REPLICAWATCHER: Training-less Anomaly Detection in Containerized Microservices

Asbat El Khairi
University of Twente
a.elkhairi@utwente.nl

Marco Caselli
Siemens AG
marco.caselli@siemens.com

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

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

***Abstract*—Despite its detection capabilities against previously unseen threats, anomaly detection suffers from critical limitations, which often prevent its deployment in real-world settings. In fact, anomaly-based intrusion detection systems rely on comprehensive pre-established baselines for effectively identifying suspicious activities. Unfortunately, prior research showed that these baselines age and gradually lose their effectiveness over time, especially in dynamic deployments such as microservices-based environments, where the concept of "normality" is frequently redefined due to shifting operational conditions. This scenario reinforces the need for periodic retraining to uphold optimal performance — a process that proves challenging, particularly in the context of security applications.**

We propose a novel, training-less approach to monitoring microservices-based environments. Our system, REPLICAWATCHER, observes the behavior of identical container instances (i.e., *replicas*) and detects anomalies *without* requiring prior training. Our key insight is that replicas, adopted for fault tolerance or scalability reasons, execute analogous tasks and exhibit similar behavioral patterns, which allow anomalous containers to stand out as a notable deviation from their corresponding replicas, thereby serving as a crucial indicator of security threats. The results of our experimental evaluation show that our approach is resilient against normality shifts and maintains its effectiveness without the necessity for retraining. Besides, despite not relying on a training phase, REPLICAWATCHER performs comparably to state-of-the-art, training-based solutions, achieving an average precision of 91.08% and recall of 98.35%.

I. INTRODUCTION

The landscape of software development has seen a significant paradigm shift, with enterprises embracing the microservice approach for developing complex applications [8]. This approach's ubiquity stems from its ability to break down a complex application into dozens of distinct and scalable services instead of a single monolithic stack, thus providing a flexible software development process [31]. In this context, container technology has emerged as a perfect companion to microservices [44]. Indeed, platforms such as Docker [38] and Kubernetes [30] offer a powerful infrastructure that simplifies deployment and management of microservices. This is evidenced in a recent survey [9], which reported that 83% of respondents use Kubernetes in production environments.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24286
www.ndss-symposium.org

Nevertheless, with container adoption soaring, attackers are shifting their direction to this technology, causing massive impairment of business performance [36]. To account for this, many enterprises concede that the *shift security left* [54] approach, despite preventing security breaches from being deployed, is not exhaustive. Zero-day attacks can still manifest at runtime [2], allowing attackers to compromise critical applications. Such threats mandate a *shield-right* security approach [52], which emphasizes continuous monitoring to detect and alert security teams of suspicious activities.

The idea of monitoring containers for anomalies has been explored in both industry and academia. Falco [53], a monitoring tool with container support, stands out as a leading solution in industry [54]. This tool enforces stringent security rules at the kernel level, allowing for good coverage of suspicious activities. In academia, various studies have been centered on the analysis of system calls (syscalls) to detect anomalies. One noteworthy approach [1] utilizes the frequency of syscalls within a sliding window to define container behavior as n-gram syscall sequences, and it then employs a mismatch-based threshold to detect anomalies. Another work [33] leverages machine learning techniques to model the frequency of syscalls in short time-based sequences. In a different approach, a recent work [14] combines machine learning and graph modeling to analyze the context around various syscall properties (e.g., frequency, arguments) to unveil abnormal behavior.

However, despite their detection capabilities against previously unseen threats, all approaches that build upon anomaly-based detection suffer from a critical limitation: the dependence on a training baseline. Traditional anomaly-detection assumes that such a baseline model provides an accurate, steady, and comprehensive representation of normal behavior. Unfortunately, this assumption does not hold in *real-world* settings [21], especially in dynamic microservices-based environments. In fact, the evolving nature of microservices [54], often in response to new requirements, features or vulnerabilities, causes frequent drifts in the definition of "normal". Failure to identify and adapt to such drifts leads to an unmanageable amount of false positives [21]. This calls for periodic retraining to uphold performance, a process that is proved challenging within security domains [37], [60], [63].

In contrast to existing solutions, we propose a novel approach to container anomaly detection that *does not require a predefined training baseline*. Instead, we base our solution on the key idea of comparing replicas, i.e., identical instances of a container replicated for scalability reasons. Our intuition is that replicas behave similarly as they execute analogous

tasks. On the contrary, when subject to an attack, replicas show inconsistency in their behavior, enabling us to detect suspicious activity in a training-less fashion.

While intuitively this approach shows the great advantage of not relying on a predetermined baseline model, training-less anomaly detection proves feasible only when it can effectively manage 1) the inherent *background noise* produced by replicas, while 2) guaranteeing a minimal *processing overhead* in operational settings. We address these challenges, which make training-less anomaly detection non-trivial.

Background noise. While replicas are designed to perform the same function, they inevitably generate a certain level of noise due to normal operational inconsistencies (e.g., varying user interactions, load fluctuations, etc). This poses a significant challenge in the selection of decision features to detect anomalies. Besides carrying strong and diverse security semantics that assist in generic anomaly detection, the selected features need to demonstrate resilience against the inherent noisy patterns of replicas, mitigating the risk of false alerts.

Processing overhead. In a microservices-based setup that involves replicas, we need to select features that not only show effectiveness and resilience, but also promote computational efficiency. The features should warrant that their extraction and processing do not incur undue computational burdens, thus balancing performance and resource utilization.

To handle these challenges, we develop REPLICAWATCHER, a novel training-less approach to monitor containerized clusters for anomalies. *One good trait of replicas lies in their execution of functionally identical tasks, resulting in highly congruent behavioral patterns* [41]. Capitalizing on this inherent property, we uncover anomalies by comparing the behavior of replicas. On a high-level, we first collect kernel events generated by replicas, a process designed to impose *minimal overhead*. Following this, we proceed to extract pertinent features from these events, identifying those features that are immune against *background noise* and adept at flagging various anomalies. We compare these features across replicas employing a similarity-based technique to identify dissimilar instances. Finally, we classify replicas based on the degree of observed dissimilarity, identifying suspicious activity.

In summary, we make the following contributions:

- We propose the concept of *training-less anomaly detection* for identifying anomalies *without* requiring an initial training and subsequent retraining(s).
- We present REPLICAWATCHER, a training-less anomaly-based detection system specifically designed to identify anomalies in microservice-based environments.
- We implement and evaluate REPLICAWATCHER. Our results show that our approach is resilient against normality drifts without requiring any retraining and maintaining a negligible runtime overhead. We also show that our system performs comparably to state-of-the-art training-based techniques, achieving an average precision of 91.08% and recall of 98.35% on 13 attack scenarios across two microservices-based platforms.

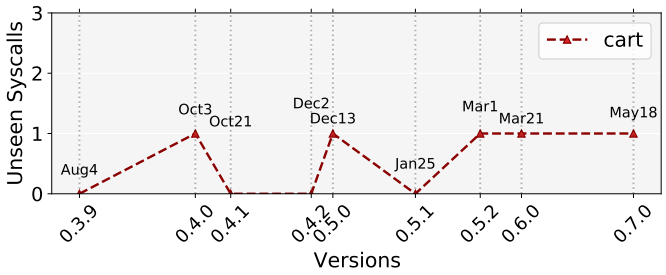


Fig. 1: **Unseen syscalls count at each *cart* update.**

```
cart0.6.0
PROC TID >/< Syscall
grpc_h_p (308) > fcntl
grpc_h_p (317) < futex
grpc_h_p (317) > futex
grpc_h_p (315) < futex
grpc_h_p (315) > switch
```

```
cart0.7.0
PROC TID >/< Syscall
grpc_h_p (273) < fcntl
grpc_h_p (290) > futex
grpc_h_p (273) > getrlimit
grpc_h_p (290) < futex
grpc_h_p (273) < getrlimit
grpc_h_p (290) > futex
grpc_h_p (290) > switch
```

Fig. 2: **Syscall normality shift between *cart-0.6.0* and *cart-0.7.0*.**

In the spirit of open science, we make our tool available at https://github.com/utwente-scs/Replicawatcher.

II. MOTIVATION

This section presents a concrete example of a *normality drift* scenario, a critical factor that underpins the essence of our research. In microservices-based environments, changes in the base OS images, dependency package updates, or deployment of new functionalities are common events that cause drifts in normality—e.g., Netflix makes hundreds of production changes per day [61]. When considering syscall-based detection systems, such updates might lead to the execution of previously unseen syscalls, triggering false positives.

To demonstrate this potential drift, we conducted an empirical study using *Google Online Boutique (GOB)* [19], a microservices-based application that simulates an e-commerce platform. It comprises of 11 different microservices, including checkout, payment, cart, and others. These services work together to provide users with the ability to browse items, add them to a cart, and complete purchases. We deployed the application in Google Kubernetes Engine (GKE) [1] and monitored the *cart* microservice across its last ten versions over the span of nine months. We executed each version for three hours. During this time, we recorded all the syscalls that were executed by a new version but not by the preceding one.

Figure 1 shows that almost each new version of the *cart* microservice generates one previously unseen syscall, possibly due to changes to the underlying base operating system (OS) images, packages, or application code. Indeed, during the microservice upgrade from *cart:0.6.0* to *cart:0.7.0*, the `.NET SDK` was updated from 7.0.201 to 7.0.302, and `GRPC_HEALTH_PROBE` from 0.4.15 to 0.4.18. These updates introduced the execution of the `getrlimit` syscall, as shown in Figure 2. This syscall is responsible for obtaining the

[1] https://cloud.google.com/kubernetes-engine/

resource limits of a given process or thread [35]. Although the new version may employ `getrlimit` due to modifications in system resource handling or the implementation of new features, its previously unseen nature inadvertently triggers false positives in existing container-based detection systems, as we show in detail in Section VI. Our work fills this gap: We develop a detection approach based on comparing the runtime behavior of replicas, which typically run the same and updated version of container images. Doing so, our solution automatically adapts to the evolving nature of microservices, ensuring its robustness and reliability over time, while eliminating the need for training and retraining.

We stress that the example we discussed in this section is based on a simple yet real-world application. We also note that this is not just an isolated example: our experiments and case studies, discussed at length in Section VI, show that different dynamic factors affect the normal behavior of microservices and that existing approaches are unable to deal with such changes, leading to substantial performance degradation.

III. Preliminary Assessment

Building a training-less anomaly-based IDS calls for two key considerations: (❶) We need to select features that cover different aspects of container behavior and can generically detect attacks. (❷) We need to identify, among those features, the ones that exhibit less noise among replicas in normal mode to prevent raising false alarms. To this end, we conducted a preliminary assessment. This process began with exploring potential features, followed by selecting those most suitable for our training-less detection approach.

A. Preliminary Setup

We deploy each microservice of GOB with four replicas. To collect logs, we leverage the *loadgenerator* service to automatically interact with the application under various settings (i.e., traffic volume, waiting times, and input sizes) to produce realistic workloads that reflect user activity. We use Sysdig [49] with a customized *Chisel* [5] (i.e., a *lua* script that extends Sysdig's functionality) to collect kernel events at four distinct time intervals, namely 5s, 10s, 30s, and 60s. For each microservice, we generate a unique set of 1,000 logs per interval. Each log serves as a record, encapsulating all events (e.g., executed syscalls, accessed files, etc) performed by the replicas within that specific interval. In total, we collect 44,000 logs, which approximates to 13 days of recording.

B. Feature Exploration

(❶) To effectively address the first point, we select a set of features that includes both commonly-used and novel features related to three fundamental elements of the operating system (OS) kernel: syscalls, file descriptors, and processes. In fact, these elements are critical for the execution of any malicious task and capture artifacts left by attackers. Appendix E provides a detailed description of the studied features.

Syscalls. The primary way programs interact with the OS kernel for tasks like accessing the disk, working with files, establishing network connections, and managing processes is via syscalls [16]. We focus on several syscall attributes to monitor potential security threats, including the frequency of executed and failed syscalls, executed syscall names and their categories (e.g., *network* for `socket`, `bind`), and syscall latency and delta time.

File Descriptors (FD). Unix abstracts several resources as files, including I/O, pipes, signals, and sockets, and it assigns non-negative integers as file descriptors when opened [24]. We leverage this to tap into network and filesystem activities, observing the frequency of accessed files, distinct accessed directories and filenames (focusing here on shorter file paths limited to three subdirectories or fewer, and devoid of random elements like cache strings), the buffer size for specific syscalls (e.g., `read`, `recvfrom`), and client IP addresses and ports.

Processes. A container can run a diverse range of processes, spanning from its primary application to essential supporting operations and intricate system interactions. To obtain detailed insights into the behavior of replicas, we monitor several aspects: executed processes, commands, and their arguments, executables, and current working directories.

C. Feature Engineering

(❷) To address the second consideration, we analyze the behavior of replicas under normal conditions. Within each interval, we employ two techniques to evaluate the dissimilarity among features. For list-generating features, such as distinct list of accessed directories, we first compute the average Jaccard similarity of the lists, which are produced by all replicas for a given feature f_k, and then subtracting it from 1. The average Jaccard similarity of n replicas is computed by finding the mean of the Jaccard similarity across all possible replicas pairings $(R_{i_{f_k}}, R_{j_{f_k}})$, as illustrated in Equation 1. The dissimilarity value spans from 0 to 1. A zero value means absolute alignment, implying that all replicas generate identical lists. Conversely, a value of 1 indicates total discrepancy, suggesting the absence of any common elements in the produced lists. For features that generate individual values, such as the max latency, we use the standard deviation to measure their dissimilarity. This is calculated as the square root of the average of the squared differences between each replica's value $R_{i_{f_s}}$ and the mean [22], as shown in Equation 2. Given that the standard deviation is naturally higher for metrics with large values, such as the frequency of syscalls, we perform min-max normalization between replicas to scale the values into a range between 0 and 1, allowing for a reliable measurement.

$$Jacc_{avg} = \frac{1}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{|R_{i_{f_k}} \cap R_{j_{f_k}}|}{|R_{i_{f_k}} \cup R_{j_{f_k}}|} \tag{1}$$

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (R_{i_{f_s}} - \bar{x})^2} \tag{2}$$

D. Feature Evaluation

We analyze and rank features based on their maximum dissimilarity observed across the logs of each interval. Our findings, shown in Table I, reveal that features such as syscall

frequency, network IP addresses and ports, latency, delta time, and buffer length exhibit substantial dissimilarity across replicas. This can be attributed to factors such as varying workloads, network conditions, and user behavior. Conversely, all process-based features manifest little to no dissimilarity, primarily due to their inherent association with the applications or processes executed within replicas. As a result, they are less susceptible to variability.

Additionally, we note that longer monitoring intervals result in enhanced similarity across replicas (Table I). This can be attributed to three primary factors: the broadened perspective provided by extended intervals, the potential synchronization of environmental conditions over time, and the likelihood of stabilized resource utilization and user behavior. Armed with these observations, we opt for a 30-second monitoring interval and features with maximum dissimilarity scores less than 0.2. It is important to note that our preliminary assessment focuses exclusively on the normal behavior of *GOB* microservices. Nevertheless, our results, discussed in Section VI, indicate that the selected features not only show immunity against normal noise, but also possess sensitivity to a wide array of attacks, making them applicable in diverse threat scenarios.

TABLE I: MAXIMUM DISSIMILARITY VALUES OF THE STUDIED FEATURES ACROSS REPLICAS. THE FEATURES WE SELECT ARE HIGHLIGHTED IN BOLD.

Category	Feature	5s	10s	30s	60s
Syscall	frequency	0.533	0.553	0.524	0.549
	frequency (failed)	0.557	0.577	0.528	0.577
	type	0.501	0.331	0.131	0.102
	category	0.227	0.058	0.021	0.018
	max latency	0.552	0.556	0.508	0.546
	max delta time	0.557	0.551	0.571	0.553
File Descriptor	frequency	0.566	0.561	0.488	0.512
	directory	1.000	0.410	0.025	0.022
	filename	1.000	0.059	0.032	0.026
	file operation	0.333	0.000	0.000	0.000
	max buffer length	0.555	0.571	0.459	0.561
	client ip	0.502	0.251	0.211	0.205
	client port	1.000	1.000	1.000	1.000
Process	**proc**	0.333	0.166	0.107	0.107
	cmdline	0.333	0.166	0.107	0.107
	cwd	0.000	0.000	0.000	0.000
	executable	0.333	0.166	0.000	0.000
	args	0.333	0.166	0.000	0.000

IV. THREAT MODEL

We consider a Kubernetes cluster with a set of *deployments* [2], each running a heterogeneous set of replicas. We assume that these replicas execute specific tasks, adhering to the principle of microservice design, where each service is responsible for a specific area of concern. We further consider that all replicas are configured to run the same version of the code and that any updates are rolled out to all replicas simultaneously (e.g., through *Deployment Controllers*). This a practical standard in microservices-based environments, where replicas are built from a common configuration repository (e.g., Dockerfile). We also assume that all replicas consistently receive incoming requests, representing active user interactions. This aligns with the scenario of high-traffic platforms (e.g., Netflix [56], eBay [43]), where each replica within a microservice is engaged in continual user interactions. In fact, replicas are meant for scalability. An idle replica is an unnecessary resource that orchestrators (e.g, Kubernetes) automatically shut down.

We focus on remote network-triggered attacks where an adversary exploits a vulnerability in a public-facing microservice, delivering malicious payloads via TCP/UDP. Specifically, we focus on software-level, remote control flow hijacking attacks. Our scope consists of scenarios where an attacker targets replica(s) to access sensitive data (e.g., `authentication bypass`, `directory traversal`), compromise the filesystem's integrity (e.g., `file inclusion`), or execute unauthorized code/commands (e.g., `command injection`). We assume that such actions deviate the behavior of replica(s) compared to their counterparts. We also assume that attackers cannot initiate an *identical* attack on *all* replicas, *simultaneously*. While such an attack is theoretically possible, its practical execution is challenging and requires an attacker to have full knowledge of the target system. Furthermore, such a massive attack would unavoidably leave traces that approaches orthogonal to ours can spot, e.g., via network traffic. Nonetheless, we further discuss this scenario, as well as potential methods to address it, in-depth in Section VIII. Besides, we do not consider attacks that bypass or remain inconspicuous to the syscall interface, as well as side-channels, hardware-related attacks (e.g., spectre), resource exhaustion, network flooding, remote passive fingerprinting, spoofing attacks, and attacks that require physical or internal access to the monitored Kubernetes cluster.

Finally, our system is an anomaly-based IDS, focusing on monitoring replicas for anomalies without prior training. Our solution leverages Sysdig [49] as a privileged *daemonset* [3] to cover all worker nodes within a Kubernetes cluster. Despite the possibility of attackers tampering with Sysdig to conceal their activities, they would first need to escape the container namespace and elevate privileges on the underlying host OS to do so. While container escape is an important subject, it is out of scope for this work. Besides, existing tamper-evident monitoring approaches [42] alleviate this assumption.

V. APPROACH

We propose REPLICAWATCHER, a training-less solution to monitor containers for anomalous behavior. We base our approach on the idea that replicas execute analogous tasks, thus normally manifesting comparable behavioral patterns. On the contrary, replicas manifest inconsistent behavior when they are targeted by attacks. Our focus centers on analyzing and comparing such patterns to spot potentially malicious activity.

Figure 3 depicts our system design. REPLICAWATCHER consists of three phases: event chunking, event encoding, and anomaly detection. In event chunking (Ⓐ), we process the continuous stream of events from all running replicas into short monitoring intervals, referred to as *snapshots*. These snapshots allow for an efficient comparison of replicas within particular timeframes, thus facilitating the timely detection of anomalous patterns. In event encoding (Ⓑ), we construct a feature vector from each replica's events, capturing the degree of difference

[2]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[3]https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

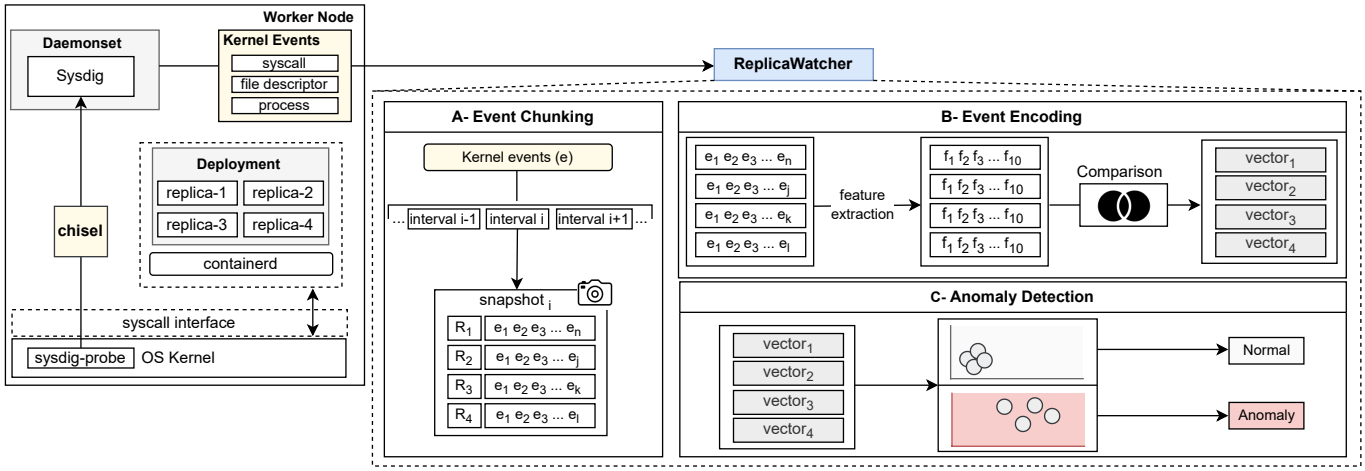


Fig. 3: **REPLICAWATCHER Overview.** (A) We split the continuous flow of kernel events into short monitoring intervals, defined as *snapshots*. (B) Within each snapshot, we generate a vector for every replica, encapsulating its relative dissimilarity in comparison to its counterparts. Then, (C) we input these vectors into a detection framework that, through their distance from the *normality region*, classifies snapshots.

that each replica exhibits when compared to the others. In anomaly detection (C), we feed these vectors to a detection scheme, which evaluates each vector's deviation from the ideal state, i.e., where all replicas are in complete alignment. If any replica exhibits a marked deviation, we trigger an alert, and the corresponding snapshot is accordingly labeled as anomalous.

Note that we compare together replicas related to the same microservice. Thus, in practice, we apply our approach to multiple sets of replicas, e.g., one set of replicas for the login service, one set of replicas for the user profile service, etc.

A. Event Chunking

We configure Sysdig [49] with a customized *Chisel* [5] to capture granular interactions between replicas and their underlying OS kernel. This involves kernel events associated with the filesystem (e.g., accessed files), processes (e.g., executed commands), as well as syscalls (e.g., executed syscalls). The first step of our approach is to split the ongoing flow of kernel events into sequential *snapshots*, with each describing the activities of replicas for a designated time interval $\tau_{snapshot}$. In the determination of $\tau_{snapshot}$, we consider the trade-off between performance and attack lead time (i.e., elapsed time between the detection of an attack and its completion). A shorter interval affords greater attack lead time and enables prompt response to security incidents, however, it also increases the risk of false alarms triggered by inconsistencies among replicas. Conversely, a longer interval reduces the number of false alarms through a more robust understanding of replicas, yet at the cost of sacrificing some attack lead time. Therefore, following our preliminary assessment (Table I), we experimentally set the snapshot interval $\tau_{snapshot}$ to 30 seconds, which provides actionable insights about replicas while still enabling timely detection of anomalies.

B. Events Encoding

At this stage, we first extract key features from kernel events of each replica. We select these features based on our early assessment (Section III). Our primary aim is to compare behavioral patterns of replicas to identify those replicas that exhibit a considerable degree of dissimilarity, indicating suspicious activity. To this end, we represent each replica by a vector of dissimilarity scores, each score corresponds to a distinct feature. Our features, shown in Table I (highlighted in bold), are inherently list-based (i.e., syscalls). For a given replica R_i and a particular feature f_k, we leverage the Jaccard similarity technique to calculate the dissimilarity score $DS_{f_k}(R_i)$, as shown in Equation 3. Specifically, we subtract the average Jaccard similarity between feature f_k in R_i and the corresponding feature in other replicas from 1. A high dissimilarity score indicates greater uniqueness in the elements encompassed by the feature, implying a higher likelihood of the replica being anomalous. Conversely, a low dissimilarity score suggests that the elements within the feature are more commonly observed across other replicas, lowering the likelihood of the replica being anomalous.

$$DS_{f_k}(R_i) = 1 - \frac{1}{n-1} \sum_{j=1 \ \ j \neq i}^{n} \frac{|R_{i_{f_k}} \cap R_{j_{f_k}}|}{|R_{i_{f_k}} \cup R_{j_{f_k}}|} \qquad (3)$$

By applying this process to each feature, we generate a dissimilarity score vector for every replica, denoted as $v_i = [DS_{f1}(R_i), DS_{f2}(R_i), ..., DS_{f_k}(R_i)]$. This vector serves as a concise and meaningful representation of a replica's degree of alignment with its counterparts, and can be used as input to our decision engine to detect anomalies.

C. Anomaly Detection

The underlying premise of our detection framework is that replicas, when operating normally, display minor behavioral differences. As a result, we expect their dissimilarity vectors to converge towards the origin of the vector space, represented by $[0, 0, .., 0, 0]$. The origin serves as a reliable *normality region*. Specifically, replicas residing near the origin suggest alignment in behavior, while those deviating indicate potential misalignment and, thus, abnormal behavior.

TABLE II: ATTACK SCENARIOS. (*) DENOTES GOOGLE ONLINE BOUTIQUE (GOB) SCENARIOS.

Threat Impact	CVE/CWE	Microservice
Authentication Bypass	CWE-89	LOGIN
	CWE-307	LOGIN
Information Disclosure	CVE-2019-5418	CHECKOUT
	CVE-2018-3760	CHECKOUT
	CVE-2017-14849 *	PAYMENT *
Remote Code Execution	CVE-2017-12636	PRODUCTDB
	CVE-2022-24706	PRODUCTDB
	File Inclusion	USER PROFILE
	CWE-502 *	RECOMMENDATION *
Command Injection	CWE-434	USER PROFILE
	CVE-2012-1823	LOGIN
	CVE-2018-19518	SIGNUP
	CVE-2014-6271	CART
	CWE-78 *	SHIPPING *
Privilege Escalation	CVE-2017-12635	PRODUCTDB

Leveraging this understanding, we calculate the Euclidean distance [32] between each replica's dissimilarity vector and the origin. This distance plays a critical role in controlling the trade-off between the detection and false alarm rate. We evaluate a vector's distance at multiple threshold values, ranging from 0.1 to 1 with a 0.1 increment. If the distance exceeds the employed threshold, we label the snapshot as anomalous. Intuitively, a lower threshold can improve the detection rate, but may also increase false alarms. We delve into this trade-off through experiments presented in Section VI.

VI. EVALUATION

We implemented REPLICAWATCHER in Python [57] and evaluated its performance against various attack scenarios. First, we evaluated the resilience of REPLICAWATCHER against normality drifts induced by image updates. Additionally, we compared our approach against state-of-the-art, training-based container HIDSes [1], [14], [33], both in terms of detection capabilities and resilience against normality drifts. Finally, we delved into how our approach sustains its performance at scale and retains minimal runtime overhead.

A. Dataset & Experimental Setup

Evaluation dataset. We developed an e-commerce application with seven microservices, such as *checkout* and *cart*, mirroring real-world platforms. We used different technologies for each service, showcasing the tech diversity in microservices-based setups. We intentionally embedded vulnerabilities from common software libraries, presenting vectors for container attacks. Furthermore, our evaluation also includes the GOB e-commerce platform, where we injected three vulnerabilities covering three attack scenarios, each targeting a different microservice and leading to different threat impacts. Table II outlines attack scenarios across our "homebrew" and GOB platforms. For more details, see Appendices C and D.

Experimental setup. We deployed the target applications with replicas on a two-node *GKE* cluster. Each node is an *e2-standard-4 (4vCPU, 16GB memory)* machine with *ubuntu* and *containerd*. Our experiment involves normal and attack modes and varies the replicas per microservice from two to six. In the normal mode, 30 to 50 users engage with the application, exhibiting "benign" usage patterns. In the attack mode, alongside regular users, a distinct user acts as an attacker, exploiting vulnerabilities or misconfigurations to compromise replica(s) using malicious payloads.

TABLE III: EVALUATION DATASET STRUCTURE.

	No. replicas				
Snapshots	**2**	**3**	**4**	**5**	**6**
Normal	200	200	200	200	200
Attack	50	50	50	50	50

We use Sysdig [49] and a custom *chisel* [5] to capture kernel events of replicas. For each vulnerable microservice, we collect 1,000 normal and 250 attack snapshots split evenly across five settings with varying replica counts, detailed in Table III. A snapshot captures the behavior of replicas for a duration of 30 seconds. In total, our dataset has 15,000 normal and 3,750 attack snapshots, equating to six days of monitoring.

B. Existing works.

In our experiments, we compared REPLICAWATCHER with existing, training-based detection approaches. Here, we briefly describe them to provide a better understanding of their techniques, and, thus, of our results.

STIDE-BoSC [1]. The authors blend STIDE [59] and Bag-of-words [64] techniques to model syscall traces. In training, they build a database of bag of syscalls vectors (i.e., *BoSCs*) by recording each syscall within a window of size ten. In testing, they compare the encountered *BoSCs* against the training database. If the number of mismatched *BoSCs* within a certain interval surpasses a pre-set threshold, the authors classify the interval as anomalous.

CHIDS [14]. Using a graph-based structure, the authors generate vectors that encapsulate the contextual influence of unseen syscalls and arguments, along with their frequency in short sequences. In training, they train the auto-encoder network with normal vectors to reduce the reconstruction errors. In testing, the authors classify a sequence as anomalous if its reconstruction error surpasses the pre-set threshold.

CDL [33]. The authors transform syscall sequences into frequency-based vectors. Then, they train an autoencoder network with normal vectors to reduce the reconstruction errors. In testing, the authors classify a sequence as anomalous if its reconstruction error exceeds the predetermined threshold.

C. Robustness against Normality Drifts

We evaluated how robust REPLICAWATCHER is against environmental changes and normality drifts. We delved into this aspect by evaluating three update scenarios, which update the base OS image, a dependency package, and the application code, respectively. We monitored the performance of REPLICAWATCHER across these updates to assess its resilience to environmental changes, and compared its performances with the aforementioned existing tools.

Base OS image update. We update the *Cart* microservice from a version incorporating *php:8.1.18-apache-buster* to another with *php:8.1.18-apache-bullseye*. As shown in Figure 4, this update introduces one previously unseen syscall, namely `clock_nanosleep`. The introduction of this syscall is caused by a change in *glibc* [18] from 2.28 to 2.31. As shown in Figure 4, the `clock_nanosleep` syscall seems to replace `nanosleep` in the *Bullseye*-based image, offering high-resolution *sleep* functionality with a selectable clock [34].

```
php:8.1.18-apache-buster
PROC TID >/< Syscall
apache2 (246) < getpid
apache2 (246) > nanosleep
apache2 (246) > switch
apache2 (246) < nanosleep
apache2 (246) > getcwd
```

```
php:8.1.18-apache-bullseye
PROC TID >/< Syscall
apache2 (997) < getpid
apache2 (997) > clock_nanosleep
apache2 (997) > switch
apache2 (997) < clock_nanosleep
apache2 (997) > getcwd
```

Fig. 4: **Comparison of syscalls between *php-apache* versions.**

Dependency package update. We update the *Checkout* microservice, specifically upgrading the *Puma*[4] package from version 3.2 to 5.5. This upgrade introduces some previously unseen syscalls. As shown in Figure 5, the `select` syscall used in the older version is now replaced by a trio of syscalls, namely `epoll_wait`, `epoll_ctl`, and `ppoll`. Here, `epoll_wait` waits for events on the *epoll* instances, `epoll_ctl` adds, modifies, or removes file descriptors from the interest list of the *epoll* instance, and `ppoll` monitors multiple file descriptors for I/O readiness. These new syscalls enhance *Puma*'s ability to manage multiple file descriptors, thus optimizing network I/O handling.

```
------- puma 3.11 ------
PROC TID >/< Syscall
puma (198) < getpid
puma (198) < select
puma (198) < switch
puma (198) > recvfrom
puma (198) < recvfrom
```

```
------- puma 5.5 --------
PROC TID >/< Syscall
puma (213) < getpid
puma (213) < ppoll
puma (213) < switch
puma (213) <  epoll_wait
puma (213) <  epoll_ctl
puma (213) <  epoll_ctl
puma (213) <  epoll_wait
puma (213) < switch
puma (213) <  ppoll
puma (213) > recvfrom
puma (213) < recvfrom
```

Fig. 5: **Comparison of syscalls between *Puma* versions.**

Application code update. We update the *Signup* microservice, specifically adding a new function that validates the input address based on the Google's Geocoding API [5]. As shown in Figure 6, the new change in the code incorporates an HTTP network request to this API, which is used to verify the authenticity of the provided address. This modification introduce some previously unseen syscalls during the application's execution. These syscalls are associated with network operations. Specifically, the `bind` syscall assigns a local protocol address to a *socket*, preparing it for the outgoing connection to the API. Moreover, the `recvmsg` syscall is employed to receive incoming data (i.e., API validation response) from the *socket*.

Post-update performance analysis. Figure 7 shows the performance of all the evaluated approaches before and after

[4]https://puma.io/

[5]https://developers.google.com/maps/documentation/geocoding/overview

```
----- insertData.php -----
<?php
...
$_addr = urlencode($addr);
$url= $API . $_addr . $KEY;
$r=file_get_contents($url);
...
?>
```

```
-------- signup --------
PROC TID >/< Syscall
php-cgi (64) > bind
php-cgi (63) < socket
php-cgi (63) < bind
php-cgi (64) < getsockname
php-cgi (64) < sendto
php-cgi (64) > recvmsg
php-cgi (64) < recvmsg
php-cgi (64) < close
```

Fig. 6: **Syscalls of the new *Signup* version.**

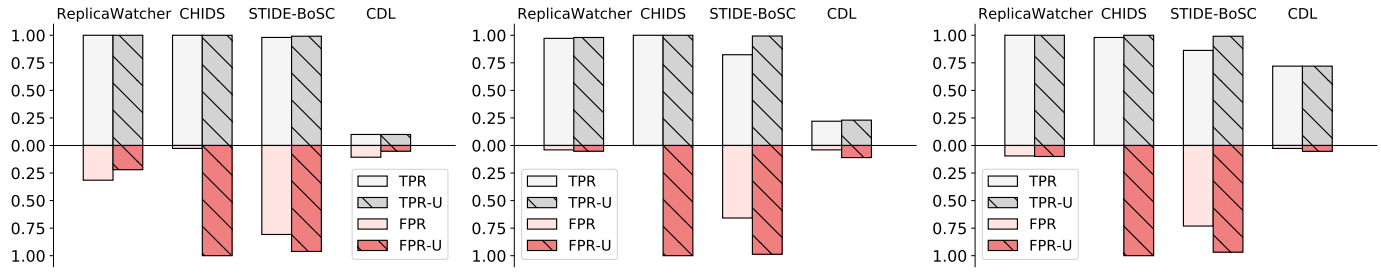
the update, for each of the three scenarios. We note a significant increase in false positives for both STIDE-BoSC and CHIDS after implementing the three updates. This high sensitivity towards updates can be attributed to the emergence of previously unseen syscalls. Pre-update versions generated a different array of syscalls, thereby shaping the trained model's perception of normal behavior. As a result, these syscalls lead both CHIDS and STIDE-BoSC to encounter challenges in accurately classifying their occurrence. More specifically, these calls trigger a high unseen syscalls influence (USI) in the feature vector of CHIDS and also cause BoSC mismatches for the STIDE-BoSC work. Consequently, this leads to a high number of false positives. Post-updates, CDL performs better and the increase of false positives is less significant. As CDL operates on a frequency-based methodology, it only flags deviations in syscall frequency rather than the syscalls themselves. This mechanism allows CDL to maintain a low false positive rate, even after updates. However, this approach also renders CDL incapable of detecting attacks, as it is primarily designed to track frequencies rather than identify anomalous or malicious syscalls. Therefore, while it performs well in terms of limiting false positives, its ability to provide generic detection is fundamentally compromised.

In contrast, REPLICAWATCHER operates on a distinctly different principle. Our underlying approach works by comparing replicas that run similar tasks. In microservices-based environments, any update is automatically propagated to all replicas, preserving the consistency of their behavior—in practice, replicas are usually executed from the same Dockerfile. As a result, REPLICAWATCHER maintains a nearly steady false positive rate, almost unaffected by system updates, and it upholds a high detection rate, making it adept at detecting attacks while handling the dynamic nature of microservices.

More fundamentally, our experiments show how training-based approaches suffer from performance degradation when the monitored environment changes. Even worse, while our scenarios focused on a single update deployment, production environment are subject to various and constant changes, further emphasising the need for an approach like ours.

D. Detection Capabilities

Figure 8 shows the performance of our approach against multiple attack scenarios spanning different microservice-based applications. For specific threats, REPLICAWATCHER yields high AUC scores of 0.9771, 0.9883, and 0.9970 for information disclosure, remote privilege escalation, and code execution scenarios, respectively. Delving deeper into the application categories, in the GOB scenarios, our system yields



(a) Base OS Image (b) Dependency Package (c) Application Code

Fig. 7: **Assessment of Post-Update Performance.** (-U) indicates post-update values.

an average AUC of 0.9960, and for our homebrew scenarios, it produces an average AUC of 0.9827. This consistent performance demonstrates the robust generalizability of our features across different microservices-based applications.

Nonetheless, in the context of command injection, particularly pertaining to the *CVE-2012-1823* scenario, our approach exhibits a slightly reduced AUC score of 0.9479. This is attributable to the incidence of a few false positives, stemming from the inherent noise of replicas in normal operation. In addition to the AUC scores, we extend our evaluation by considering metrics such as precision, recall, etc. To maintain uniformity in evaluation, we empirically select a detection threshold of $\epsilon = 0.3$. While this threshold may not be optimal for each individual scenario, it guarantees consistently favorable performance overall.

As illustrated in Table IV, our approach achieves an average precision of 0.9248 and recall of 0.9813. For precision, we observe lower scores for *PHP-LFI*, *CWE-434*, and *CVE-2014-6271*. Here, the uniform threshold struggles to differentiate between attacks and the inherent noise from regular microservice usage. However, the high AUC scores in such scenarios, suggest that by adopting higher thresholds, our approach can notably improve performance. For recall, the lower score observed for *CVE-2018-3760* underscores the challenge our solution face in reliably detecting *path traversal* attacks. Sometimes, these attacks introduce only subtle variance among replicas, slipping past the threshold. However, the high AUC score achieved in such scenarios suggests that, given the microservice's inherently low noise level, our solution might be more adept at spotting such attacks with a lower threshold.

In short, while REPLICAWATCHER has limitations with authentication bypass attacks (see section VIII), it performs optimally across various scenarios of different microservices-based applications. It is noteworthy that we use the uniform threshold to evaluate our solution under the most challenging and generalized settings. Therefore, adjusting the threshold for specific use cases can further enhance its effectiveness.

E. Comparison with Existing Container HIDSes

We compared REPLICAWATCHER with STIDE-BoSC, CHIDS, and CDL [33] also in terms of detection capabilities. We evaluated these works on a dataset of 1000 normal and 100 attack traces per scenario, each lasting 30 seconds. We performed a 4-fold cross validation using 25:75 of normal traces, along with 100 attack traces as test data. Moreover, in adherence to the original papers' methodology, we apply the same sequence length and detection threshold in our evaluation. As per this, a trace is labeled anomalous if any of its constituent sequences are classified as such.

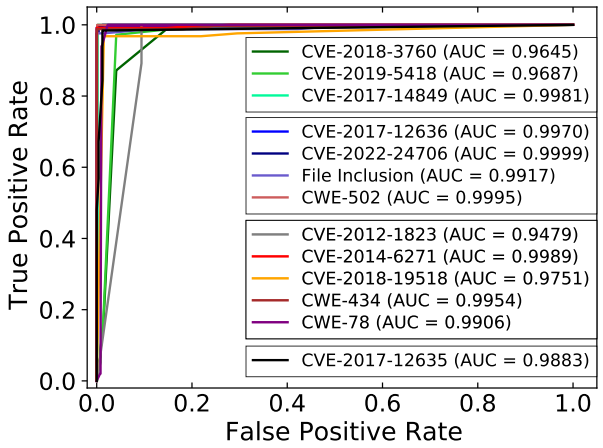


Fig. 8: AUC scores among different attack scenarios.

STIDE-BoSC. This work achieves an average recall exceeding 98% (Table IV), yet, it also experiences a substantial false positive rate of approximately 60%. This stems from unseen *BoSCs* emerging during regular activities, even when composed of previously seen syscalls, highlighting the challenge of modeling behavior with short sequences.

CHIDS. This work, leveraging diverse syscall properties to construct a baseline for containers, outperforms REPLICAWATCHER, CDL, and STIDE-BoSC across all attack scenarios. This superiority is visible in Table IV, with CHIDS achieving over 96% recall and 97% precision in all scenarios.

CDL. This work, relying on frequency analysis, is only able to detect attacks that result in unusual syscall frequencies. As reflected by the low recall (below 50%) across various scenarios in Table IV, this method falls short in detecting low-frequency attacks, consequently impacting its overall effectiveness.

REPLICAWATCHER outperforms both CDL and STIDE-BoSC across various scenarios, attributed to the use of features with diverse security semantics. Nonetheless, CHIDS slightly

TABLE IV: PERFORMANCE OF OUR APPROACH AND EXISTING WORKS. (*) INDICATES THE THRESHOLD SET BY THE AUTHORS.

Scenario	ReplicaWatcher ($\epsilon = 0.3$) Precision	Recall	F1-score	Accuracy	STIDE-BoSC ($mismatch = 10$) Precision	Recall	F1-score	Accuracy
CVE-2018-3760	0.9551	0.8720	0.9116	0.9155	0.5729	0.9803	0.7232	0.6248
CVE-2019-19518	0.9595	0.9720	0.9657	0.9655	0.5298	0.8235	0.6448	0.5463
CVE-2017-14849	0.9980	1.0000	0.9990	0.9990	0.5098	1.000	0.6753	0.5192
CVE-2022-24706	0.9881	1.0000	0.9940	0.9940	0.7016	0.9903	0.8214	0.7846
CVE-2017-12636	0.9881	0.9960	0.9920	0.9920	0.6995	0.9803	0.8165	0.7796
PHP-LFI	0.8000	0.9880	0.8841	0.8705	0.6340	0.9803	0.7701	0.7073
CWE-502	0.9990	1.0000	0.9995	0.9995	0.5048	0.9803	0.6664	0.5094
CWE-434	0.8012	0.9960	0.8881	0.8745	0.6340	0.8627	0.7426	0.7011
CVE-2012-1823	0.9029	0.9680	0.9343	0.9319	0.5984	0.9803	0.7432	0.6612
CVE-2018-19518	0.9132	1.0000	0.9546	0.9525	0.6519	0.8627	0.7426	0.7011
CVE-2014-6271	0.7604	1.0000	0.8639	0.8425	0.5482	0.9803	0.7032	0.5863
CWE-78	0.9699	1.0000	0.9847	0.9845	0.4983	0.9804	0.6608	0.4967
CVE-2017-12635	0.9877	0.9640	0.9757	0.9760	0.6952	0.9607	0.8067	0.7698

Scenario	CHIDS ($\gamma = 1.4$) Precision	Recall	F1-score	Accuracy	CDL ($99.99th\ of\ RE$) Precision	Recall	F1-score	Accuracy
CVE-2018-3760	1.0000	1.0000	1.0000	1.0000	0.9090	0.1000	0.1801	0.5450
CVE-2019-19518	1.0000	1.0000	1.0000	1.0000	0.9564	0.2200	0.3577	0.6050
CVE-2017-14849	1.0000	1.0000	1.0000	1.0000	0.6250	0.2000	0.3030	0.5400
CVE-2022-24706	1.0000	1.0000	1.0000	1.0000	0.9603	0.9700	0.9651	0.9650
CVE-2017-12636	1.0000	0.9800	0.9898	0.9900	0.9615	1.000	0.9803	0.9800
PHP-LFI	0.9740	1.0000	0.9868	0.9866	0.9374	1.000	0.9677	0.9666
CWE-502	1.0000	1.0000	1.0000	1.0000	0.4090	0.2400	0.3025	0.4466
CWE-434	0.9729	0.9600	0.9664	0.9666	0.4736	0.0600	0.1065	0.4966
CVE-2012-1823	0.9795	0.9601	0.9696	0.9701	0.8371	0.4800	0.6101	0.6933
CVE-2018-19518	1.0000	0.9800	0.9898	0.9900	0.9638	0.7200	0.8242	0.8464
CVE-2014-6271	0.9740	1.0000	0.9868	0.9867	0.4838	0.1000	0.1657	0.4966
CWE-78	1.0000	1.0000	1.0000	1.0000	0.1999	0.0200	0.0363	0.4700
CVE-2017-12635	1.0000	0.9800	0.9898	0.9900	0.7777	0.1400	0.2372	0.5500

outperforms our tool by establishing a baseline from various aspects of container behavior, significantly reducing noise and facilitating a more efficient anomaly detection. Importantly, REPLICAWATCHER stands out for its ability to handle updates effectively, a capability lacking in many other approaches.

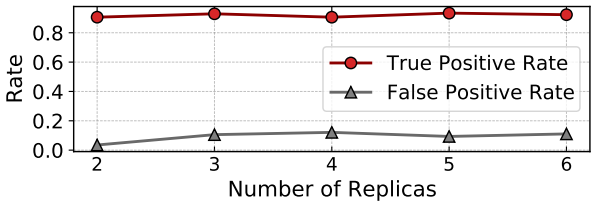


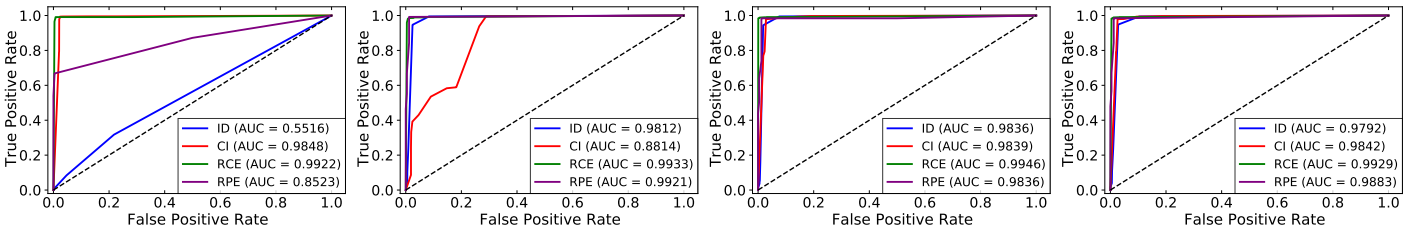
Fig. 9: **Performance of REPLICAWATCHER vs changes in the number of replicas.** Evaluated at the threshold of (ϵ = 0.3).

Impact of the number of replicas. We evaluate the performance of our solution in scenarios characterized by a varying number of replicas, specifically within the range of two to six. Figure 9 illustrates the correlation between the number of replicas and the respective average true positive rate (TPR) and false positive rate (FPR) across all attack scenarios. In evaluating the FPR, we find it relatively lower when the replica count stands at two. This can be due to a higher probability of user requests reaching all replicas, which induces a more evenly spread user behavior pattern. Conversely, we notice a moderate fluctuation in the FPR as the number of replicas increases. This can be largely attributed to the difficulty in discerning consistent behavioral patterns across multiple replicas. In fact, certain replicas may serve a distinct subset of users, thereby introducing noise and, thus, false positives. When examining the TPR, we observe that, on average, it remains higher, regardless of the number of replicas. Although some fluctuations are evident, they do not directly correlate with the number of replicas and could be ascribed to a variety of factors, including the inherent noise within the environment at a given moment. Overall, even with minor performance fluctuations, REPLICAWATCHER maintains reliable performance regardless of the number of replicas.

F. Ablation Study

To assess the impact of each feature category on our solution's performance, we conduct an ablation study, omitting one feature category at a time. We used ROC curves and AUC scores for evaluation. Figure 10 displays average ROC curves for each attack type based on threat impact, covering information disclosure (ID), command injection (CI), remote code execution (RCE), and remote privilege escalation (RPE).

Without FD-based features. We observe a decrease in the detection of ID attacks, with an AUC score of 0.5735. As these attacks involve unauthorized file/directory access, they do not execute distinct processes or syscalls. Thus, both normal and anomalous replicas often appear similar, making our approach generate a random classifier (AUC $\approx$ 0.5). We also see a drop in the detection of RPE attacks. These attacks, marked by unexpected file interactions and unauthorized changes, leave observables beyond just FD-based prints, leading to a moderate



(a) Without FD-based features (b) Without Process-based features (c) Without Syscall-based features (d) REPLICAWATCHER

Fig. 10: **Ablation Study**. Impact of feature category removal on AUC scores across attack scenarios.

impact. In short, we find that FD-based features are crucial for a comprehensive threat detection.

Without process-based features. Omitting process-based features leads to a drop in CI attack detection, underscoring the significance of commands, executables, and arguments as CI indicators. While CI detection suffers, other features detect diverse scenarios. Still, integrating process-based features is pivotal for comprehensive detection with REPLICAWATCHER.

Without syscall-based features. Upon the exclusion of syscall-based features, the remaining features still achieve strong performance across all attack scnearios. However, incorporating syscalls boosts performance for RCE and RPE scenarios, while slightly reducing it for RCE and ID. Despite some scenarios not benefiting, certain attacks like memory corruptions are primarily detected through syscalls. Therefore, for thorough detection, we recommend including syscall-based features, as also employed by prior research [1], [14], [33].

G. Runtime Performance & Scalability

The time complexity of REPLICAWATCHER depends on the number of pods within a node. While REPLICAWATCHER can be deployed on a third-party server, we favored implementing it as an HIDS on the worker nodes. This direct integration avoids potential lag from external server communications, offering a reliable execution time assessment. Note that the latency introduced by having replicas on multiple nodes does not affect the detection performances of our tool.

We spun all replicas of a microservice on the same worker node. Our evaluation covered the full pipeline, from *chisel*'s execution and log generation, to our solution's snapshot grouping, encoding, and classification. We segmented our analysis into five phases, beginning with eight microservices with two replicas each, and incrementally increasing to six replicas, totaling 48 pods. This number aligns with the median container-per-host density observed in real-world settings [54].

Figure 11 shows that REPLICAWATCHER's execution time grows sublinearly with increasing replicas. However, even with six replicas for each of the eight microservices (48 pods in total), our solution processes and classifies in under 2.25 seconds, making it feasible to run in practice. This stems from two factors: 1) *chisel* filters events upon receipt, ensuring quick logging and reduced latency, and 2) our selected feature set allows for fast processing at scale and supports generic anomaly detection.

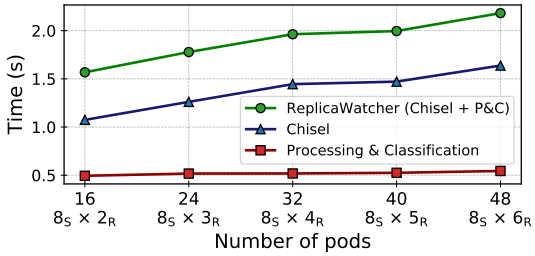


Fig. 11: **Average execution time of REPLICAWATCHER against different number of pods.** S = services; R = replicas.

H. Robustness against Evasion

We assessed the robustness of REPLICAWATCHER against evasion attempts by performing a set of stealthy attacks that *spread* malicious activity over time with the goal of staying under the detection threshold in each snapshot. We then gradually increased the amount/severity of the malicious activity to evaluate the detection capabilities of our system.

Command Injection (CWE-78) scenario. We deployed four replicas of the *shipping* microservice in GOB and designed our command injection (*CWE-78*) attack in a `Kill Chain` [62] fashion, with four phases: (Ⓐ) reconnaissance, (Ⓑ) preparation, (Ⓒ) exfiltration, and (Ⓓ) cleanup. We used eight commands (c = 8) for each phase—choosing a set of stealthy commands that reduce their footprint on the system (see Appendix B for more details). In each phase, we adjusted the attack intensity by incrementally executing one to six commands per snapshot. When using multiple commands, we appended them with a separator into a single command, ensuring responses from the same replica and mitigating load-balancing distribution noise.

Path Traversal (CVE-2017-14849) scenario. We deployed four replicas of the *payment* microservice in GOB. Using eight requests, we conducted *path traversal* to access unauthorized files. We adjusted the attack intensity by incrementally executing one to six requests per snapshot.

Snapshot collection. In both scenarios, we collected three snapshots for each combination $\binom{c}{n}$, yielding 3,690 snapshots in total. In *command injection*, n represents the number of distinct executed commands in a snapshot, while in *directory traversal*, it denotes the number of distinct *curl* requests.

Performance analysis. Table V presents our solution's performance against the two classes of attacks. For command injection, in Ⓐ, REPLICAWATCHER missed five of the eight commands when executing a single command per snapshot. This can be attributed to the use of shell built-in commands, avoiding noisy external commands. Yet, with increased command volume per snapshot, the detection rate rises. In Ⓑ, involving file creation, aggregation, and encoding, REPLICAWATCHER detects six of eight single commands. When executing multiple commands, REPLICAWATCHER correctly detected all snapshots. In Ⓒ, entailing data transfer to remote destinations, our solution yields a high detection rate in all settings. This is because exfiltration typically results in executing new processes and syscalls, producing a heavy footprint on the system, which our features capture. Note, however, that this does not mean that our tool always holds a perfect detection rate in all exfiltration scenarios. Our results only hold for our attack implementation (using `wget` and `curl`). For more stealthy exfiltration techniques, we have to realistically assume that our approach can be evaded. In Ⓓ, the attacker evades our solution with three singular commands to erase files or clear history. However, as command volume rises, so does the detection rate. Overall, it is possible for attackers to evade detection while injecting commands, however, such evasion is bound to performing limited malicious activity.

For *path traversal*, our solution missed seven of eight singular requests. These undetected requests predominantly targeted files within the `/etc` directory (e.g., `passwd`). The *payment* microservice, as part of its operation, accesses `/etc/localtime` to verify credit card expiration dates. Due to this benign interaction with the `/etc` directory, accessing an unusual file therein caused only subtle deviations, thus circumventing detection. However, similar to command injection, our detection rate rises with increased request volumes, as accessing multiple files in a single snapshot creates pronounced variations. In short, although it is possible for attackers to exfiltrate files while staying under the threshold, such an attack is bound to be slow, allowing for other approaches orthogonal to ours (e.g., network monitoring) to potentially spot it. Yet, adversarial attacks are a limitation of our approach.

REPLICAWATCHER during Kill Chain. We evaluated our command injection scenario in a dynamic setting where an attacker methodically progresses through the kill chain phases over time. We defined each phase to span 60 seconds, consisting of two snapshots. Within each phase, the attacker may execute one to six commands per snapshot. In an even stealthier scenario, the attacker might execute one command in the first snapshot and remain inactive in the subsequent one, all within the same phase.

TABLE V: AVG. DETECTION RATE BY NUMBER OF COMMANDS (TOP)/REQUESTS (BOTTOM) PER SNAPSHOT.

	No. Commands per Snapshot					
Phase	1	2	3	4	5	6
Ⓐ	0.3750	0.6428	0.8214	0.9285	0.9821	1.0000
Ⓑ	0.7500	1.0000	1.0000	1.0000	1.0000	1.0000
Ⓒ *	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Ⓓ	0.6250	0.8928	0.9821	1.0000	1.0000	1.0000

Note that our results only hold for our exfiltration implementation and do not generalize to any (more stealthy) exfiltration scenarios.

No. Requests per Snapshot					
1	2	3	4	5	6
0.125	0.250	1.000	1.000	1.000	1.000

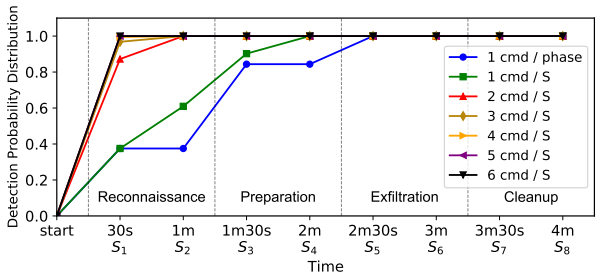


Fig. 12: **Performance of REPLICAWATCHER under stealthy kill chain attack.** `1 cmd / phase` denotes executing one command in a snapshot (S) followed by inactivity in the next one. Note that our results hold for our attack implementation and do not generalize to all (more stealthy) evasion scenarios.

To show REPLICAWATCHER's performance throughout the kill chain, we employed the cumulative detection probability, aggregating probabilities over phases. For a sequence of independent snapshots, the cumulative detection probability at snapshot S_n combines the accumulated probability at S_{n-1}, denoted as $P(cml_{S_{n-1}})$, with the detection probability at S_n, as expressed in Equation 4. The detection probability at S_n is given by the detection rates of our system as in Table V.

$$P(cml_{S_n}) = P(cml_{S_{n-1}}) + P(S_n) - P(cml_{S_{n-1}} \cap S_n) \quad (4)$$

Figure 12 showcases REPLICAWATCHER's detection probability across the kill chain phases. In reconnaissance, stealthy activities with fewer commands allow attackers a temporary evasion. However, as the command frequency rises, our detection capability sharpens. This sensitivity becomes more pronounced through the phases, culminating in the exfiltration stage, when REPLICAWATCHER consistently and effectively flags all anomalies. Nonetheless, our results only hold for our exfiltration implementation. As discussed in Section VIII, for more stealthy techniques, we realistically assume that our approach can be evaded.

In summary, this experiment showed that, while it is possible to craft an attack that evades our selected detection threshold, such an attack is bounded in terms of activity that attackers can perform without being detected. Thus, REPLICAWATCHER effectively mitigates evasion attempts.

VII. RETRAINING CHALLENGES & LIMITATIONS

Retraining an IDS is complex due to challenges such as handling labeling tasks and determining the optimal time to retrain [21]. Yet, a major challenge is distinguishing normality drifts from anomalies. In fact, attackers can manipulate retraining, making models less attack-sensitive [29]. We delve into how these issues manifest in existing container-based IDSes.

Consider an application that opens a file, reads data, and creates an *epoll* instance to handle multiple I/O events. Subsequently, it accepts a new connection from a client to accommodate an increased number of users. Finally, it closes the file and exits cleanly, resulting in a sequence S_1. Assume that both `epoll_create` and `accept` emerge as previously unseen syscalls, because of the increased load handling.

$$S_1 = \texttt{open}, \texttt{read}, \textbf{\texttt{epoll\_create}}, \textbf{\texttt{accept}}, \texttt{close}, \texttt{exit}$$

Imagine the same application is under an attack that exploits a *bashdoor* vulnerability to spawn a shell inside the container and exfiltrate data to an externally controlled server, generating S_2. In this context, `execve` and `sendto` emerge as previously unseen syscalls.

$$S_2 = \texttt{open}, \texttt{read}, \textbf{\texttt{execve}}, \textbf{\texttt{sendto}}, \texttt{close}, \texttt{exit}$$

CHIDS. When creating the syscall sequence graph (SSG) from $S1$ and $S2$, CHIDS groups all previously unseen syscalls into a single node. However, this aggregation method produces identical graphs and, thus, indistinct feature vectors for both $S1$ and $S2$, ignoring their specifics. As a result, in production settings, with only feature vector analysis, this technique obscures unseen syscalls, impeding the distinction between normal drifts and anomalies, and hindering retraining.

STIDE-BoSC. In forming *BoSCs*, the authors create the *other* category in the last cell of a *BoSC* vector, aggregating previously unseen syscalls and those that occur rarely in training. This aggregation technique leads to identical *BoSCs* for both $S1$ and $S2$, irrespective of their inherent differences. Such *BoSC* overlaps make differentiating true anomalies from normality drifts challenging, hence impeding retraining.

CDL. This work converts syscalls into a frequency vector, assigning each syscall a distinct position in the vector. Although this aids in distinguishing between normality drift and anomalies, the method's emphasis on frequency hinders its detection capability. Thus, despite $S1$'s unseen syscalls, their low frequency prevents alarm triggering. However, as in $S2$, a real attack with subtle frequency changes remains undetected.

Unlike the mentioned solutions, REPLICAWATCHER compares replicas in identical settings, removing the need for training or retraining. Our self-adaptive detection bypasses retraining challenges and avoids introducing room for adversarial attacks, which retraining inevitably brings in.

VIII. DISCUSSION

Despite effectively detecting attacks without prior training, REPLICAWATCHER is not exempt from limitations.

A. Detection Challenges & Potential Evasion

Brute-force attacks. One limitation of REPLICAWATCHER lies in detecting brute-force attacks. While these attacks involve a high number of benign attempts, their cumulative frequency suggests malicious intent. In Kubernetes, with the *Ingress* load-balancer [11], malicious requests are "evenly" distributed via *round-robin*, obscuring anomalous patterns and facilitating evasion. A solution is to monitor syscall frequency and enforce stickiness techniques (e.g., *session* or *IP affinity*), directing repeated attacker requests to a single replica. Yet, using multiple sessions or IPs complicates detection.

Potential for evasion. To evade our detection, attackers have two choices. 1) They can *simultaneously* affect all replicas with the same input, aiming for consistent behaviors. However, this requires knowing the replica count, which often varies with workload (i.e., horizontal pod autoscaling). Also, concurrent benign traffic can interfere with the distribution of the attack, complicating evasion. Employing stickiness techniques might accentuate abnormal patterns. However, these measures can still be susceptible to evasion from varied IPs or sessions. 2) Attackers can stage the attack over time, making each stage produce activity below the system's threshold. Determining the precise detection threshold is however challenging. Even with such knowledge, ensuring every stage consistently routes to the same replica is not always feasible due to the *round-robin* distribution. Furthermore, as the attacker progresses through the attack chain, they must rely on more conspicuous activities (e.g., downloading tools, using external commands) to achieve their aims, making their actions less subtle (Section VI-H).

Attack lead time. REPLICAWATCHER employs a 30-second monitoring interval to compare replicas. This time frame may not align well with the average microsecond (or even shorter) length of some attacks. For example, if an attacker swiftly injects a malicious command to exfiltrate confidential data, by the time REPLICAWATCHER's 30-second monitoring window detects anomalies, some data might already be compromised.

Identification of compromised replicas. We use the Jaccard similarity to compare replicas. Given its *symmetric* nature, we yield equal dissimilarity scores between a compromised and a normal replica and the other way around, complicating the identification of the compromised instance. While our focus is on classifying snapshots, we plan to explore methods for identifying individual compromised replicas in future work.

B. Deployment Challenges

Deploying REPLICAWATCHER in real-world settings requires facing a few challenges to guarantee generalization.

Replicas synchronization amidst rolling updates. We assume that all replicas run the same code version and updates are applied simultaneously across replicas. While this is challenging in large infrastructures, Kubernetes allows for rolling updates ensuring zero downtime [12]. Leveraging *readiness-probes*, our system can temporarily exclude replicas being created or deleted, focusing on the stable pods and preventing false alarms. Once new replicas stabilize, monitoring can resume normally, alleviating our assumption.

Rare bookkeeping activities. Replicas, despite processing different requests, consistently behave due to their narrow tasks. While we employ robust features and a 30-second interval to flatten out outliers, some rare tasks such as activities related to DevOps and maintenance (e.g., *execing* into a pod for debugging), might diverge from the microservice's standard function, and thus can be classified as anomalies.

Scalability in complex multi-cluster architectures. Scaling up replicas does not lead to noise. Our experiments validate

consistent performance even when scaling up. Yet, nowadays, large platforms (e.g., Skyscanner) leverage hundreds of nodes distributed over several clusters to deploy their microservices [15]. We expect such multi-cluster setups to introduce heterogeneity and latency, thus hindering the performance of our approach. Therefore, future work should explore the scalability of monitoring replicas in multi-cluster environments.

Threshold tuning and maintenance. In our research, we selected a threshold that is optimized for average performance across different microservices. In specific real-world deployments, this threshold might not be optimal. Thus, operators might need to tune the threshold based on their unique use cases, potentially requiring adjustments on a per-service basis.

Reliance on Sysdig. REPLICAWATCHER uses Sysdig to extract kernel events. This can pose challenges in fully-provisioned and managed environments (e.g., *GKE Autopilot*). Specifically, when using pre-configured node images not natively supported by Sysdig (e.g., container-optimized OS), syscall interception may be hindered. To address this, kernel headers might need manual configuration on each worker node, raising overhead and scalability concerns.

Multi-layered defenses. To cope with attacks and scenarios not included in our threat model (e.g., attackers attempting to target all replicas simultaneously), real-world deployments should combine REPLICAWATCHER with orthogonal approaches, e.g., network and resource monitoring.

IX. Related Work

Anomaly-based IDSes. These solutions monitor containers by leveraging either resource usage or syscalls. Resource usage based solutions [13], [27], [45], [46] work under the assumption that anomalies usually trigger a spike in a container's resource usage (e.g., CPU, memory, etc). While these techniques can successfully detect attacks that demand an excess of resources (e.g., Cryptojacking [28]), they overlook those that do not require such high resource consumption. For syscall-based solutions, besides CHIDS [14], CDL [33], and STIDE-BoSC [1], there exist various other approaches in this field. For instance, one approach focuses on tracking the frequency of Falco alarms as a key indicator of anomalies [55]. Another solution involves the integration of syscalls with explainable machine learning algorithms, offering a new perspective on how to interpret container anomalies [28]. Yet another approach uses n-grams of syscalls to identify anomalies based on their occurrence probabilities [48]. Contrary to these baseline-based approaches that demand periodic retraining to adapt to normality shifts, our approach effectively detects anomalies by comparing the patterns of replicas while accommodating drifts without the need for retraining.

Rule-based IDSes. These solutions [3], [40], [50], [53] monitor container activity using predefined security rules. These rules, comprised of specific filesystem operations, syscall patterns, and network activities, are regularly checked to match the actual container behavior. These solutions may struggle with the diversity of microservices. On one hand, enforcing out-of-the-box security rules may trigger unnecessary alerts given their broadness [51], thus leading to alert fatigue [23]. On the other hand, tailoring those rules to fit specific use cases entails deep knowledge of each microservice, and can be time-consuming and hard to scale for larger systems [51]. As opposed to these solutions, our approach compares each container with its identical replicas. This provides context-aware monitoring, thereby differentiating unexpected behavior with more precision and less effort.

Container isolation mechanisms. Several mechanisms leverage the principle of isolation such as Seccomp [7], SELinux [39], and AppArmor [20]. Seccomp restricts the syscalls a container can execute, significantly limiting its interaction with the kernel. Several works [17], [58] assist in identifying the *whitelisted* syscalls that a container should execute. AppArmor, another Linux kernel security module, confines program capabilities with specific profiles, enforcing the Unix access control. SELinux introduces mandatory access controls, offering even finer-grained control over system interactions. Though these solutions restrict unusual syscalls and capabilities, they might inadvertently block regular activities not exercised during training, potentially disrupting the normal functioning of the container.

Training-less anomaly detection. Prior research [47] used a training-less approach for anomaly detection in crowd scenes. Our work is the first to employ a training-less approach for the detection of anomalies in security applications.

Multi-version execution. Several works [4], [6], [10], [25], [26], enhance software security through n-version execution frameworks, monitoring syscalls on n executed variants for divergence. While REPLICAWATCHER shares some similarities with these works, it distinguishes itself as the first training-less anomaly detector for containerized environments, addressing unique challenges inherent to this context, such as scalability and intrusiveness. In fact, our solution monitors replicas at scale without altering the protected environment.

X. Conclusion

We presented REPLICAWATCHER, a training-less anomaly-detection approach for microservices-based environments. Our solution is based on comparing behavioral patterns of replicas, enabling the detection of anomalies without the need for a training baseline. In our evaluation, we showed that REPLICAWATCHER is resilient against updates and normality shifts, and it maintains its effectiveness without performance degradation and without the necessity for retraining. Our system effectively detects attacks with a low false positive rate and an acceptable overhead, serving as a stepping stone for effective intrusion detection in microservices-based settings.

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. This work is also supported by the SeReNity project, Grant No. cs.010, funded by Netherlands Organisation for Scientific Research (NWO).

REFERENCES

[1] A. S. Abed, C. Clancy, and D. S. Levy, "Intrusion detection system for applications using linux containers," in *Proceedings of Security and Trust Management (STM)*, 2015.

[2] R. Admin, "Runtime security in rancher with falco," 2020. [Online]. Available: https://www.suse.com/c/rancher_blog/runtime-security-in-rancher-with-falco/

[3] Aqua, "We stop attacks on cloud native applications." [Online]. Available: https://www.aquasec.com/

[4] K. Bauer, V. Dedhia, R. Skowyra, W. Streilein, and H. Okhravi, "Multi-variant execution to protect unpatched software," in *IEEE Resilience Week (RWS)*, 2015.

[5] G. Borello, "Extending sysdig with chisels." [Online]. Available: https://sysdig.jp/extending-sysdig-with-chisels/

[6] C. Cadar and P. Hosek, "Multi-version software updates," in *Proceedings of the International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, 2012.

[7] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *Proceedings of the 2021 on Cloud Computing Security Workshop (CCSW)*, 2021.

[8] R. Chandramouli, Z. Butcher *et al.*, "Building secure microservices-based applications using service-mesh architecture," *NIST Special Publication*, vol. 800, p. 204A, 2020.

[9] Cloud Native Computing Foundation, "CNCF Survey: Container Usage in 2021," 2021. [Online]. Available: https://www.cncf.io/wp-content/uploads/2021/08/CNCF_Survey_Report_2021.pdf

[10] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity."

[11] S. Dinesh, "Kubernetes nodeport vs loadbalancer vs ingress? when should i use what?" *Marzo de*, 2018.

[12] K. Documentation, "Performing a rolling update." [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

[13] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Algorithms and Architectures for Parallel Processing: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part IV 18*. Springer, 2018, pp. 560–572.

[14] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection," in *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2022.

[15] S. Engineering, "Kubernetes Security monitoring at scale with Sysdig Falco," *Medium*, 2020.

[16] Geeksforgeeks, "Introduction of syscall," 2019. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-system-call/

[17] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[18] GNU, "The GNU C Library (glibc)." [Online]. Available: https://www.gnu.org/software/libc/

[19] Google Cloud Platform, "Google Online Boutique." [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[20] A. Gruenbacher and S. Arnold, "AppArmor Technical Documentation," 2007.

[21] D. Han, Z. Wang, W. Chen, K. Wang, R. Yu, S. Wang, H. Zhang, Z. Wang, M. Jin, J. Yang *et al.*, "Anomaly detection in the open world: Normality shift detection, explanation, and adaptation."

[22] M. Hargrave, "Standard Deviation Formula and Uses vs. Variance," 2022. [Online]. Available: https://www.investopedia.com/terms/s/standarddeviation.asp

[23] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[24] C. Hope, "File descriptor." [Online]. Available: https://www.computerhope.com/jargon/f/file-descriptor.html

[25] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[26] ——, "Varan the unbelievable: An efficient n-version execution framework," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, 2015.

[27] S. Ji, K. Ye, and C.-Z. Xu, "Cmonitor: A monitoring and alarming platform for container-based clouds," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2019.

[28] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 3, pp. 674–691, 2020.

[29] Ł. Korycki and B. Krawczyk, "Adversarial concept drift detection under poisoning attacks for robust data stream mining," *Machine Learning*, 2022.

[30] Kubernetes, "Production-grade container orchestration." [Online]. Available: https://kubernetes.io/

[31] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices." in *Proceedings of the USENIX Security Symposium*, 2021.

[32] L. Liberti, C. Lavor, N. Maculan, and A. Mucherino, "Euclidean Distance Geometry and Applications," *SIAM review*, vol. 56, no. 1, pp. 3–69, 2014.

[33] Y. Lin, O. Tunde-Onadele, and X. Gu, "CDL: Classified Distributed Learning for Detecting Security Attacks in Containerized Applications," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2020.

[34] man7, "clock_nanosleep(2) — linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html

[35] ——, "getrlimit(2) — linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/getrlimit.2.html

[36] P. Mell, M. Quade, K. Scarfone, and S. Romanosky, "Nist special publication 800-190: Application container security guide," *National Institute of Standards and Technology*, vol. 2, no. 1, pp. 1–112, 2019.

[37] W. Meng, Y. Liu, S. Zhang, F. Zaiter, Y. Zhang, Y. Huang, Z. Yu, Y. Zhang, L. Song, M. Zhang *et al.*, "Logclass: Anomalous log identification and classification with partial labels," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1870–1884, 2021.

[38] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[39] A. Miller and L. Chen, "Securing your containers: An exercise in secure high performance virtual containers," in *Proceedings of the International Conference on Security and Management (SAM)*, 2012.

[40] NeuVector, "Full lifecycle container security platform." [Online]. Available: https://neuvector.com/

[41] OWASP, "Kubernetes security cheat sheet." [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html

[42] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Proceedings of the Network and Distributed System Security symposium (NDSS)*, 2020.

[43] R. R, "Microservices at ebay-what it looks like today." [Online]. Available: https://www.sayonetech.com/blog/microservices-ebay/

[44] O. Rick, "Containers and microservices — a perfect pair," 2021. [Online]. Available: https://developer.ibm.com/tutorials/cl-ibm-cloud-microservices-in-action-part-2-trs/

[45] A. Samir and C. Pahl, "Anomaly detection and analysis for clustered cloud computing reliability," *CLOUD COMPUTING*, vol. 2019, p. 120, 2019.

[46] ——, "Detecting and localizing anomalies in container clusters using markov models," *Electronics*, vol. 9, no. 1, p. 64, 2020.

[47] A. Sikdar and A. S. Chowdhury, "An adaptive training-less system for anomaly detection in crowd scenes," *arXiv preprint arXiv:1906.00705*, 2019.

[48] S. Srinivasan, A. Kumar, M. Mahajan, D. Sitaram, and S. Gupta, "Probabilistic real-time intrusion detection system for docker containers," in *Security in Computing and Communications: 6th International Symposium, SSCC 2018, Bangalore, India, September 19–22, 2018, Revised Selected Papers 6*. Springer, 2019, pp. 336–347.

[49] Sysdig, "Secure DevOps Platform." [Online]. Available: https://github.com/draios/sysdig

[50] ——, "Sysdig Secure." [Online]. Available: https://sysdig.com/wp-content/uploads/2019/03/sysdig-secure-compliance-feature-brief.pdf

[51] ——. (2021) Automated falco rule tuning. [Online]. Available: https://sysdig.com/blog/falco-rule-tuning/

[52] ——. (2021) Strengthen cybersecurity with shift-left and shield-right practices. [Online]. Available: https://sysdig.com/blog/strengthen-cybersecurity-with-shift-left-and-shield-right-practices/

[53] ——, "Falco: container native runtime security," https://falco.org/, 2022.

[54] ——, "Sysdig 2023 cloud-native security and usage report," pp. 1–29, 2023.

[55] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "Kubanomaly: anomaly detection for the docker orchestration platform with neural network approaches," *Engineering reports*, vol. 1, no. 5, p. e12080, 2019.

[56] S. Tonse, "Scalable microservices at netflix. challenges and tools of the trade," 2018.

[57] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[58] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 92–102.

[59] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE symposium on security and privacy (S&P)*, 1999.

[60] C. Xu, J. Shen, and X. Du, "A method of few-shot network intrusion detection based on meta-learning framework," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3540–3552, 2020.

[61] Z. Xu, "The Four Innovation Phases of Netflix's Trillions Scale Real-time Data Infrastructure." [Online]. Available: https://zhenzhongxu.com/the-four-innovation-phases-of-netflixs-trillions-scale-real-time-data-infrastructure-2370938d7f01

[62] T. Yadav and A. M. Rao, "Technical aspects of cyber kill chain," in *Proceedings of the Internation Symposium on Security in Computing and Communications (SSCC)*, 2015.

[63] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "Cade: Detecting and explaining concept drift samples for security applications." in *Proceedings of USENIX security symposium*, 2021.

[64] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International journal of machine learning and cybernetics*, vol. 1, pp. 43–52, 2010.

APPENDIX A
CASE STUDY

In this section, we delve into detailed examples on how REPLICAWATCHER detects attacks.

ShellShock (CVE-2014-6271). Table VI reveals that the compromised replica yields some observables not found in its counterparts. To be specific, the attacker spawns a *reverse shell* with the command `bash -c bash -i >& /dev/tcp/IP/444 0>&1`, introducing unusual syscalls: `execve` to start a `bash` process, `dup` to redirect STDIN and STDOUT to a socket, and `pipe` to link the shell process and network socket. Moreover, the attacker executes `uname -r` for reconnaissance, triggering the `uname` syscall and the `system` syscall category. Comparing features reveals distinct dissimilarity at both the syscall and process levels between the compromised replica and its counterparts. This results in a significant *Euclidean* distance from the origin and certain vectors, making the snapshot anomalous.

TABLE VI: SHELLSHOCK ATTACK. OBSERVABLES IN GRAY ARE INDICATORS OF ATTACK ACTIVITY.

Feature	Normal Replicas	Compromised Replica

cmdline	`apache2 -DFOREGROUND`	`apache2 -DFOREGROUND`
		`bash -c bash -i >& /dev/tcp/IP/4444 0>&1`
		`uname -r`

syscall type	`open, read, .., brk`	`open, read, .., brk` `execve, dup, pipe, uname`
syscall category	`ipc, file, .., net`	`ipc, file, .., net` `system`

CouchDB RCE (CVE-2022-24706). Table VII indicates the compromised replica's significant deviation. The attacker's use of the command `sh -c exec /bin/sh -s unix:cmd` leads to unusual syscalls, namely `execve`, `dup`, and `pipe`, related to the `sh` process. Thus, comparing features across replicas reveals notable differences at the syscall and proc levels, creating a large Euclidean distance between certain vectors and their origin, clearly marking the snapshot as anomalous.

TABLE VII: COUCHDB RCE ATTACK. OBSERVABLES IN GRAY ARE INDICATORS OF ATTACK ACTIVITY.

Feature	Normal Replicas	Compromised Replica
proc	`beam.smp, epmd, ..`	`beam.smp, epmd, ..` `sh`

command	`epmd -daemon`
	`erl_child_setup`
		`epmd -daemon`
		`erl_child_setup`
		`sh -c exec /bin/sh -s unix:cmd`

syscall type	`open, read .., brk`	`open, read .., brk` `execve, dup, pipe`

Path Traversal (CVE-2019-5418). Table VIII shows that the compromised replica behaves differently compared to the others. The attacker uses crafted paths to access sensitive data, leading to the `passwd` file and the `/etc` directory being generated. In comparing features across different replicas, a clear difference is observed at the file descriptor level between the compromised replica and the others. This difference leads to a large Euclidean distance between some vectors and the origin, making this particular snapshot stand out as unusual

TABLE VIII: PATH TRAVERSAL ATTACK. OBSERVABLES IN GRAY ARE INDICATORS OF ATTACK ACTIVITY.

Feature	Normal Replicas	Compromised Replica
filename	`*.html, *.css, *.erb`	`*.html, *.css, *.erb` `passwd`

directory	`/usr/src/rails/*`
	`/usr/local/bundle/gems/*`
		`/usr/src/rails/*`
		`/usr/local/bundle/gems/*`
		`/etc`

CouchDB RPE (CVE-2017-12635). Table IX demonstrates

a notable discrepancy in the compromised replica. The attack leads to unusual directories and files, such as `/usr/local/var/lib/couchdb` and `_users.couch`. This shows that the attacker is submitting `_users` documents with duplicate *roles* keys to circumvent access restrictions. Upon the examination of features across replicas, we find a notable discrepancy mostly at the file descriptor levels between the compromised replica and its counterparts. This difference translates into a considerable *Euclidean* distance from specific vectors to the origin, making the snapshot anomalous.

APPENDIX B
KILL CHAIN PHASES AND COMMANDS

In our simulation, we acted as an attacker, progressing through the *kill chain* stages: 1) reconnaissance, 2) preparation, 3) exfiltration, and 4) cleanup.

Reconnaissance. In this phase, the attacker reconnoiters the target system (e.g., OS release, current user, etc) using built-in shell commands, detailed in Table X. These commands, unlike external ones in `/usr/bin`, operate within the shell, yielding no new processes or syscalls. To illustrate, instead of directly reading `passwd` with a command like `cat`, the attacker uses `read < /etc/passwd` to extract the `passwd` content line by line, accessing the output with `echo $REPLY`. This can be more subtle and less likely to trigger alerts. Similarly, while executing `whoami` leads to the invocation of the `whoami` process and executing `uname -s` triggers the `uname` syscall, using `echo $USER` and `echo $OSTYPE` achieves the same goals with minimal footprints.

Preparation. In this phase, the attacker uses shell redirection to create an empty file using `> null`, a more subtle method than using editors (e.g., `vi`). By choosing a filename like `null` - a common in Linux systems (`e.g., /dev/null`), the attacker ensures the file blends seamlessly with typical system files, evading detection. Next, using redirection once more, the attacker channels the content of sensitive files into the `null` file with `> null < /etc/passwd`, avoiding heavy commands like `cp`. Last, to ensure the file content remains obfuscated during transfer, and given the lack of built-in shell commands for this purpose, the attacker resorts to external commands for encoding (e.g., `base64`).

Exfiltration. In this phase, the attacker aims to transfer the `null` file to a remote endpoint. Given the constraints of maintaining stealth in exfiltration, the attacker executes external commands like `curl` and `wget`, in their quiet modes to avoid noisy logging (e.g., `wget --quiet --post-file=null endpoint`).

Cleanup. In this phase, the attacker aims to conceal their activities by erasing evidence of their presence. They clear the command history using `history -c` or empty the command history with `echo "" > ~/.bash_history`. Additionally, they empty the `null` file they had transferred via `> null`. For file removal tasks, the attacker typically relies on external commands like `rm` or `unlink`.

TABLE IX: COUCHDB RPE ATTACK. OBSERVABLES IN GRAY ARE INDICATORS OF ATTACK ACTIVITY.

Feature	Normal Replicas	Compromised Replica
filename	`libc.so.6, mtab, ..`	`libc.so.6, mtab, .. _users.couch`
directory	`/dev, /etc /lib/x86_64-linux-gnu`	`/dev, /etc /lib/x86_64-linux-gnu /usr/local/var/lib/couchdb`

TABLE X: COMMANDS AND THEIR STEALTHY EQUIVALENTS IN RECONNAISSANCE

Attacker Aim	Less Stealthy	Stealthy
Get current user	`whoami`	`echo $USER`
Get OS release	`uname -s`	`echo ${OSTYPE}`
List directory	`ls`	`echo *`
Environment vars	`env`	`set`
Reading files	`cat /etc/passwd`	`read < /etc/passwd`

APPENDIX C
MICROSERVICE APPLICATION

As shown in Table XII, our homebrew application consists of several microservices, each developed using a unique technology. These services allow users to browse items, add their selections to a shopping cart, and complete purchases, replicating the shopping journey typically encountered on real-world e-commerce platforms.

APPENDIX D
ATTACK SCENARIOS

Table XIII describes the attacks used in our evaluation dataset. To thoroughly assess the detection capabilities of REPLICAWATCHER, we incorporate scenarios that reflect diverse threat impacts and scopes, targeting both our homebrew and GOB platforms.

APPENDIX E
FEATURE DESCRIPTION

Table XI gives a more detailed description of the studied features. We select features to incorporate in REPLICAWATCHER based on their resilience to *background noise*.

TABLE XI: THE STUDIED FEATURES ACROSS REPLICAS.

	Feature	Description	Security Rationale
Syscall	freq - executed	Frequency of syscalls	A spike in either successful or failed syscalls can signal repititive patterns to brute-force credentials.
	freq - failed	Frequency of syscalls returning *error* status	
	type	The name of the syscall (e.g. `open`)	An unexpected syscall of a distinctive category might indicate an unusual request to the kernel.
	category	The category of the syscall (e.g., *net* for `socket`)	
	max buffer len	The data buffer length for events that have one, like `read`	A large size may indicate the reading or exfiltration of a substantial amount of data
	max latency	The time spent waiting for the event to return (e.g., `poll`)	A prolonged latency/delta duration could hint at malicious activities, such as attempts to hook certain syscalls.
	max delta time	Delta between the current and the previous event	
File Descriptor	frequency	Frequency of accessed files	High frequency may hint at data exfiltration attempt.
	directory	If the `FD` is a file, the directory that contains it	Unexpected directories or filenames could signal unauthorized access or compromiseof the filesystem.
	filename	If the `FD` is a file, the filename without the path	
	type	Type of `FD` (e.g., `file`, `ipv4`, `unix`, `pipe`)	Unexpected type of FD could signal an unsual file-activity.
	client ip	The client IP address	Unusual addresses or/and ports may suggest external communication for malicious activities (e.g.,reverse shell).
	client port	The client port	
Process	proc	The name (excluding the path) of the process	Unusual process names can suggest rogue applications or malware on the system.
	cmdline	The full process command line	Unusual commands, executable names, and arguments might indicate unusual instructions prompting unauthorized code execution on the system.
	executable	The executable name (i.e., the first command line argument)	
	args	The arguments passed on the command line	
	cwd	The current working directory	Unusual cwds can hint at unauthorized activities within the system's file structure.

TABLE XII: MICROSERVICES AND THEIR APPLICATIONS.

Microservice	Description	Technology
SIGNUP	Handles the registration of new users on the e-commerce application.	`PHP-CGI`
LOGIN	Handles user authentication, validating credentials and initiating user sessions.	`PHP-IMAP`
PRODUCTCATALOG	Manages product listings, including details, images, and pricing.	`PHP-FPM`
PRODUCTDB	Stores product data in a CouchDB database, serving JSON responses to the Product Catalog service.	`CouchDB`
CART	Manages shopping cart operations including item additions, deletions, and quantity adjustments.	`PHP-Apache`
CHECKOUT	Handles the checkout process, including order confirmation, payment processing.	`Ruby and Rails`
USER PROFILE	Enables users to view and update their personal information.	`PHP-Apache`
USERDB	Stores and retrieves user data.	`MySQL`
SESSIONMANAGER	Manages user sessions to keep users authenticated.	`Redis`

TABLE XIII: DESCRIPTION OF THE ATTACK SCENARIOS INCLUDED IN THE EVALUATION DATASET.

CVE/CWE	Short Description
CWE-89	An attacker utilized `sqlmap` to exploit an SQL injection vulnerability within a web form, with the goal of bypassing login procedures and gaining unauthorized access
CWE-307	An attacker leveraged the brute-force tool `Hydra` to guess login credentials, exploiting the software's lack of effective measures against multiple failed authentication attempts in a short time period
CVE-2019-5418	An attacker employed the `directory traversal` technique via a manipulated *Accept* header, thereby accessing sensitive files on a targeted web server.
CVE-2018-3760	An attacker used `directory traversal` to craft requests for accessing files outside the application's root directory on *Sprockets*.
CVE-2017-14849	An attacker exploited a flawed `".."` handling vulnerability in `Node.js` to access unauthorized files.
CVE-2017-12636	An attacker with `admin role` in `CouchDB` executed arbitrary shell commands (admin role gained through *CVE-2017-12635*).
CVE-2022-24706	An attacker accessed an improperly secured default installation in `CouchDB` without authenticating and gained admin privileges.
File Inclusion	An attacker exploited a weakness in `PHP`'s file inclusion mechanism, tricking the system into including a malicious external file, which was then executed within the environment.
CWE-502	An attacker exploited a RCE vulnerability in the `Python Pickle` library to execute code remotely.
CWE-434	An attacker exploited insufficient file validation, uploaded a harmful script, and executed it in the environment.
CVE-2012-1823	An attacker exploited a flaw in `PHP CGI` and executed arbitrary commands by placing command-line options in the query string.
CVE-2018-19518	An attacker exploited a flaw in `PHP imap_open()` and executed arbitrary shell commands.
CVE-2014-6271	An attacker spawned a shell via the `Shellshock` vulnerability and executed arbitrary commands.
CWE-78	An attacker leveraged an input handling misconfiguration to inject commands remotely.
CVE-2017-12635	An attacker inserted `_users` documents with duplicate *role* keys (i.e., `_admin` key) and obtained unauthorized privileges.