. The OS kernel thus implements security policies that determine which system calls may be called by which user-level processes.

Due to the position in which *system calls* are implemented, their observation gives rich information about the activities performed by user-level processes. The set of *system calls* found in a system is directly related to the Operating System (OS) and the architecture used. Tools like *strace* and *ftrace* [13, 14], allows to show the sequence of all *system calls* used by a command or a running process.

Regardless of the approaches used to execute malicious code on a system, they usually exploit the *system calls* interface for performing malicious operations [15]. It is only through this interface that a compromised application interacts with the system services and resources. The monitoring of *system calls* is a widely used technique that makes use of this feature in common to detect suspicious behavior of an application that may have been compromised, so that a countermeasure is possible to minimize the problem [16].

B. Containers

Although the concept of container has only been popularized in the last few years, this is a technique introduced as early as the 1980s, to perform software "isolation" using the *chroot* tool on Linux systems. Currently it is known as an application virtualization technique for the most diverse purposes, generally associated with popular tools like docker. The container is a virtual environment that runs in a single OS and allows the loading and execution of a specific application and its dependencies contained in a virtualization of an operating system [17]. In this way, the container gathers the necessary components for the execution of the application, which includes code, libraries, and environment variables, while the host OS manages access to the hardware resources as memory and processor, and all necessary kernel operations.

Containers promote isolation between host and guests, but this isolation is not as strong as in full virtualization, where a

hypervisor is responsible for managing and running the guest systems. The hypervisor is the intermediate layer between the virtual machines and the host, responsible for the instruction conversion and resource allocation [18].

The strong isolation provided by the hypervisor comes with a cost, in terms of processing, memory usage, and ease of resource sharing. Such restrictions are much lighter when using containers [19]. As containers share same the kernel, a single OS instance is capable of supporting multiple isolated containers. In addition, the virtualized OS in the container is lighter than a virtual machine, having only the resources necessary for the execution of the application [20].

IV. Proposal

Monitoring vulnerabilities at the application level is a processor-consuming task, and it is impracticable to carry out a monitoring from inside the container due to its restrictions and limited resources, as well as the risk of having the IDS exposed to the attacker [21]. Therefore, an option would be to monitor the guest processes from the host system, using as data the system calls they generated.

In this context, [7] introduces a window-based method of system calls monitoring, which is simple and effective for real-time detection. Approaches using frequency tables [21, 22], Machine Learning (ML) [23, 24] algorithms, and Markov chains [25] also showed good results regarding detection rates.

In order to compare different possible techniques for anomaly detection based on system calls, the proposed approach focuses on an application running in a container environment and being observed from the host. Our goal is to explore ML techniques to identify attacks, and understand how the intrusion detection systems will behave with this outside perspective. As we used a sliding window technique, we focus on evaluating how the size of the window will impact the results, and we make a thorough assessment of the correct size to be used.

Figure 1 presents three approaches to collect system call information from guest processes running inside a container. The first strategy is to run the IDS at the host level, outside the container (represented by `App1` in Fig. 1). This protects the IDS against a compromised container [26].

The second approach consists of using the trace application inside the same container the application is running (represented by `App2` in Fig 1), which may be harmful to the IDS. The third approach defines a privileged container for running the IDS (represented by `App3` and `App4` in Fig. 1). The IDS container should have access rights to the processes being monitored, to gather the system calls they issue. This approach is interesting, because it allows to define a complete IDS container, that can be easily deployed elsewhere.

This work used the first approach to collect data, using the `strace` tool to collect all the interactions between the guest process and the kernel. We collected enough information to build a base of normal behaviors, containing sequences of system calls that represent the normal functioning of the application, and also sequences that represent it under an attack or malicious behavior, to evaluate the proposal.

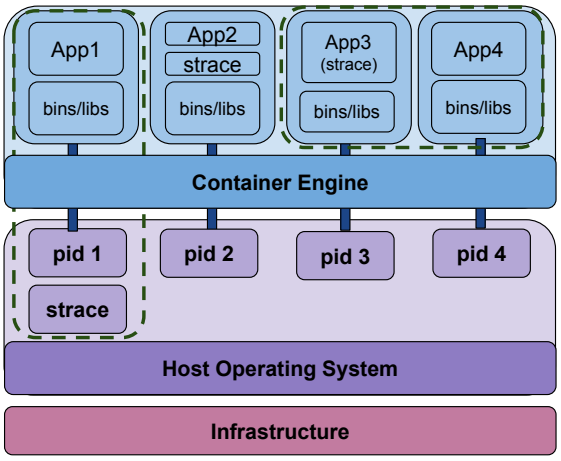


Fig. 1: Approaches to monitor a guest application.

V. Evaluation

We built a prototype of our proposal, to verify whether monitoring a guest application from outside the container is effective, and also to assess the algorithms used to detect anomalies in the stream of system calls issued by an application. A labelled dataset of normal behavior and attacks was also generated and is publicly available[1].

The experiments were run in a Linux host using Docker as the virtual environment. The application selected for the tests was `Wordpress`, a web application that runs over the Apache web server. This choice is due to its popularity among web applications, and due to the existence of a wide set of vulnerabilities. The extensive use of third-party *plugins* and personalized themes opens the door to security flaws involving Cross-site scripting (XSS), *SQL injection*, Remote Code Execution (RCE), among others.

The application configuration was adjusted to reduce the amount of threads and child processes forked by the Apache server, in order to have a cleaner and more consistent stream of system calls issued by the application. collected syscall information contains their names, parameters, and return values.

The dataset built contains fifty traces, half from normal behavior that consist of expected interactions with the application (five different interactions), and the other half anomalous behavior consisting of attacks (five different attacks focusing in XSS and RCE). They are: `WordPress` store XSS via the `wp-admin` [27], CSV import/export allowing unauthenticated PHP code execution, file manager allowing allows the upload and execution of PHP code [28], bypass extensions to upload PHP shells [29], and file uploading from unauthenticated users.

[1]The dataset and the code used in the experiments can be found in https://github.com/gabrielruschel/hids-docker

Two groups of tests were developed: in the first case all system calls were used, without applying any type of filtering. For the second case, system calls were classified into threat levels, and the least dangerous ones were discarded.

For the data processing, a sliding window approach was used, rebuilding the trace with a window of size n. In total, seven different window sizes were tested using four algorithms, these being *K-Nearest Neighbors* (KNN) with a $k = 3$, *Random Forest* (RF), *Multilayer Perceptron* (MLP), and *AdaBoost* (AB), all of them with the default scikit-learn parameters [30]. For the training phase, the dataset was split in 50/50 for training/testing, and ten executions were run for each classifier, changing the random seed used in the split phase to generate different sets, to avoid overfitting issues.

The efficiency of each algorithm was evaluated using four metrics: precision, recall, f1-score, and accuracy. *Precision* represents the ratio between correctly predicted detections and all detections that have occurred, where high values represent a low occurrence of false positives. *Recall*, on the other hand, represents the fraction of detections identified within all possible detections. *F1-score* combines the values of *precision* and *recall* into a single result, which indicates the overall quality of the model.

The sections V-A and V-B discuss the experiments carried out and the results obtained, and Section V-C presents an evaluation of the window size, considering the impact for intrusion detection.

A. Using all system calls

The first set of experiments consists of using all the system calls issued by the application. Each system calls is represented by a unique numeric identifier. Table I presents the results obtained with ten executions, with no filtering. In general the results seem adequate, with precision and accuracy above 90% for all window sizes and classifiers. The difference between the classifiers are quite small, with a f1-score and recall number not having a relevant difference between each algorithm.

Looking only at the window size in Table I, some points can be highlighted: (1) window sizes 3 and 5 present the most "unstable" values in relation to the other window sizes; (2) the difference between the results is small for windows sizes between 7 to 15, which indicates that they could present a good trade-off between detection time and classification performance; and (3) *Random Forest* and *AdaBoost* present the best overall results.

B. Using filtered system calls

The work [31] proposed a security classification for system calls, in which each syscall is labelled with a threat level, meaning how dangerous that system call can be to the system if misused. Proposed levels range from 1 to 4, where level 1 represents system calls that allow complete control of the system; level 2 represents system calls that can be used for Denial of Service (DoS) attacks; level 3 system calls can be used to subvert the responsible process; finally, level 4 system calls are classified as harmless. The calls classified as low threat by the article can be found in Table II, which represents part of the structure defined by [31].

TABLE I: Ten executions considering all calls (without filter).

Classifier	Metric	Window Size 3	5	7	9	11	13	15
KNN	*precision*	84.5	90.4	90.3	88.5	91.8	90.4	87.9
	recall	63.1	71.3	74.0	76.2	76.5	77.7	79.4
	f1-score	71.4	79.7	81.3	81.8	83.4	83.5	83.3
	accuracy	90.1	92.9	93.4	93.4	94.1	94.0	93.8
RF	*precision*	99.1	98.7	98.7	98.8	98.8	98.7	98.7
	recall	58.5	70.1	73.0	74.4	75.6	76.7	77.5
	f1-score	73.6	82.0	83.9	84.9	85.7	86.3	86.9
	accuracy	91.8	94.0	94.6	94.9	95.1	95.3	95.4
MLP	*precision*	91.3	89.1	90.7	92.2	91.6	90.0	90.5
	recall	54.3	64.5	67.0	68.2	69.0	69.9	70.5
	f1-score	68.1	74.8	77.1	78.4	78.7	78.6	79.2
	accuracy	90.1	91.5	92.2	92.7	92.7	92.6	92.8
AB	*precision*	99.1	98.7	98.7	98.8	98.8	98.7	98.7
	recall	58.5	70.1	73.0	74.4	75.6	76.6	77.5
	f1-score	73.6	82.0	83.9	84.9	85.7	86.3	86.8
	accuracy	91.8	94.0	94.6	94.8	95.1	95.3	95.4

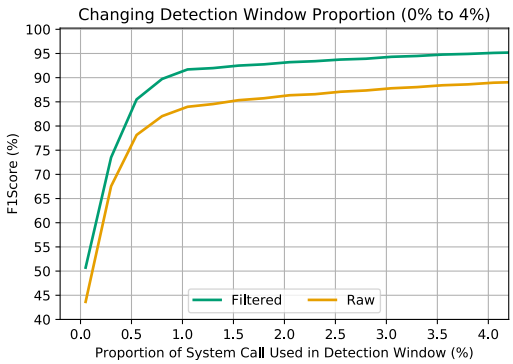
TABLE II: *System calls* classified as harmless. Obtained from [31].

4	I	oldstat, oldfstat, access, sync, pipe, ustat, oldstat, readlink, readdir, statfs, fstatfs, stat, getpmsg, lstat, fstat, oldname, bdflush, sysfs, getdents, fdatasync
	II	getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched_getscheduler, sched_getparam, sched_get_priority_min, sched_rr_get_interval, capget, getpid, getsid, getcwdm getresgid, getresuid
	III	get_kernel_syms, create_module, query_module
	IV	times, time, gettimeofday, getitimer
	V	sysinfo, uname
	VI	idle
	VII	break, ftime, mpx, stty, prof, ulimit, gtty, lock, profil

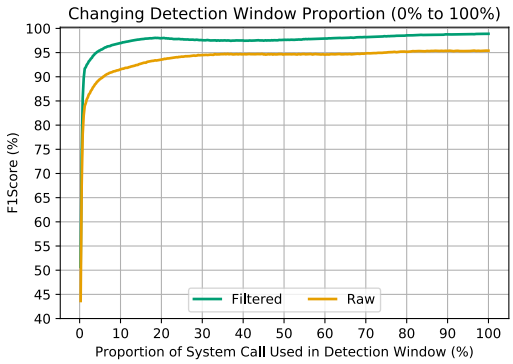
This "harmless" group consists of system calls that return some value or attribute on a specific file, as in the case of the *stat* syscall and its variations, or about system resources like *getpid*, *getuid*, *gettimeofday*, for example. This group of calls is not used for manipulating files, memory, nor executing commands or programs, and does not make changes to the system, so they can be classified as harmless in relation to system security.

Based on the system calls presented in Table II, they are filtered out from the system call stream and not taken into consideration. Despite discarding system calls of only one of the 4 threat levels, the volume of data being analyzed was divided by two compared to the previous experiment. This means that the training and the decision made by the classifiers will be faster, i.e., an attack will be detected faster.

Table III presents the results using the filtered system calls.



a Online scenario. Results using fewer system calls (representing an online/real scenario) presents worse results than using the full traces (offline scenario).



b Overall scenario. Results changing the proportion of system calls from 0% to 100% (full trace) in both scenarios (filtered and raw data).

Fig. 2: Comparing online and offline solutions. Results using the offline approach (almost the full trace) present much better results than the online approach (with few system calls).

The results show improvements in relation to the classification using all system calls presented in Table III. However, it is important to highlight that *f1-score* presented a growth, which demonstrates a better adequacy of the results concerning all the classifiers. The classifiers that presented the best performance were *Random Forest* and *AdaBoost*. Considering that *Random Forest* is computationally cheaper than *AdaBoost*, we consider that it is the best option for this classification problem. *AdaBoost* and *Random Forest* have similar results, although they have different strategies to train the models; the only similarity between them is the use of decision trees as base classifiers.

TABLE III: Ten executions with call filtering (with filter).

Classifier	Metric	Window Size 3	5	7	9	11	13	15
KNN	*precision*	88.5	96.4	95.9	95.5	96.4	95.8	97.0
	recall	71.4	82.0	85.1	86.1	86.8	87.5	88.0
	f1-score	78.6	88.6	90.2	90.5	91.3	91.5	92.3
	accuracy	89.7	94.4	95.1	95.2	95.6	95.7	96.1
RF	*precision*	99.1	99.1	99.2	99.3	99.2	99.2	99.2
	recall	68.1	81.9	85.2	86.1	87.0	87.8	88.5
	f1-score	80.7	89.7	91.7	92.2	92.7	93.1	93.5
	accuracy	91.4	95.0	95.9	96.2	96.4	96.6	96.8
MLP	*precision*	92.1	94.6	95.0	96.6	95.4	96.2	96.2
	recall	64.1	77.9	81.9	82.2	82.9	83.5	83.8
	f1-score	75.5	85.4	87.9	88.8	88.7	89.4	89.6
	accuracy	89.1	93.0	94.1	94.6	94.4	94.8	94.9
AB	*precision*	99.1	99.1	99.2	99.3	99.2	99.2	99.1
	recall	68.1	81.9	85.2	86.1	87.0	87.8	88.5
	f1-score	80.7	89.7	91.7	92.2	92.7	93.1	93.5
	accuracy	91.4	95.0	95.9	96.2	96.4	96.6	96.8

C. Window size impact

To verify the impact of the size of the sliding window on an intrusion detection system, a new set of experiments was performed. This test focuses on growing the window size in small portions in order to analyse the behavior of the intrusion detection system. The window change steps are relative to the size of the smallest trace in our dataset, in this case 594 (i.e., a trace with 594 system calls). The traces have an average size of $8,646$ system calls, with the biggest trace having a size of $27,171$ system calls.

These tests are directly related to online/offline approaches. An online strategy tends to use sliding windows for real-time attack detection, generally using small-sized observation windows to detect them as soon as possible. An offline approach tends to explore a large volume of data, and can explore all the trace size (i.e., it would detect an attack only after it happens).

Two experiments were defined. The first one consists of growing the window between 0% and 4% in small steps of 0.5%, representing a growth size of about 1-2 system calls each time. Both experiments were conducted using only the *Random Forest* classifier, given that it was the best one in the previous experiments. We tested these experiments with the filtered and raw versions of our dataset. Figure 2a shows the results of this evaluation, demonstrating a rapid growth in the *f1-score* for the very first three lowest window values. Despite the difference between the observations filtering or not the system calls, the growth always occurs. However, the strategy of filtering the syscalls provides a *f1-score* above 90% using only 1% of the trace size (which consists of 6 system calls).

Finally, Figure 2b shows the results when growing the window size from 0% to 100% by 10% steps. This represents a growth of about 59-60 system calls each step. The global scenario shows the impact of system call filtering and demonstrates that online approaches can be carried out, since using only 10% of the trace it is possible to obtain an adequate *f1-score*, which does not tend to vary significantly with the growth of the window size, up to 100% of the trace size.

VI. Conclusion

In this article, the feasibility of anomaly intrusion detection on a containerized application, using the analysis of sequences of system calls. It evaluated the impact of varying the sliding window sizes on different methods of anomaly detection;

and also evaluated a pre-filtering of the system calls used in the anomaly detection, discarding system calls considered as harmless for system security.

It was possible to identify an improvement in results after performing the filtering of system calls evaluated as harmless. Although this improvement was small, it showed to be an interesting point to be studied and evaluated. As this experiment was inspired by the syscall classification made by [31], a new threat assessment of system calls also remains for future work, since the work does not cover all calls present in modern systems. In addition, some classifiers obtained good results, even when not pre-filtering the system calls.

The study highlights the possibility of using system calls to identify threats within containers; it presented good results even using a smaller set of system calls, allowing the implementation of an online anomaly detector. Thus, considering the growing number of applications using containers, our approach could be implemented to protect them against many type of attacks.

As a future work, new feature extraction algorithms could be considered, to improve the detection performance without increasing the overhead of the system, and the increase of the recall to a more adequate value, over 90%, that will reduce the number of not relevant events being detected. In addition, our dataset could also be improved with more/newer samples of normal and attack behaviors, in order to evaluate the robustness of the proposed approach against them.

ACKNOWLEDGMENT

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil* (CAPES). The authors also thank the UFPR Computer Science department.

REFERENCES

[1] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, 2019.

[2] A. Lam, "New IPS to boost security, reliability and performance of the campus network," *Newsletter of Computing Services Center*, 2005.

[3] W. Yassin, N. I. Udzir, Z. Muda, M. N. Sulaiman *et al.*, "Anomaly-based intrusion detection through k-means clustering and naives bayes classification," in *4th Int. Conf. Comput. Informatics, ICOCI*, 2013.

[4] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[5] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys (CSUR)*, 2018.

[6] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. Chen, "A survey of intrusion detection systems leveraging host data," *ACM Computing Surveys (CSUR)*, 2019.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *IEEE Symp. on Sec. and Privacy*, 1996.

[8] S. Srinivasan, A. Kumar, M. Mahajan, D. Sitaram, and S. Gupta, "Probabilistic real-time intrusion detection system for docker containers," in *Int. Symp. on Sec. in Computing and Communication*. Springer, 2018.

[9] C. I. Systems, "Sequence-based intrusion detection," http://www.cs.unm.edu/ immsec/systemcalls.htm, 1998.

[10] J. Flora and N. Antunes, "Studying the applicability of intrusion detection to multi-tenant container environments," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019.

[11] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015.

[12] M. Mitchell, J. Oldham, and A. Samuel, *Advanced linux programming*. New Riders Publishing, 2001.

[13] J. Cespedes and P. Machata, "ltrace(1), linux manual page," 2013, https://man7.org/linux/man-pages/man1/ltrace.1.html.

[14] ftrace, "perf-ftrace(1) — linux manual page," mar 2018, https://man7.org/linux/man-pages/man1/perf-ftrace.1.html.

[15] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement." in *NDSS*, 2000.

[16] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting, "System call monitoring using authenticated system calls," *IEEE Transactions on Dependable and Secure Computing*, 2006.

[17] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, 2014.

[18] L. Litty, *Hypervisor-based intrusion detection*. University of Toronto, 2005.

[19] S. S. Durairaju, "Intrusion detection in containerized environments," 2018.

[20] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *17th International Middleware Conference*, 2016.

[21] A. S. Abed, C. Clancy, and D. S. Levy, "Intrusion detection system for applications using linux containers," in *International Workshop on Security and Trust Management*. Springer, 2015.

[22] S. S. Alarifi and S. D. Wolthusen, "Detecting anomalies in IaaS environments through virtual machine host system call analysis," in *Int. Conf. for Internet Technology and Secured Transactions*, 2012.

[23] Y. Liao and V. R. Vemuri, "Using text categorization techniques for intrusion detection." in *USENIX Security Symposium*, 2002.

[24] D. Yuxin, Y. Xuebing, Z. Di, D. Li, and A. Zhanchao, "Feature representation and selection in malicious code detection methods based on static system calls," *Computers & Security*, 2011.

[25] W. Wang, X.-H. Guan, and X.-L. Zhang, "Modeling program behaviors by hidden markov models for intrusion detection," in *2004 International Conference on Machine Learning and Cybernetics*. IEEE, 2004.

[26] M. Laureano, C. Maziero, and E. Jamhour, "Intrusion detection in virtual machine environments," in *30th Euromicro Conference, 2004.* IEEE, 2004, pp. 520–525.

[27] "CVE-2014-0160," Available from National Vulnerability Database, CVE-ID CVE-2019-9978., may 2019.

[28] "CVE-2020-25213," Available from National Vulnerability Database, CVE-ID CVE-2020-25213., may 2020.

[29] "CVE-2020-12800," Available from Common Vulnerabilities and Exposures, CVE-ID CVE-2020-12800., may 2020.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[31] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Remus: a security-enhanced operating system," *ACM Trans. on Information and System Security (TISSEC)*, 2002.