Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection

Asbat El Khairi
Siemens AG & University of Twente
asbat.el_khairi@siemens.com

Marco Caselli
Siemens AG
marco.caselli@siemens.com

Christian Knierim
Siemens AG
christian.knierim@siemens.com

Andreas Peter
University of Oldenburg
andreas.peter@uni-oldenburg.de

Andrea Continella
University of Twente
a.continella@utwente.nl

ABSTRACT

Container technology has gained ground in the industry for its scalability and lightweight virtualization, especially in cloud environments. Nevertheless, research has shown that containerized applications are an appealing target for cyberattacks, which may lead to interruption of business-critical services and financial damage. State-of-the-art anomaly-based host intrusion detection systems (HIDS) may enhance container runtime security. However, they were not designed to deal with the characteristics of containerized environments. Specifically, they cannot effectively cope with the scalability of containers and the diversity of anomalies.

To address these challenges, we introduce a novel anomaly-based HIDS that relies on monitoring heterogeneous properties of system calls. Our key idea is that anomalies can be accurately detected when those properties are examined jointly within their context. To this end, we model system calls leveraging a graph-based structure that emphasizes their dependencies within their relative context, allowing us to precisely discern between normal and malicious activities. We evaluate our approach on two datasets of 20 different attack scenarios containing 11,700 normal and 1,980 attack system call traces. The achieved results show that our solution effectively detects various anomalies with reasonable runtime overhead, outperforming state-of-the-art tools.

CCS CONCEPTS

• **Security and privacy → Container security**.

KEYWORDS

Containers; Anomaly Detection; System Calls

ACM Reference Format:
Asbat El Khairi, Marco Caselli, Christian Knierim, Andreas Peter, and Andrea Continella. 2022. Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection. In *Proceedings of the 2022 Cloud Computing Security Workshop (CCSW '22), November 7, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3560810.3564266

CCSW '22, November 7, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9875-6/22/11.
https://doi.org/10.1145/3560810.3564266

1 INTRODUCTION

Container technology has become an effective means of enhancing productivity in IT development processes, especially in cloud environments. To leverage the scalability, portability, and ease of deployment offered by orchestration tools such as Kubernetes [8], many enterprises have moved their workloads to cloud-based containerized environments (e.g., Google GKE). Indeed, consistent with a recent survey conducted by the Cloud Native Computing Foundation (CNCF) [14], 92% of respondents declare they use containers in production, a significant increase over their first survey in 2016 when only 23% of respondents used containers. Despite their ubiquity, containers have some flaws. When asked to cite their major hurdles with containers, 32% of respondents selected security as a top concern, implying that containers can widen an enterprise's threat landscape and expose its mission-critical workloads to significant risks [4].

Container security is commonly viewed as image scanning and vulnerability management [3]. While these are key practices in securing the CI/CD build pipeline, they fail to handle security threats that, per se, only arise at runtime, such as exploitation of zero-day vulnerabilities and internal privilege escalation attacks. Such risks mandate adopting an intrusion detection system (IDS) that monitors containers for previously unknown threats and alerts DevOps and security teams to investigate and mitigate incidents.

Generally, IDS solutions come in two flavors: signature-based [37] and anomaly-based [32]. The signature-based approach compares a process behavior against a list of predefined attack signatures. On the other hand, the anomaly-based method establishes a "normal" baseline to evaluate behavioral deviations. While signature-based solutions detect known attacks with high accuracy, the continual emergence of previously unknown threats compromises their effectiveness, rendering anomaly-based solutions more favorable. Moreover, an IDS can either be deployed as a host-based (HIDS) [40] or a network-based solution (NIDS) [54]. A HIDS detects host attacks by monitoring metrics such as resource usage, system calls, or event logs. On the other hand, a NIDS detects threats at the network level by inspecting traffic flows. Unfortunately, NIDSes struggle to cope with the encryption of network traffic [47] and cannot catch advanced kernel attacks [40, 42], effectively undermining their use in containerized environments. This motivates our work to opt for a host-based approach to detect container anomalies.

HIDSes can rely on multiple host metrics and observation points to distinguish between normal and abnormal activities. Of particular importance are system calls. A system call (syscall) is a

request to the operating system (OS) kernel to provide a service to a userspace program [22]. That is, syscalls are the rawest form of communication between a process and the OS kernel, allowing for fine-grained observables from which to infer solid discriminators for anomalies. Furthermore, most container attacks rely on kernel features to perform privileged tasks [11], leaving discernible relics in their generated syscall patterns. Besides, syscalls can be collected in real-time with minimal overhead compared to other host metrics [16]. In short, all these traits make syscalls a sound data source for detecting container anomalies.

The idea of leveraging syscalls to detect anomalies has already been applied to traditional environments and applications [38]. Some solutions are enumeration-based, such as the sequence time delay embedding (STIDE) [21], representing a process behavior as n-gram syscall sequences and employing a mismatch-based threshold to detect anomalies. Several other works are frequency-based, such as the bag of syscalls (BoSC)[33], which models the frequency of syscalls in short sequences. Probabilistic models, including Hidden Markov models (HMM)[28, 29] and Finite-State Machines (FSM) [65], have also been leveraged to classify syscall patterns, counting on their sequence recognition skills. Other methods rely on syscall arguments and return values to identify anomalous behavior. Remarkably, few works have been proposed to deal with container anomalies. These solutions model container behavior either by 1) combining the STIDE and the BoSC techniques [2] or 2) measuring the frequency of syscalls within short time intervals [39]. While these host and container-based approaches may improve the runtime security for containers, they lack the capabilities to effectively cope with the properties of containerized environments, specifically with the *high scalability* of containers and the *diversity* of container attacks.

High scalability. VM-based and monolithic infrastructures are less dense with fewer workloads to secure [30], compared to containerized environments, which by design, support the scaling of containers (i.e., replicas) on a single host in response to applications' resource usage. According to a recent study [10], there has been an increase in the container-per-host density. The median number of containers running per host reached 30 in 2019, up from 15 in 2018. To put this number into perspective, 30 containers running simultaneously on a single host can result in syscall batches 30 times larger than in traditional environments. This renders HIDSes designed for monolithic and VM-based infrastructures inefficient at handling multiple syscall streams simultaneously. For example, despite their robust modeling of syscalls, HMM-based approaches require large amounts of storage and high computational resources for long syscall traces [2, 63], making their use impractical within containerized environments. The same holds for solutions based on enumeration. Rolling a short window over voluminous traces to identify mismatched sequences is time-consuming, imposing processing delays and desynchronizing real-time activities. Such constraints entail a monitoring scheme that efficiently models container behavior while being computationally lightweight. Our work represents syscalls in an abstract way that simplifies their processing without losing their fundamental semantics.

Diverse attacks. There are two primary vectors by which container attacks can occur. The first vector relates to applications running within containers. For instance, an external attacker can exploit a vulnerability in a public-facing containerized application to bypass an authentication mechanism (e.g., brute force attack), disclose sensitive information (e.g., directory traversal attack), or remotely execute malicious code (e.g., command injection attack). Some of these attacks can also provide an avenue for an external attacker to gain a foothold on a container and further elevate privileges on its underlying host. The second vector relates to kernel vulnerabilities (e.g., *Dirty Cow* [64]) and container misconfigurations (e.g., exposed docker socket). Unfortunately, the OS's isolation techniques such as namespaces [52] and capabilities [27] cannot prevent an adversary inside a container from exploiting kernel vulnerabilities or excessive capabilities to escape its environment. In short, adversaries can attack containers from multiple angles with different techniques, thus yielding various patterns of syscall anomalies. Unfortunately, existing HIDSes cannot generically capture those diverse anomalies. For example, techniques based on the order of syscalls are limited to merely detecting attacks that break the execution flow. Hence, any stealthy exploit that leverages syscall arguments without violating their ordering would pass undetected. On the other hand, solutions that rely solely on syscall arguments cannot detect attacks that manifest in high syscall frequency. This suggests that a generic detection of container attacks requires the analysis of multiple aspects of syscalls. Therefore, we base our detection scheme on consolidating different syscall properties.

To account for these challenges, we introduce an unsupervised anomaly-based HIDS that raises alarms when a container behaves abnormally. Our key observation is that anomalies manifest in various patterns, requiring the monitoring of *multiple syscall properties*. Further, to fully leverage such properties and distinguish anomalies from normal behavior, our key insight is to analyze the executed syscalls within their *context*, defined by their preceding and succeeding syscalls. Upon this, we develop an automated host-based anomaly detection system that learns the "normal" behavior of each container individually and flags behavioral deviations in production. We evaluate our approach on 11,700 normal and 1980 attack syscall traces of 20 different attack scenarios, and we demonstrate that our system is effective and detects 19 out of 20 attacks with a low false-positive rate and an acceptable runtime overhead.
In summary, we make the following contributions:

- We collect and release a dataset of container syscalls executed during different attack scenarios. To the best of our knowledge, this is the first labeled dataset containing container breakout attacks.
- We perform an empirical analysis of different attack scenarios and identify the main discriminators for container anomalies, as well as the limitations of existing approaches.
- We introduce an anomaly-based approach for detecting container anomalies by analyzing different syscall properties within their context.
- We implement our approach and we show that our solution detects 19 out of 20 attacks with an average precision of 99.29%, significantly outperforming existing approaches.

In the spirit of open science, we make both our tool and the CB-DS dataset available at https://github.com/Asbatel/ContainerHIDS.

Dataset	Threat Impact	CVE/CWE/MISCONF	Vulnerability/Misconfiguration	CVSS	Application	Version
LID-DS (AV: 1)	Authentication Bypass	CWE-307	Brute Force Login	9.8	OpenSSL	1.0.1
		CVE-2012-2122	MySQL Authentication Bypass	7.5	Oracle MySQL	5.1.62
	Information Disclosure	CVE-2014-0050	Heartbleed	5.0	OpenSSL	1.0.1
		CVE-2017-7529	Nginx Integer Overflow	7.5	Nginx	0.5.6
		CVE-2018-3760	Sprocket Information Leak	7.5	Sprockets	3.7.1
		CVE-2019-5418	Rails File Content Disclosure	7.5	Rubygem	5.2.2
	Arbitrary Code Execution	CWE-89	SQL Injection	7.5	DVWA	-
		CWE-434-PHP	Unrestricted File Upload - (PHP)	7.5	DVWA	-
		CWE-434-EPS	Unrestricted File Upload - (EPS)	7.5	Converter (EPS to SVG)	-
		CVE-2019-0191	Zipslip	6.5	Apache Karaf	4.2.2
CB-DS (AV: 2)	Container Breakout	CVE-2016-9962	(ent: Shellshock) File-descriptor Insecure Access	6.4	E-shop + Bash	4.2.2
		MISCONF	(ent: Shellshock) Docker Socket Abuse	-	E-shop + Bash	4.2.2
		CVE-2019-5736	runC Overwrite	9.8	E-shop + Docker Engine	18.06.1
		CVE-2022-0847	Dirty Pipe	7.8	E-shop	-
		CVE-2022-0492	RELEASE_AGENT Abuse	7.0	E-shop	-
		MISCONF	SYS_MODULE Abuse	-	E-shop	-
		MISCONF	SYS_ADMIN Abuse	-	E-shop	-
		MISCONF	MKNOD Abuse	-	E-shop	-
		MISCONF	Host Network Sniffing	-	E-shop	-
		MISCONF	UEVENT_HELPER Abuse	-	E-shop	-

Table 1: Overview of the attack scenarios included in the LID-DS and CB-DS datasets. (*AV*) stands for Attack Vector. We selected four scenarios ▢ to conduct our preliminary analysis.

2 PRELIMINARY ASSESSMENT

To determine how the behavior of a container under attack compares to a container in a normal state, we conducted a preliminary study on a small dataset. Specifically, we examined heterogeneous properties of syscalls and explored how they can translate into decision features to capture anomalies generically.

2.1 Dataset

We use two datasets of container syscall traces representing various routes to attacking containers. While the first dataset comes from a recent work [24], we collected the second dataset to represent container breakout attacks, which are not present in any publicly available dataset. As shown in Table 1, each dataset expresses an attack vector and comprises ten different attack scenarios, categorized by their impacts. Further, to conduct our preliminary assessments, we selected the severest attack scenario (i.e., highest CVSS) from each *threat impact* category to avoid bias in the final evaluation.

LID-DS. The Leipzig Intrusion Detection DataSet (LID-DS) [24] is a HIDS dataset consisting of container syscall traces of 10 different attack scenarios. The authors of LID-DS generated syscall traces using automated interactions on eight various containerized applications. Specifically, they submitted non-malicious inputs and performed random walks to simulate regular interactions, whereas, in the attack mode, they leveraged malicious payloads and penetration tools (e.g., *Metasploit*) to exploit vulnerabilities in the containerized applications. The authors of LID-DS used Sysdig [58] to collect syscalls. Therefore, the resulting traces contain rich metadata such as timestamps, process names, syscalls, and arguments. For each attack scenario, LID-DS has 1,000 normal and 99 attack syscall traces. Each trace represents the behavior of a container for 30 seconds. In total, LID-DS has approximately 10,000 normal traces and 990 attack traces, worth roughly four days of recording.

CB-DS. Since LID-DS lacks container breakout scenarios, we collected the Container Breakout Dataset (CB-DS). Typically, a container breakout attack occurs when an attacker with access to a container exploits a kernel vulnerability or leverages extra capabilities to escape the container environment and elevate privileges on the underlying host. To collect CB-DS, we generated syscall traces via automated interactions on a containerized online-shopping application (*E-shop*). In detail, we performed random walks, payment checkout funnels, and admin debugging tasks (e.g., accessing a container via SSH and checking *Apache* logs) to simulate container normal behavior. For attacks, we replicated 10 common container breakout attacks [26, 41]. In the first two attacks, an external attacker exploits a *Shellshock* [55] vulnerability in the *E-shop* application to access the container. Next, they leverage the mounted and exposed *docker* socket (i.e., `/var/run/docker.sock`) to spin up a new container with either 1) a mounted volume pointing to the host root filesystem or 2) a host PID (i.e., `-pid=host`) to enter the host namespace using `nsenter`. The remaining attacks involve a local attacker with root access to containers (i.e., member of the *docker* group [15]). The attacker leverages an over-permissive container (i.e., running with the `-privileged` or `-cap-add=all` flag) to escape its isolation using eight different techniques, such as exploiting the `sys_module` capability to load a reverse shell module into the OS kernel, executing a malicious payload to overwrite the runC binary [50], and leveraging the *Dirty Pipe* [35] vulnerability to read the `/etc/shadow` file. Similar to LID-DS, our normal scenario consists of 1,700 syscall traces, whereas each of our ten attacks has 99 syscall traces, worth ~23 hours of recording in total.

2.2 Feature Exploration

Most anomaly-based HIDSes rely on a single behavioral attribute to establish their "normal" baseline. As a result, they are limited

Table 2: Preliminary results. The percentage of traces that comprise suspicious syscall observables under normal and attack settings. (-) denotes that no trace contains the suspicious observable.

	Suspicious Syscall Observables					
	Unseen Syscall		Unseen Args		High Frequency	
Scenario	Normal	Exploit	Normal	Exploit	Normal	Exploit
CWE-307	0.03%	-	27.75%	21.42%	0.21%	86.73%
CVE-2017-7529	-	97.72%	-	-	0.22%	-
CWE-434-PHP	0.78%	98.27%	15.14%	98.27%	-	0.09%
CVE-2019-5736	0.52%	99.22%	-	99.18%	0.43%	-

to specific classes of anomalies and cannot perform effectively on a generalized basis, implying that no single attribute can tackle the full spectrum of abnormal behavior. In this work, we propose an approach to detect various anomalies generically. This requires studying prints commonly emitted by different abnormalities. To this end, we first select suspicious syscall-based observables that might discern between normal and abnormal behavior, namely previously unseen syscalls, previously unseen arguments, and high syscalls frequency. We rest this selection on three assumptions: 1) an unseen syscall can indicate that a container has requested unusual kernel service, 2) an unseen argument can reveal that a container has opened a suspicious file, has written to an unexpected directory, or has executed an unusual program, and 3) a high frequency of syscalls can imply that a container has excessively requested many kernel services within a limited time interval. Using our preliminary dataset, we study how a container under attack deviates from its norm with respect to our selected suspicious observables. For each scenario, we calculate the percentage of traces that comprise suspicions under both normal and attack modes. We conduct 4-fold cross-validation using a 25:75 train-test split of normal traces and 99 attack traces as test data. We choose 75% of our normal data as a test set to gain accurate insights into unseen data. In the training phase, we collect three elements: 1) Seen syscalls – syscalls observed in training traces. 2) Seen arguments – here we restrict our attention to only filesystem-related arguments (i.e., *file paths*) that are usually generated when processes are cloned (`clone` syscall) or when files are accessed (`open` or `stat` syscalls) or executed (`execve` syscall) [36]. Moreover, to avoid sensitivity towards application-specific file names (e.g., random cache strings, user uploaded files), we exclude file names for long paths containing more than three subdirectories (e.g., `/var/www/html/e-shop/asset/avatar.png`). However, we maintain the full length of short paths given that sensitive files generally reside in a few subdirectories from the top-level root directory, such as configuration files in `/etc` and command binaries in `/usr/bin`. The third element is 3) the maximum trace length within training traces - the highest number of syscalls that a normal trace contains. In testing, we consider a syscall or argument unseen if it does not exist in the training lists, and the frequency of syscalls is high if the trace length exceeds the maximum training length.

2.3 Feature Engineering

From the standpoint of traditional syscall-based HIDSes, it should be feasible to detect attacks by simply flagging unseen syscalls, unseen arguments, or overstated numbers of syscalls. This perspective stands on two hypotheses: 1) that it is unlikely for a regular operation to invoke an excessive number of syscalls or request kernel services that the process never meant to request, and 2) that attacks usually make use of unusual syscalls and arguments to abnormally access system resources (e.g., memory, filesystem) or conceivably invoke excessive amount of syscalls to crack credentials or encryption keys. While these points may be theoretically plausible, in reality, they do not comprehensively hold. As shown in Table 2, the regular operations of containers can also trigger suspicious syscall observables. For example, the CWE-307 scenario has few normal traces with unseen syscalls, while no attack traces have so. Under the same scenario, unseen arguments are triggered more often during the normal execution of containers than during the attack phase. Generally, we can attribute these observations to two possible reasons: 1) that the behavior of users is usually non-repetitive and, thus, can unintentionally generate unseen syscalls or/and unseen arguments, and 2) that the frequent updates to upstream base OS images (e.g., *alpine:latest*) used in building containerized applications may trigger different execution paths that have never been exerted in training, thus potentially invoking unusual syscalls or arguments. Another observation is related to the frequency increase of syscalls. Our preliminary assessment shows that sometimes, containers can invoke more syscalls in normal mode than in attack mode. For example, in both CVE-2017-7529 and CVE-2019-5736 scenarios, we see a syscall frequency increase more often in normal traces than in attack traces. A plausible reason for this can be traffic-related. Specifically, the number of users visiting an application may differ from the training settings, resulting in an augmented number of syscalls during specific periods. Armed with these observations, we argue that *"it is infeasible to set up a correct training environment that can accurately cover the comprehensive spectrum of container "normal" behavior."* Therefore, the fundamental challenge is to control the trade-off between sensitivity (i.e., a regular activity has accidentally triggered a suspicious syscall observable) and specificity (i.e., a real anomaly has generated a suspicious syscall observable). Having already selected suspicious observables that distinguish between normal and anomalous behavior to some extent, we choose to refine those observables based on two key insights to address the challenge above. Our first insight is that in addition to systematically tracking unseen syscalls and unseen arguments within a time interval, we need to analyze the dependencies of such observables within their surrounding context. We define a context as the n most recent syscalls that occur close in time to a suspicious observable. Also, we define a dependency as the link between two consecutive syscalls that intrinsically relate to each other from a semantic perspective – e.g., binding a socket to an address (`bind`) depends on the socket creation (`socket`). The second insight is to establish a loose upper limit on the syscall frequency to understand the rationale for the increase. Our intuition here is that frequency-based attacks usually produce at least a quintupled number of syscalls compared to the standard. Our evaluation results demonstrate that incorporating both the context around syscalls and a flexible ceiling on their frequency helps us differentiate normal from abnormal behavior properly. While our preliminary assessment uses only four scenarios from our datasets, our results in Section 5 show that the features are also generalizable to other scenarios.

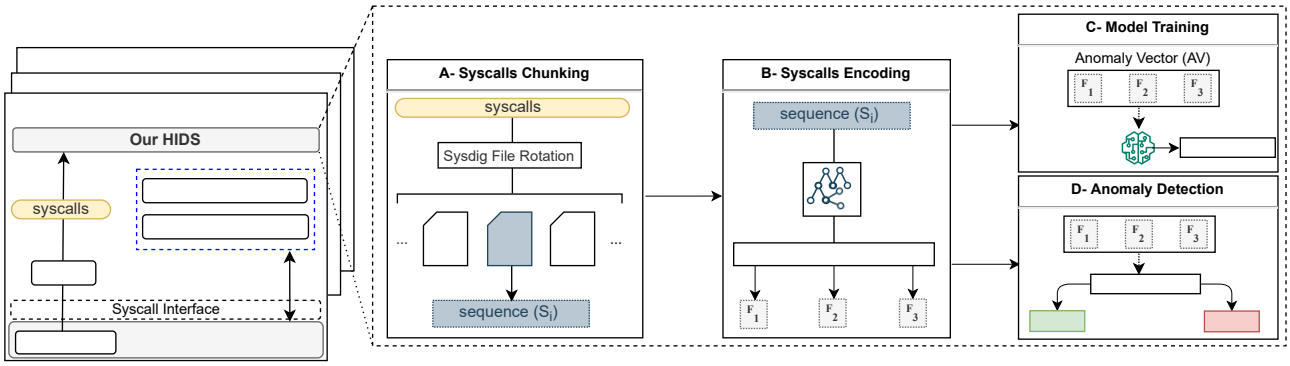


Figure 1: System Overview. (A) We split the continuous flow of syscalls into sequential captures (*scaps*), defined as *syscall sequences*. (B) We convert each sequence (S_i) into a *syscall sequence graph* (SSG_i), from which we construct our *anomaly vector*. Next, we feed the vector into an unsupervised auto-encoder neural network for either (C) training to minimize the training errors or (D) classifying the sequence using the trained model and a selected threshold.

Unseen syscalls influence. We evaluate the abnormality of unseen syscalls by analyzing their contextual influence. This influence is measured based on the number of dependencies to other syscalls occurring relatively close in time. Our view is that unseen syscalls with multiple dependencies within a time interval are likely to play a key role in determining a container activity, thereby considerably altering its behavior. Conversely, unseen syscalls with few dependencies are less likely to influence the operations performed by a container in a specific period, thus minimally varying its behavior.

Unseen arguments influence. Similar to unseen syscalls, we measure the contextual influence of syscalls with unseen arguments by analyzing their dependencies within their context.

Fold frequency increase. According to our preliminary analysis, exceeding the frequency baseline is a sensitive indicator of anomalies. Accordingly, we adopt the fold-increase technique to reasonably determine the frequency's rise intent. We define the fold increase as the increased ratio to the frequency baseline set during training.

3 THREAT MODEL

We consider a container monitoring tool with complete visibility into all communications between the host OS kernel and the running containers. We assume that the monitoring agent is installed on the host and monitors each container separately (i.e., syscalls are filtered based on container ID or name) — monitoring the orchestrator control plane (e.g., kube-apiserver) is out of scope for our work. We consider two types of attackers: 1) an external attacker with access to a public-facing containerized application and 2) an internal attacker with root privileges inside a container. External attackers can exploit vulnerabilities at the application level to possibly read sensitive data, exhaust system resources, or obtain a remote *shell* into a container, while internal attackers can leverage kernel vulnerabilities or extra capabilities to elevate privileges on the underlying host system.

While, in principle, attackers with elevated privileges could interfere with our monitoring system, since Sysdig runs with root privileges on the host, our approach can raise alarms for privilege escalations *before* an attacker can tamper with Sysdig itself, as we show in Section 5. In fact, compromising the kernel requires the execution of (anomalous) syscalls, which are analyzed according to their context and classified by our system in real time. In addition, we envision a setup in which such captured malicious syscalls are immediately forwarded to an external component (e.g., separate physical machine/co-processor, cloud backend) for analysis, providing additional isolation.

4 APPROACH

We aim to detect anomalies efficiently, with few false alarms and reasonable computational overhead. We build our solution on the premise that anomalies can be accurately identified when multiple syscall properties are examined jointly within their context. As shown in Figure 1, we first break the continuous flow of syscalls into short intervals, defined as *syscall sequences*, each representing a time point in container activity. Next, we convert each sequence into a graph representation, allowing us to contextualize syscalls and extract a feature set, defined as *anomaly vector*, illustrating the extent to which a syscall sequence is abnormal. Last, we feed the generated anomaly vector into an auto-encoder neural network to perform training or detection. In training, our network learns to minimize the training errors, while in detection, we classify syscall sequences using the trained model and a selected threshold. In short, our approach consists of three phases: syscalls chunking, syscalls encoding, and model training or anomaly detection.

4.1 Syscalls Chunking

We use Sysdig as a tracing tool to collect syscalls. Sysdig operates at both the kernel and the userspace. At its core, it uses a kernel module (i.e., *sysdig-probe*) to intercept and push syscalls to userspace, where parsing and decoding occur (e.g., resolving *file descriptors* numbers into human-readable data [19]). The first step of our approach is to break the continuous capture of syscalls into multiple *scap* files. Each file is a syscall sequence representing the container's behavior for a timespan τ. We leverage Sysdig's file rotation technique to perform the capture split. Specifically, we create a policy that defines the duration τ of each generated *scap* file and their retention period before being overwritten to restrict

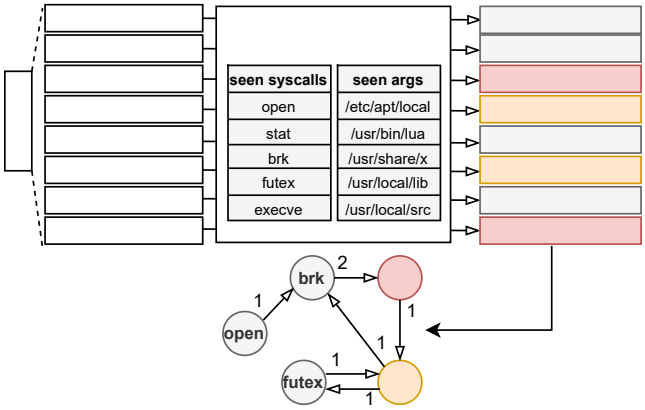


Figure 2: Overview of the creation of a syscall sequence graph *(SSG)* from a syscall sequence. The *SSG* nodes represent distinct syscalls in a sequence, directed edges represent moves from syscalls to their respective successors, and weights represent the number of times ordered pairs of syscalls occurred in a sequence. The *USN* represents unseen syscalls while the *UAN* represents syscalls (`stat`, `open`, `execve`, `clone`) with unseen arguments.

the disk space. Our selection of the timespan τ rests on two motivations: 1) 49% of containers live less than five minutes [59], which requires monitoring at short intervals, and 2) monitoring extremely short intervals (e.g., 50 milliseconds) is insufficient to characterize container behavior and properly contextualize syscalls. Thus, we experimentally define the sequence duration τ as one second (1s).

4.2 Syscalls Encoding

Given our preliminary assessment results, the encoding of syscalls should consider the frequency increase and the influence unseen syscalls and unseen arguments exert within a sequence. To this end, we leverage a graph structure to model syscalls. We rest this selection on the perspective that a graph representation of syscalls provides a rich data structure from which to extract solid behavioral features as efficiently as possible. It is noteworthy that organizing syscalls in a graph has already been explored in malware detection and classification and has shown good detection capabilities and robustness against malware obfuscation techniques [44, 57].

4.2.1 Syscall Sequence Graph (SSG). We convert each syscall sequence to a weighted directed graph, referred to as syscall sequence graph $SSG = (N, E, W)$. In our setting, N is the set of nodes, each representing a distinct syscall in a sequence – here, we define one single node to represent all unseen syscalls in a sequence, which we refer to as the unseen syscall node (*USN*). In a similar manner, we define the unseen argument node (*UAN*) to account for syscalls (`open`, `stat`, `execve`, `clone`) with unseen arguments. This representation enables us to reduce the number of nodes in the *SSG* for lightweight graph processing. To illustrate, as Linux has roughly 380 syscalls, a *SSG* graph can have only a maximum of 381 nodes, and this would happen only when a sequence has 1) a syscall or syscalls with unseen arguments, making the *UAN* node, and 2) all Linux syscalls (previously seen), representing the other 380 nodes. Next, E is the set of directed edges, each representing a move from a syscall sc_i to the succeeding syscall sc_{i+1}. W represents the weights of edges i.e., how frequently an ordered pair of syscalls appeared in a sequence. Figure 2 demonstrates the construction process of the *SSG* from a syscall sequence.

4.2.2 SSG Properties. We construct our feature set by leveraging two properties of the *SSG* graph, namely the degree centrality [7] and the graph size [46].

Degree centrality. In graph theory, the centrality metric captures the topological importance of nodes based on different metrics, namely degree, betweenness, and closeness [6]. Aiming to assess the influence of unseen syscalls and unseen arguments within their context, we use the degree centrality to evaluate the dependencies of the unseen syscall node (USN) and the unseen argument node (*UAN*) within their respective *SSG*. Further, since the *SSG* is a directed graph, we express the degree centrality using two sub-metrics: in-degree and out-degree, each reflecting a different interpretation of "influence". The in-degree centrality (*IDC*) reflects the prominence of a node by counting its incoming links, whereas the out-degree centrality (*ODC*) measures its impact by counting its successors [48].

SSG size. In our setting, each edge in the *SSG* corresponds to a dependency between two adjacent syscalls. Additionally, every edge is assigned a weight representing how frequently a particular dependency occurs. Thus, an abnormal syscall sequence that involves an overstated number of syscalls can be reflected in the number of dependencies within its corresponding *SSG*. With this detail, we represent the frequency of syscalls using the *SSG* size, which is simply the sum of all its edges' weights.

4.2.3 Unseen Syscall Influence (USI). As shown in Equation 1, we calculate the in-degree centrality of the *USN* node by dividing the number of its incoming edges by $N-1$, while we calculate its out-degree centrality by dividing the number of its outgoing edges by $N-1$, where N is the number of nodes in the *SSG* graph. Generally, the in-degree and out-degree centralities are similar, and can only differ for sequences beginning or ending with an unseen syscall. Hence, we obtain the *USN* centrality by summing both metrics. Additionally, since we aggregate all unseen syscalls under the *USN*, we calculate the *USI* by multiplying the *USN* centrality by D_{us}, the number of distinct unseen syscalls grouped under the *USN*. We speculate that distinct unseen syscalls point to the request of various unusual kernel services. Therefore, the product of the *USN* centrality with D_{us} can properly judge the maliciousness of a sequence.

$$IDC_{usn} = \frac{deg_{in}(USN)}{N-1}\ ,\ ODC_{usn} = \frac{deg_{out}(USN)}{N-1} \quad (1)$$

$$USI = D_{us} \times (IDC_{usn} + ODC_{usn}) \quad (2)$$

4.2.4 Unseen Argument Influence (UAI). Similar to the *USI*, we calculate the *UAI* by considering the degree centrality of the *UAN* node. As shown in Equation 3, we multiply the degree centrality of the *UAN* by two factors: 1) the number of distinct unseen arguments aggregated under the *UAN*, and 2) the number of distinct syscalls (`open`, `stat`, `execve`, `clone`) used to pass unseen arguments. Our view is that distinct unseen arguments D_{ua} can point

to different unusual file paths, while distinct syscalls with unseen arguments D_s can refer to different kernel functions using those file paths. Thus, the combination of both factors with the UAN degree centrality can precisely interpret the suspiciousness of a sequence.

$$UAI = D_{ua} \times D_s \times (IDC_{uan} + ODC_{uan}) \quad (3)$$

4.2.5 Frequency Increase. (FI). We leverage the SSG size property to capture the frequency of syscalls. We obtain the SSG_{size} of a sequence by summing all its edges' weights. Next, we use the fold increase technique to assess the nature of the frequency rise. As shown in Equation 4, we calculate the FI as the ratio of the SSG_{size} to the product of the max training SSGs' size and a configurable constant β, which we define experimentally as 5.

$$FI = \frac{SSG_{size}}{\alpha \times \beta}\ ,\ \alpha = max\left\{SSG_{tr_{size_1}}, .., SSG_{tr_{size_n}}\right\} \quad (4)$$

4.2.6 Anomaly Vector. We represent each sequence by an anomaly vector (AV). This vector consolidates the extracted features (USI, UAI, and FI) to jointly articulate the extent to which a syscall sequence deviates from the normal baseline. It is important to note that the extracted anomaly vector (AV) is vastly smaller than the original syscall sequence, leading to a short sequence execution time, as we demonstrate in Section 5.

4.3 Model Training

To build our models, we leverage the auto-encoder neural network. The latter is an unsupervised deep learning algorithm [62], consisting of 1) the encoder to convert the input to low-dimensional representation and 2) the decoder to reconstruct the original input from its compressed representation. In training, we feed the auto-encoder with "normal" anomaly vectors to minimize the reconstruction error. In testing, if the trained model comes across "abnormal" vectors that vary from the training vectors, the model reproduces those vectors with a significant reconstruction error. Therefore, our approach uses the reconstruction error as a threshold to detect anomalies.

4.4 Anomaly Detection

To determine a reasonable threshold for detecting anomalies, we first test the training data against our trained model to collect the training reconstruction errors. Next, we adopt a customized function to select an optimal detection threshold. As shown in Equation 5, we select the threshold based on the product of 1) the max of the training reconstruction errors and 2) a constant γ. We evaluate γ at multiple values ranging from 0.2 to 2 with a 0.2 step. In this range of values, we also assess thresholds that closely sit outside our training reconstruction errors list. In general, the choice of γ emphasizes the trade-off between detection and false positive rates.

$$D_{thresh} = \gamma \times max\left\{RE_{tr_1}, .., RE_{tr_n}\right\} \quad (5)$$

5 EVALUATION

In this section, we present our experimental evaluation. We evaluate our approach on both the LID-DS and CB-DS datasets.

Autoencoder neural network setup. Our auto-encoder consists of four layers with the Sigmoid activation function. We implement the network using Keras [13] with Tensorflow [1] as a back-end. We set two neurons in the first and fourth hidden layers and one in the bottleneck layers. Each input trains the model for 120 epochs. We utilize the Adam optimizer with 0.001 as a learning rate to minimize the reconstruction error. After training the network, we evaluate the reconstruction errors generated by the training data to select a proper detection threshold.

Table 3: Parameter Selection. The first row highlights the default values for the parameters and the next rows show the optimal values found for each parameter.

β	τ	Avg. AUC%	Avg. sequence execution time (s)
1.5	100ms	84.73	0.71
10	100ms	97.72	0.71
5	100ms	97.98	0.71
5	2s	99.22	0.88
5	**1s**	99.22	0,84

5.1 Parameter Selection

As described in Section 4, our solution requires two configurable parameters to monitor container behavior and detect anomalies:

- $\boldsymbol{\beta}$ controls sensitivity to syscalls frequency.
- $\boldsymbol{\tau}$ specifies the duration of the sequence.

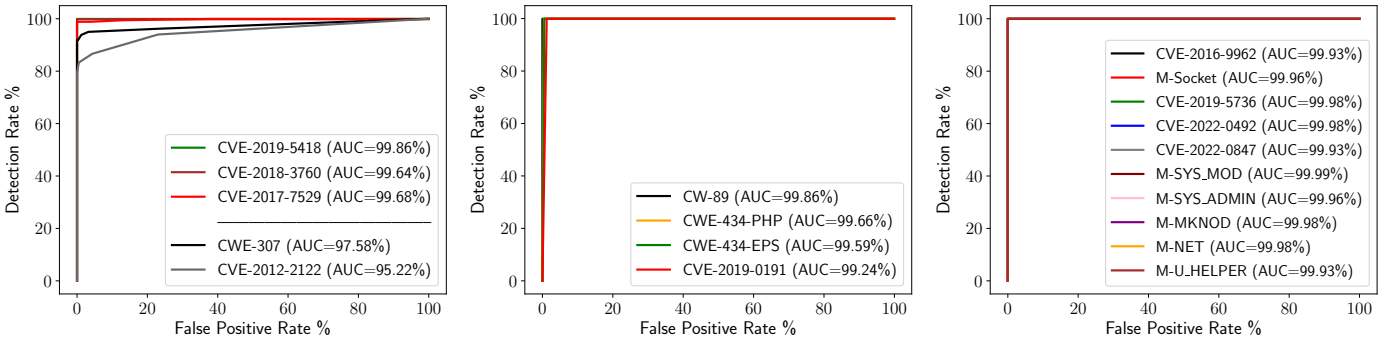
Optimization metric. We empirically select each parameter by optimizing the Area Under the Curve (AUC) [45] and the sequence execution time. The AUC metric assists in measuring the capability of our parameters in differentiating between normal and abnormal container behavior at different threshold levels, while the sequence execution time assesses the time spent by our approach to encode and classify a syscall sequence.

Parameter selection. We test each parameter individually using the following values:

- $\boldsymbol{\beta}$: 1.5, 3, 5, 7 and 10.
- $\boldsymbol{\tau}$: 100ms, 200ms, 500ms, 1s and 2s.

The fold-frequency increase β ranges from 1.5 to 10, with smaller values sensitive to minor frequency increases in normal syscall sequences, while larger values might overlook slight frequency increases within anomalous sequences. For the sequence duration τ, its values range from 100ms to 2s, where shorter periods may contain no syscalls and, thus, lead to poor container behavior characterization. In contrast, extended periods may cause processing delays due to the number of executed syscalls. We iterate over each parameter's possible values while maintaining the rest of them at their default values. When an optimal value is found for a parameter, that value becomes the new default for optimizing other parameters. We conduct a 4-fold cross-validation analysis for each setting on our preliminary dataset. Although the optimal parameters may be biased toward our preliminary dataset, our evaluation results demonstrate that the selected values are generalizable to other scenarios.

As shown in Table 3, we find optimal values for β=5 and $\tau = 1s$. Nonetheless, DevOps and security teams can further tweak these parameters according to their policies and setups in production.



(a) Information Disclosure & Authentication Bypass **(b) Arbitrary Code Execution** **(c) Container Breakout**

Figure 3: AUC scores among different attack scenarios.

Table 4: Performance of our approach compared to related works.

	Our Approach				CDL				STIDE-BoSC			
Scenario	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Precision**	**Recall**	**F1-score**	**Accuracy**
CWE-307	0.9942	0.9183	0.9547	0.9565	0.9898	0.9183	0.9527	0.9544	0.6975	0.9795	0.8148	0.7774
CVE-2012-2122	0.9983	0.7935	0.8842	0.8961	0.9748	0.6645	0.7902	0.8236	0.6195	0.9935	0.7631	0.6917
CVE-2017-7529	0.9946	0.9885	0.9915	0.9916	0.2447	0.0057	0.0112	0.4940	0.9850	0.9885	0.9867	0.9867
CVE-2018-3760	0.9960	0.9998	0.9979	0.9979	0.9717	0.8248	0.8922	0.9004	0.7116	0.9999	0.8315	0.9773
CVE-2019-5418	0.9986	0.9989	0.9987	0.9987	0.3642	0.0102	0.0198	0.4961	0.7497	0.9999	0.8569	0.8331
CW-89	0.9986	0.9987	0.9987	0.9987	0.9693	0.9999	0.9844	0.9841	0.5687	0.9999	0.7250	0.6208
CWE-434-PHP	0.9933	0.9987	0.9961	0.9961	0.9494	0.9902	0.9694	0.9687	0.5638	0.9999	0.7211	0.6133
CWE-434-EPS	0.9304	0.9988	0.9634	0.9620	0.9344	0.4848	0.6384	0.7254	0.6613	0.9999	0.7961	0.7440
CVE-2019-0191	0.9881	0.9987	0.9934	0.9934	0.7356	0.1224	0.2099	0.5392	0.5165	0.9999	0.6811	0.5320
CVE-2016-9962	0.9974	0.9962	0.9968	0.9968	0.7132	0.0201	0.0389	0.5059	0.5686	0.9999	0.7250	0.6207
M-Socket	0.9974	0.9937	0.9955	0.9956	0.8602	0.0495	0.0936	0.5207	0.5686	0.9999	0.7250	0.6207
CVE-2019-5736	0.9974	0.9912	0.9943	0.9943	0.7869	0.0297	0.0572	0.5108	0.5686	0.9999	0.7250	0.6207
CVE-2022-0492	0.9974	0.9912	0.9943	0.9943	0.0000	0.0000	0.0000	0.4959	0.5599	0.9651	0.7086	0.6032
CVE-2022-0847	0.9974	0.9887	0.9930	0.9931	0.9792	0.3802	0.5475	0.6859	0.5686	0.9999	0.7250	0.6207
M-SYS_MOD	0.9975	0.9987	0.9981	0.9981	0.9897	0.7802	0.8724	0.8859	0.5686	0.9999	0.7250	0.6207
M-SYS_ADMIN	0.9974	0.9962	0.9968	0.9968	0.8326	0.0401	0.0763	0.5159	0.5686	0.9999	0.7250	0.6207
M-MKNOD	0.9974	0.9937	0.9955	0.9956	0.5518	0.0101	0.0194	0.5009	0.5686	0.9999	0.7250	0.6207
M-NET	0.9974	0.9912	0.9943	0.9943	0.9878	0.6534	0.7865	0.8227	0.5637	0.9801	0.7157	0.6108
M-U_HELPER	0.9975	0.9987	0.9981	0.9981	0.5542	0.0103	0.0196	0.5009	0.5686	0.9999	0.7250	0.6207

5.2 Results Analysis

We evaluate our approach by running 4-fold cross-validation on both LID-DS and CB-DS. Since the datasets comprise syscall traces and not sequences, we consider a trace malicious if one of its sequences is malicious. Figure 3 illustrates the performance of our approach in terms of the AUC score. The plots show that our approach produces average scores of 99.97%, 99.72%, and 99.58% for container breakout, information disclosure, and arbitrary code execution categories, respectively. This illustrates the excellent trade-off between detection and false-positive rates achieved across different attack scenarios. However, our approach generates a slightly low AUC of 96.40% for the authentication bypass category. To understand the cause, we conduct a new evaluation of our approach at only one threshold using precision, recall, F1-score, and accuracy. We select the threshold based on the optimal γ value, which we empirically define as γ=1.4. According to Table 4, we achieve a recall of 79.35% and 91.83% in the *CVE-2012-2122* and *CWE-307* scenarios, respectively. We attribute that to two causes: 1) our approach fails to raise alarms in 11.07% of the attack syscall traces, showing that attacks can sometimes succeed without emitting suspicious syscall observables, as we discuss in further detail in Section 5.7, and 2) detecting high-frequency attacks is always sensitive to the configurable upper limit β set during training. Specifically, very loose β can always overlook attacks exhibiting a slight syscall frequency increase, while very tight β can flag minor frequency increases manifested in normal behavior. Yet, we achieve high precision and recall rates across the other 17 attack scenarios, suggesting that our approach incorporates solid features capable of learning the underlying patterns of container behavior and correctly classifying its activities. Nonetheless, as we discuss in Section 7, our approach shows some limitations concerning the Heartbleed scenario.

5.3 Comparison with Existing Approaches

We compare our approach with state-of-the-art anomaly-based detection solutions, namely STIDE-BoSC [2] and CDL [39]. Unfortunately, neither of these container-based approaches published their code or raw datasets. Hence, to evaluate these approaches on the LID-DS and CB-DS datasets, we faithfully reimplemented their decision engine techniques.

CDL. In this work, the authors convert syscalls into time-based sequences, each representing 100 milliseconds of syscalls. Further,

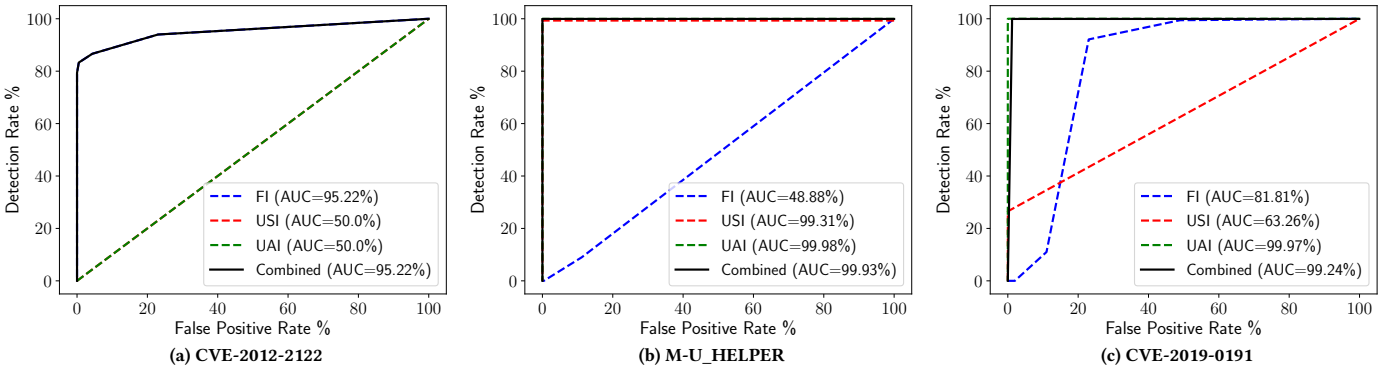


Figure 4: Ablation Study. AUC scores of each individual feature of the anomaly vector among different attack scenarios.

they encode each sequence into a frequency vector, which is further fed to an autoencoder neural network for either training or detection. In training, the network learns to reduce the training reconstruction errors. After sufficient training, the authors select a detection threshold as the 99.99 percentile of the reconstruction errors to deploy their models. In our settings, we apply the same sequence duration (i.e., 100ms) and percentile threshold (i.e., 99.99) across the entire trace. We deem a trace malicious if one of its sequences is classified as malicious. As shown in Table 4, the CDL approach works well for attacks exhibiting a high number of syscalls. However, it fails to detect attacks emitting different syscall patterns, such as unseen arguments. Therefore, CDL achieves a precision rate of 75.72% on average.

STIDE-BoSC. This work incorporates both the STIDE and BoSC techniques. Specifically, the authors construct a bag of syscalls by rolling a window of size ten across syscall traces and recording the occurrence of every call within each window. This work operates in two modes: training and detection. In training, the authors convert training syscall traces into a list of BoSCs to construct a database that represents containers' normal behavior, while in testing, they read syscalls in groups, defined as epochs. For each epoch, the authors roll a window of size ten to construct BoSCs and verify their presence in the normal database. A mismatch is declared if a BoSC does not exist in the normal database. On a dataset collected by authors, the STIDE-BoSC approach yields a detection rate of 100% and a false alarm rate of 2% using an epoch size of 1000 syscalls and a threshold of 10 mismatches. In our setup, we apply the same epoch size and mismatch threshold across the entire trace. We deem a trace malicious if one of its epochs is classified as malicious. Table 4 shows that combining the STIDE and bag of syscalls achieves good detection results but with a significant number of false positives. This can be attributed to the occurrence of previously unseen BoSC sequences also during normal activities. Therefore, the STIDE-BoSC approach achieves a precision rate of 61.82%.

5.4 Ablation Study

Analyzing multiple properties of syscalls is essential for detecting various container attacks. In our work, we build the anomaly vector by consolidating three key features: *USI*, *UAI*, and *FI*. In this subsection, we conduct an ablation study to assess each feature's contribution to our approach's performance. Specifically, we evaluate each feature (e.g., *USI*) by keeping the remaining two features (e.g., *FI*, *UAI*) constant across normal and malicious data (i.e., zero-variance). Using the AUC score, we compare the features' performance separately and together over three different attack scenarios, each from a different *threat impact* category. As shown in Figure 4, *FI* is the main discriminator in the *CVE-2012-2122* scenario. In detail, both normal and attack sequences do not contain unseen syscalls or arguments. Therefore, relying solely on *USI* or *UAI* leads to an identical encoding of normal and attack sequences, thus rendering a random classifier with an AUC score close to 50%. The *M-U_HELPER* scenario involves both *USI* and *UAI* as the main contributors. Here, attackers attempt to maliciously write to sensitive directories and backdoor executables, generating unseen syscalls and arguments. In the case of *CVE-2019-0191*, *UAI* is the main differentiator. In this scenario, attackers try to overwrite arbitrary files in the filesystem, yielding several unseen arguments. In summary, counting on one feature cannot gauge the full scope of container "anomalous" behavior. Therefore, it is necessary to incorporate multiple syscall properties describing different aspects of container behavior to detect attacks generically.

5.5 Assessment of Execution Time

Our approach's effectiveness also lies in its time complexity. In this subsection, we evaluate the sequence execution time of our approach compared to related works. The time complexity of our system relies on 1) the number of syscalls in a sequence and 2) the number of containers running on the host. Thus, we conducted our experiment under two settings. In the first setting, we run a *Flask-python* web application that yields different numbers of syscalls within a sequence ($\tau = 1s$). The number of syscalls ranges from 100 syscalls, a scenario where only a few users are browsing the application, and 20,000 syscalls, representing high application traffic. In the second setting, we spin different numbers of containers (i.e., replicas), ranging from one container illustrating a significantly less

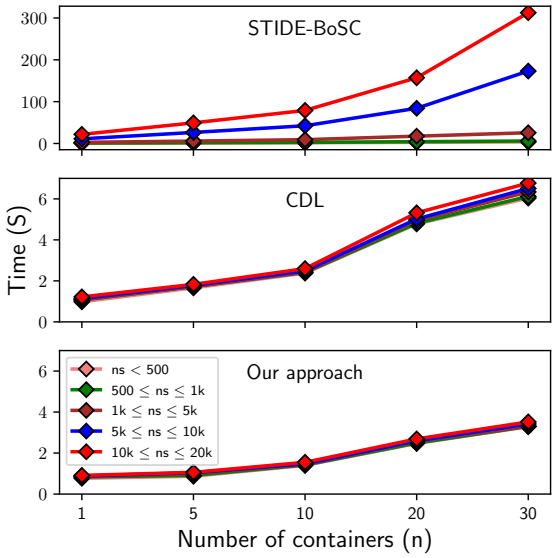


Figure 5: Average sequence execution time of our approach and related works when monitoring *n* containers.(*ns*) refers to the number of syscalls executed within a sequence.

dense host and 30 containers representing a high container-per-host density. We assessed the sequence execution time on a Lenovo Thinkpad P15 laptop with an Intel Core i7-10850H CPU 2.70GHz processor. Using 8 CPU cores, we parallelize the classification of syscall sequences generated simultaneously by running containers. Figure 5 shows our approach's average sequence execution time compared to CDL and STIDE-BOSC. Using 100 runs on each setting, our solution processes and classifies syscall sequences faster than related works. We achieve a sub-linear time complexity as the number of syscalls increases within a sequence and the number of running containers increases on the host. Specifically, we process and classify 30 sequences of 10,000 to 20,000 syscalls simultaneously in 3.52 seconds on average, faster than CDL and STIDE-BoSC by 3.25 seconds and 5.15 minutes, respectively. We attribute the achieved time complexity to our lightweight graph-based syscalls encoding and our auto-encoder network's shallow architecture. To illustrate, the average size of a *scap* file representing a raw sequence of 20,000 syscalls is 6.10 MiB, while its anomaly vector is only 36 Bytes, leading our network to process sequences faster. However, despite the lightweight frequency-based encoding of CDL, the use of a reasonably sizeable auto-encoder network makes it relatively slower. For the STIDE-BOSC approach, encoding and testing numerous short 10-grams within large syscall sequences is time-consuming, leading to minutes of processing and, thus, to a quick desynchronization with real-time activities. In short, our approach can monitor containers even at the peak container-per-host density with reasonable latency, making it practical in practice.

Table 5: Malicious trace from the CWE-434-PHP scenario. The attack is launched at **t=09:52:42.052274**, first detected at **t=09:52:43.052274**, last detected at **t=09:52:49.052274** and completed at **t=09:52:50.052274**. The lead time is **6 seconds**.

Timestamp	Anomaly Vector		
	FI	USI	UAI
09:52:40.052274	0.0221	0.0000	0.0000
09:52:41.052274	0.0472	0.0000	0.0000
09:52:42.052274	0.0181	0.0000	0.0000
09:52:43.052274	**0.0318**	**3.1956**	**3.0434**
09:52:44.052274	**0.0135**	**0.5500**	**0.3001**
09:52:45.052274	0.0593	0.0000	0.1777
09:52:46.052274	0.0287	0.0000	0.0000
09:52:47.052274	0.0319	0.0000	0.0000
09:52:48.052274	0.0761	0.0000	0.1666
09:52:49.052274	**0.0724**	**15.897**	**61.384**
09:52:50.052274	0.0571	0.0000	0.0000

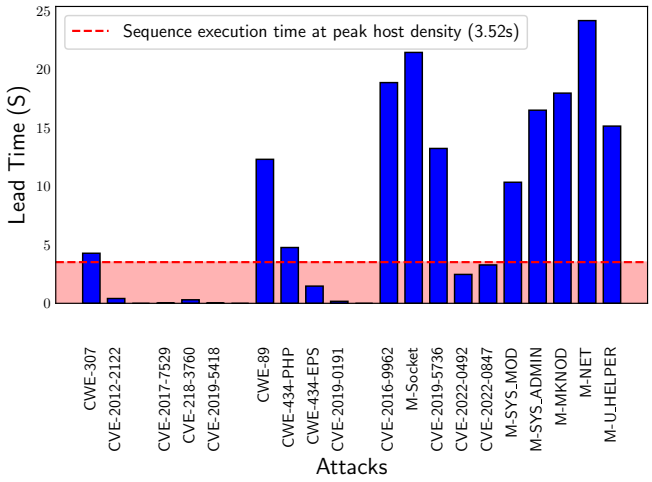


Figure 6: Average lead time of our solution on different attacks. Attacks that fall in the area are completed before being detected during the peak host density (30 containers).

5.6 Assessment of Attack Lead Time

The effectiveness of an IDS in real-world settings also relies on the attack detection lead time. The lead time is the duration from detecting the first symptoms of an attack to the time of the attack completion. That is to say, it is the period during which DevOps and security teams can act to prevent further damages (e.g., kill a compromised container to prevent lateral movement or privilege escalations to the underlying host system). In this subsection, we measure the lead time achieved by our solution. As shown in Figure 6, the lead time differs among attacks due to two factors: 1) the longitude of the attack and 2) the attacker's knowledge of the system. Generally, attacks allowing an unprivileged adversary to modify or execute root processes (e.g., tampering with Sysdig) often require multiple steps and are thus more likely to be detected early. For example, in the *CVE-2016-9962* attack, the exploit chain involves 1) *shellshock* to obtain a shell inside the victim container, 2) a reconnaissance round to know that the docker socket is mounted and exposed, and 3) creation of a container with the host's PID

namespace. Therefore, the detection of *shellshock* makes our solution generate a lead time of 18.87 seconds ahead of the privilege escalation. The same applies to the *M-SYS_MODULE* attack. The alarm triggered by downloading the *kmod* tool to load the reverse shell module into the kernel accelerates the detection of the attack by 21.45 seconds. However, attacks that involve a few steps can generate short to zero lead time. For example, in the *CVE-2019-0191* attack, the combination of the directory traversal technique with the attacker's knowledge of the system (i.e., the number of directories to climb) renders the attack very instant with zero lead time, making its early detection infeasible under our solution. Yet, our approach achieves a good detection lead time across ten attacks, even at the peak container-per-host density.

5.7 Case Study

We describe how our detection approach identifies various types of attacks, highlighting two examples from each dataset.

Sprocket Information Leak (CVE-2018-3760). Early versions of Sprockets have a vulnerability that allows attackers to craft malicious requests to access files outside an application's root directory on the filesystem [18]. Based on our analysis, the attack is a two-step procedure. First, the attacker attempts to directly access the `passwd` file in `/etc`, resulting in the `FileOutsidePaths` error message with a list of existing paths. This causes multiple unseen arguments (i.e., paths) to manifest in both `stat` and `open` syscalls. Second, armed with existing paths, the attacker leverages the directory traversal technique to retrieve the `passwd` file via crafted paths, generating `'../../../etc/passwd'` and `'%2e%2e/%2e%2e/etc/passwd'` as unseen arguments. Last, the unusual access to `passwd` triggers changes in Sprockets' asset cache directory, yielding `mkdir` and `rename` as unseen syscalls. In short, both unseen syscalls and unseen arguments lead to high *USI* and *UAI*, thereby making our solution detect this attack effectively.

MySQL Authentication Bypass (CVE-2012-2122). Early versions of Oracle MySQL have a vulnerability that enables attackers to circumvent authentication and obtain root access to the database [17]. In particular, an attacker can bypass authentication by repeatedly entering the same wrong password until the login is successful. Due to this trial and error, seven syscalls occur at a high rate, namely `getpeername`, `setsockopt`, `fcntl`, `read`, `write`, `shutdown`, and `close`. In detail, `getpeername`, `setsockopt`, and `fcntl` initially obtain the socket remote address and set its parameters. Next, `read` and `write` exchange data during the authentication process. In case of authentication failure, both `shutdown` and `close` end the connection. During the attack, these syscalls recur multiple times to authenticate successfully, yielding high syscall frequency and, subsequently, a high *FI*. However, this attack can sometimes succeed randomly with only a few login attempts [43]. That is, an attacker can enter the same wrong password only a few times to bypass authentication, yielding a low *FI*. This makes our solution detect this attack with a considerable false-negative rate.

Release_Agent Abuse (CVE-2022-0492). A logical flaw in the kernel `cgroups` allows attackers to leverage the `release_agent` feature to escape container and elevate privileges on its underlying host [12]. In detail, when a process terminates in *cgroups* and the `notify_on_release` flag is enabled, the kernel executes the `release_agent` file with elevated privileges. This suggests that if an attacker succeeds in writing to `release_agent`, they can make the kernel run arbitrary code as root. Our analysis of this attack shows that the attacker first accesses an over-permissive and unhardened container (i.e., `–security-opt seccomp=unconfined`), leverages the `sys_admin` capability to mount the `cgroupfs` driver, and creates a child `cgroup`. These steps generate `mount` and `mkdir` as unseen syscalls and `/usr/sbin/mount`, `/sbin/mount.cgroup`, and `/usr/sbin/mkdir` as unseen arguments. Next, the attacker sets `notify_on_release` to 1 to enable the release notification, yielding `dup` as an unseen syscall. Further, the attacker retrieves the container's `OverlayFS` path on the host from `/etc/mtab`, which allows for `readlink` as an unseen syscall and `/sbin/sed` as an unseen argument. Then, the attacker creates a malicious script and places it under the retrieved path, editing the `release_agent` file to run it, which induces some file creation related unseen syscalls, namely `dup` and `pipe`, followed by some permissions related unseen syscalls such as `umask`, `fchmodat`, and `setpgid`. Last, the attacker spawns the process that directly terminates inside the child `cgroup`, which yields `dup` as an unseen syscall and `/sbin/sh` as an unseen argument, allowing the attacker to execute malicious code with elevated privileges. In short, this attack induces high *USI* and *UAI*, hence making it easily discernable by our solution.

Dirty Pipe (CVE-2022-0847) An improper initialization flaw was found in `copy_page_to_iter_pipe` and `push_pipe` functions in the Linux kernel since 5.8 [5]. Usually, the CPU handles `pipe` data in memory pages. When a page becomes full, the kernel sets the `PIPE_BUF_FLAG_CAN_MERGE` flag on the page cache to merge data between `pipe` pages without rewriting data to memory. Unfortunately, when the page cache is cleared, the merge flag is retained, allowing attackers to disclose or edit interesting root files. According to our analysis, the attacker first downloads the `wget` utility to retrieve the compiled malicious payload from a server and changes its mode to executable, resulting in some socket-related unseen syscalls, namely `sendto`, `recvfrom`, and `getsockopt`, followed by some file-permission unseen syscalls such as `chmod`, `fchmodat`. Also, this yields some unseen arguments such as `/etc/apt/sources.list` and `/usr/bin/chmod`. Furthermore, the attacker executes the payload to access the content of some root files. This payload performs three tasks: 1) opens a pipe, which allows for `pipe` as an unseen syscall, 2) sets the merge flag by filling the page caches, and 3) clears and replaces the pipe with data the attacker wants to access, allowing for `splice`, `mremap`, and `dup` as unseen syscalls to perform the splicing of the pages. In short, unseen syscalls and unseen arguments lead to high *USI* and *UAI*, which makes our solution detect this attack effectively.

6 RELATED WORK

Several existing works aim to handle container runtime threats.

Anomaly-based. These works monitor containers based on two metrics, namely resource consumption and syscalls. Various solutions [53, 66] leverage container resource usage (e.g., CPU) to monitor container behavior. Fundamentally, the premise behind such solutions is that if an anomaly occurs in a container, it should trigger a deviation in its resource usage (e.g., crashing a container can consume all its CPU quota). Unfortunately, these solutions

deal merely with attacks that hijack resources, such as the Cryptojacking attack [31]. Thus, attacks that do not abuse container resources can easily circumvent these solutions. Regarding syscall-based solutions, existing container-based HIDSes rely mainly on the frequency of syscalls. Tien et al. [61] incorporate Falco [60] to define syscall-related rules to detect attacks. Specifically, they rely on the frequency at which those rules are triggered. Abed et al. [2] combines STIDE and BoSC to define container normal behavior and further apply a mismatch-based threshold to flag anomalies. Lin et al. [39] introduce a solution that turns syscall streams into timestamped frequency-based vectors and further feeds those vectors to an auto-encoder network for classification. Unfortunately, these solutions are restricted to specific forms of anomalies and cannot be applied to a generalized scale. Therefore, we propose a solution that analyzes and contextualizes different properties of syscalls. The evaluation results show that our solution detects various anomalies and robustly withstands the high scalability of containers.

Policy-based. These tools use rules to define the normal behavior of containers. They can either focus on enforcement or auditing. For example, enforcement tools such as Confine [23], Seccomp [9], SELinux [56], and AppArmor [25] react to anomalies either by blocking syscalls or stopping compromised containers. On the other hand, auditing tools such as Falco and Auditd [34] only raise alarms when a container steps outside its baseline. Unfortunately, the constant emergence of previously unknown threats impedes these tools' effectiveness, rendering anomaly-based solutions more appealing. Thus, we introduce an anomaly-based HIDS that defines the "normal" baseline of each container individually to uncover deviations in production. Our solution shows strong detection capabilities against various anomalies while avoiding the need to write and maintain detection rules.

7 LIMITATIONS

We have shown that our solution detects various container attacks with negligible false alarms. Yet, some aspects of our approach need to be considered in future research.

Failure to detect Heartbleed (CVE-2014-0050). OpenSSL 1.0.1 before 1.0.1f has a vulnerability that enables attackers to trick servers into disclosing data stored in memory [20]. SSL requires a heartbeat extension to maintain a TLS session. This extension allows clients to send heartbeat requests to the server, consisting of a payload and its length. The server stores the payload content in its memory and responds with it based on the specified payload length. Given that the server blindly allocates memory for the response without verifying the payload size, an attacker can craft a malicious heartbeat request with a length larger than the actual payload length and cause the server to reply with additional data stored in memory. Unfortunately, this attack neither produces a high number of syscalls, nor requests abnormal kernel features, and nor maliciously accesses sensitive files, making our approach and related works generate identical encoding for both normal and attack sequences, which ultimately results in random classifications.

Background knowledge attacks. Our solution can be vulnerable to attacks where adversaries know the approach. To illustrate, assume that an attacker wants to replace the `passwd` file in `/etc` with a crafted file located in `/tmp`. Assume further that the `rename` syscall is previously unseen and both `link` and `unlink` syscalls are previously seen. Armed with the knowledge of previously seen syscalls, an attacker may use the following syscalls: `unlink('/etc/passwd'),link('/tmp/crafted','/etc/passwd')` to perform the task instead of executing `rename('/tmp/ crafted', '/etc/passwd')`. Here, the attacker leverages `unlink` to delete the file from the filesystem and then uses `link` to copy the crafted file to `/etc`. Since the operation does not trigger any unseen syscalls and our approach does not consider arguments of both `link` and `unlink`, the replacement of the `passwd` file will pass undetected. Although our solution can be tricked by seen equivalent syscalls, we believe the operations of security-critical syscalls such as `clone`, `execve`, and `mount` cannot be performed by alternative syscalls, thus making sensitive activities likely detectable by our approach. Furthermore, our solution can also be theoretically prone to race condition attacks (i.e., TOCTOU [49]). Given that the value of a syscall argument is supplied by a pointer (e.g., *file path*), in the time between when the syscall handler consumes the pointer and when Sysdig extracts the value for monitoring, an attacker could replace the pointee with a "previously seen" argument to circumvent detection. Last, attackers can also leverage adversarial machine learning techniques to attack the auto-encoder, causing the model to yield false classifications and ultimately bypass detection [51]. We will investigate the feasibility of these attacks in future work.

Real-world container datasets. We merely test our approach against simulated attack scenarios. Although both LID-DS and CB-DS datasets involve different container attacks, they do not include real-world attacks data. Thus, we argue that realistic container datasets are required to fully assess the effectiveness of our approach in practical cloud settings. Unfortunately, this is unachievable at this time as there is no publicly available datasets.

8 CONCLUSION

This paper presents an anomaly-based intrusion detection technique for monitoring containers using syscalls. Unlike existing solutions, our work leverages a graph-based model to analyze different syscall properties within their context, enabling us to uncover intrinsic activities manifested by anomalies. Given the performance results achieved on various attack scenarios, and according to the comparative study we have carried out, we showed that our approach effectively detects container attacks with few false alarms and with reasonable processing overhead, hence outperforming existing tools. Also, despite the discussed limitations, our approach can still be practical in realistic cloud settings. Precisely, our HIDS can efficiently monitor containers for previously unseen attacks and also serve as a starting point for additional activities such as forensics, triaging, or incident response.

ACKNOWLEDGMENTS

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. Part of this work has received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement No. 830927. Any views, results, findings, or recommendations communicated in this material are those of the authors or originators and do not necessarily reflect the sponsors' standpoints.

REFERENCES

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] Amr S Abed, Charles Clancy, and David S Levy. 2015. Intrusion detection system for applications using linux containers. In *International Workshop on Security and Trust Management*. Springer, 123–135.

[3] Rancher Admin. 2020. Runtime Security in Rancher with Falco.

[4] Aqua. 2021. Cloud Native Threat Report: Evolution of Attacks in the Wild on Container Infrastructure.

[5] Jason Avery. 2018. CVE-2022-0847: "Dirty Pipe" Linux Local Privilege Escalation.

[6] Stephen P Borgatti. 2005. Centrality and network flow. *Social networks* (2005).

[7] Stephen P Borgatti and Martin G Everett. 2006. A graph-theoretic perspective on centrality. *Social networks* 28, 4 (2006), 466–484.

[8] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*. 167–167.

[9] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*. 139–151.

[10] Eric Carter, Ferenc Hámori, Steven J Vaughan-Nichols, Kalyan Ramanathan, Diego Ongaro, John Ousterhout, Abhishek Verma, Luis Pedrosa, Madhukar R Korupolu, David Oppenheimer, et al. 2019. Sysdig 2019 container usage report: New kubernetes and security insights.

[11] Stefano Chierici. 2019. How to detect the containers' escape capabilities with Falco.

[12] Stefano Chierici. 2022. CVE-2022-0492: Privilege escalation vulnerability causing container escape.

[13] Francois Chollet et al. 2015. *Keras*. https://github.com/fchollet/keras

[14] CNCF. 2020. CNCF SURVEY 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[15] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To docker or not to docker: A security perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

[16] Gideon Creech and Jiankun Hu. 2013. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Trans. Comput.* 63, 4 (2013), 807–819.

[17] National Vulnerability Database. 2012. CVE-2012-2122 Detail.

[18] National Vulnerability Database. 2018. CVE-2018-3760 Detail.

[19] Loris Degioanni. 2014. Interpreting Sysdig Output.

[20] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.

[21] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 120–128.

[22] Geeksforgeeks. 2019. Introduction of syscall.

[23] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 443–458.

[24] Martin Grimmer, Martin Max Röhling, D Kreusel, and Simon Ganz. 2019. A modern and sophisticated host based intrusion detection data set. *IT-Sicherheit als Voraussetzung für eine erfolgreiche Digitalisierung* (2019), 135–145.

[25] Andreas Gruenbacher and Seth Arnold. 2007. AppArmor technical documentation.

[26] Carlos Polop HackTricks. 2022. Docker Breakout / Privilege Escalation.

[27] Serge E Hallyn and Andrew G Morgan. 2008. Linux capabilities: making them work. (2008).

[28] Xuan Dau Hoang, Jiankun Hu, and Peter Bertok. 2003. A multi-layer model for anomaly intrusion detection using program sequences of system calls. In *Proc. 11th IEEE Int'l. Conf.* Citeseer.

[29] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. *Journal of computer security* 6, 3 (1998).

[30] Joab Jackson. 2016. Q&A James Turnbull: The Art of Monitoring in the Age of Microservices.

[31] Keshani Jayasinghe and Guhanathan Poravi. 2020. A survey of attack instances of cryptojacking targeting cloud infrastructure. In *Proceedings of the 2020 2nd Asia pacific information technology conference*. 100–107.

[32] VVRPV Jyothsna, Rama Prasad, and K Munivara Prasad. 2011. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications* 28, 7 (2011), 26–35.

[33] Dae-Ki Kang, Doug Fuller, and Vasant Honavar. 2005. 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.*

[34] David Karns, Katy Protin, and Justin Wolf. 2012. *iSSH v. Auditd: Intrusion Detection in High Performance Computing*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[35] Max Kellermann. 2022. The Dirty Pipe Vulnerability.

[36] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. 2003. On the detection of anomalous system call arguments. In *European Symposium on Research in Computer Security*. Springer, 326–343.

[37] Vinod Kumar and Om Prakash Sangwan. 2012. Signature based intrusion detection system using SNORT. *International Journal of Computer Applications & Information Technology* 1, 3 (2012), 35–41.

[38] Wenke Lee and Salvatore Stolfo. 1998. Data mining approaches for intrusion detection. (1998).

[39] Yuhang Lin, Olufogorehan Tunde-Onadele, and Xiaohui Gu. 2020. Cdl: Classified distributed learning for detecting security attacks in containerized applications. In *Annual Computer Security Applications Conference*. 179–188.

[40] Ming Liu, Zhi Xue, Xianghua Xu, Changmin Zhong, and Jinjun Chen. 2018. Host-based intrusion detection system with system calls: Review and future trends. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–36.

[41] Mairi MacLeod. 2021. Escaping from a Virtualised Environment: An Evaluation of Container Breakout Techniques. (2021).

[42] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. 2015. Security hardening of Linux containers and their workloads. (2015).

[43] HD Moore. 2012. CVE-2012-2122: A Tragically Comedic Security Flaw in MySQL.

[44] Anna Mpanti, Stavros D Nikolopoulos, and Iosif Polenakis. 2018. Malicious Software Detection and Classification utilizing Temporal-Graphs of System-call Group Relations. *arXiv preprint arXiv:1812.10748* (2018).

[45] Sarang Narkhede. 2018. Understanding auc-roc curve. *Towards Data Science* 26, 1 (2018), 220–227.

[46] NetworkX. 2022. Graph.size.

[47] Marcus Pendleton. 2017. *System Call Anomaly Detection in Multi-threaded Programs*. Ph. D. Dissertation. UNIVERSITY OF TEXAS AT SAN ANTONIO.

[48] James Powell. 2015. *A librarian's guide to graphs, data and the semantic web*. Elsevier.

[49] Razvan Raducu, Ricardo J Rodríguez, and Pedro Álvarez. 2022. Defense and Attack Techniques against File-based TOCTOU Vulnerabilities: a Systematic Review. *IEEE Access* (2022).

[50] Michael J Reeves. 2021. *INVESTIGATING ESCAPE VULNERABILITIES IN CONTAINER RUNTIMES*. Ph. D. Dissertation. Purdue University Graduate School.

[51] Elsa Riachi and Frank Rudzicz. 2020. Understanding Adversarial Attacks on Autoencoders. (2020).

[52] Rami Rosen. 2016. Namespaces and cgroups, the basis of Linux containers. *Seville, Spain, Feb* (2016).

[53] Areeg Samir and Claus Pahl. 2020. Detecting and localizing anomalies in container clusters using Markov models. *Electronics* 9, 1 (2020), 64.

[54] Rafath Samrin and D Vasumathi. 2017. Review on anomaly based network intrusion detection system. In *2017 international conference on electrical, electronics, communication, computer, and optimization techniques (ICEECCOT)*. IEEE, 141–147.

[55] Rushank Shetty, Kim-Kwang Raymond Choo, and Robert Kaufman. 2017. Shellshock vulnerability exploitation and mitigation: a demonstration. In *International Conference on Applications and Techniques in Cyber Security and Intelligence*. Springer, 338–350.

[56] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[57] Roopak Surendran and Tony Thomas. 2022. Detection of malware applications from centrality measures of syscall graph. *Concurrency and Computation: Practice and Experience* 34, 10 (2022), e6835.

[58] Sysdig. 2017. Secure DevOps Platform. https://github.com/draios/sysdig.

[59] Sysdig. 2021. Sysdig 2021 Container Security and Usage Report.

[60] Sysdig. 2022. Falco: Open Source Security Tool for containers, Kubernetes and Cloud.

[61] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. 2019. Kubanomaly: anomaly detection for the docker orchestration platform with neural network approaches. *Engineering reports* 1, 5 (2019), e12080.

[62] Wei Wang, Yan Huang, Yizhou Wang, and Liang Wang. 2014. Generalized autoencoder: A neural network framework for dimensionality reduction. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 490–497.

[63] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*. IEEE, 133–145.

[64] Yanjun Wen and Ji Wang. 2019. Analysis and Remodeling of the DirtyCOW Vulnerability by Debugging and Abstraction. In *International Workshop on Structured Object-Oriented Formal Language and Method*. Springer, 3–12.

[65] Fei Yu, Cheng Xu, Yue Shen, Ji-yao An, and Lin-feng Zhang. 2005. Intrusion detection based on system call finite-state automation machine. In *2005 IEEE International Conference on Industrial Technology*. IEEE, 63–68.

[66] Zhuping Zou, Yulai Xie, Kai Huang, Gongming Xu, Dan Feng, and Darrell Long. 2019. A docker container anomaly monitoring system based on optimized isolation forest. *IEEE Transactions on Cloud Computing* (2019).