PROCATCH: Detecting Execution-based Anomalies in Single-Instance Microservices

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

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

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

***Abstract*—Container anomaly-based detection systems are effective at detecting novel threats. However, their dependence on training baselines poses critical limitations. Research shows these baselines degrade rapidly in dynamic microservices-based environments, mandating frequent retraining to uphold performance — an operationally expensive process. Prior work (El Khairi et al., NDSS 2023) mitigates these challenges by comparing *replicas* — identical container instances — to detect anomalies, thereby eliminating the need for training and retraining. While effective, this approach relies on replication, making it ill-suited for single-instance deployments, such as during low-traffic periods when the orchestrator terminates idle replicas to optimize resources. Moreover, its reliance on long observation windows for replica comparison hinders its ability to detect modern, fast-moving container attacks.**

We propose a novel approach to detecting container anomalies. Our key insight is that containerized microservices, adhering to the single-concern model, execute a single workload throughout their lifecycle, resulting in *stable* execution behavior. This stability provides two key advantages. First, it enables immediate and precise profiling of expected execution behavior at container startup, eliminating the need for prior training. Second, it causes container attacks—typically involving adversarial code execution—to stand out as disruptions, forming a robust and setup-agnostic baseline for anomaly detection. Our system, PROCATCH, monitors the stability of execution behavior in microservices, promptly identifying disruptions as anomalies. We evaluate our approach against ten real-world container attack scenarios. The results demonstrate PROCATCH's effectiveness, achieving an average precision of 99.77% and recall of 100%, with an effective detection lead time.

I. INTRODUCTION

Cloud computing has driven a shift toward microservices, a preferred architecture for scalable and resilient systems. By decomposing applications into modular components, microservices enable precise scaling and streamlined development [1]. Containers have become the standard execution environment, offering lightweight architecture, efficient resource use, and rapid provisioning [2]. According to the latest CNCF survey, containers now power over 90% of cloud-native deployments, highlighting their dominance in modern infrastructure [3].

Nevertheless, the rapid evolution of cloud environments has reshaped the security landscape, exposing new exploitable attack vectors [4]. In fact, public-facing containerized microservices have become attractive targets for adversaries seeking to compromise cloud infrastructure. This risk is amplified by the ephemeral nature of containers. Recent studies show that 70% of containers terminate within five minutes [5]. While such short lifespans improve scalability and resource utilization, they also open narrow windows for launching fast, automated attacks, highlighting the need for timely and precise anomaly detection mechanisms.

Various solutions have been developed to address container attacks. In the industry, rule-based IDSes such as Falco [6] and Tracee [7] monitor kernel-level events using predefined rulesets to detect malicious activities. Beyond detection, container platforms such as Docker [8] and Kubernetes [9] provide preventive mechanisms, such as *Seccomp* profiles [10] to block "non-whitelisted" syscalls, and *read-only* root filesystems [11] to prevent filesystem writes [12]. On the research front, substantial efforts have focused on leveraging system call (syscall) analysis to detect malicious behavior in containerized environments. For example, CDL [13] monitors syscall frequency patterns, identifying anomalous spikes as indicators of potential compromise. A more advanced approach, CHIDS [14], integrates machine learning with graph-based representations of syscall traces to quantify the influence of previously unseen syscalls and their arguments on container behavior.

While these solutions provide runtime security to containers, they show significant limitations. Rule-based solutions such as Falco and Tracee do not provide comprehensive coverage of container attacks. While they support custom rule creation, translating Indicators of Compromise (IoCs) into actionable rules demands substantial time, domain expertise, and a deep understanding of each microservice's behavior, making it difficult to scale. Moreover, the *read-only* filesystem proves ill-suited for microservices that demand writable storage for normal functionality (e.g., caching). While mounting ephemeral volumes (e.g., `emptyDir`) addresses these writing needs, it inadvertently introduces attack vectors that adversaries can leverage for code execution. On the other hand, anomaly-based detection systems such as CDL and CHIDS as well as preventive mechanisms such as *Seccomp*, are fundamentally constrained by their reliance on training to establish a baseline of normal behavior (e.g., legitimate syscalls). First, constructing such a baseline demands extended training periods to capture application-specific variations and edge cases. Second, even with a well-established baseline, the dynamic nature of microservices — driven by frequent feature rollouts — inevitably alters the definition of "normal" behavior, rendering these baselines quickly obsolete [15]. This rapid drift often results in an excessive volume of false positives, necessitating periodic retraining to uphold performance — an operationally impractical process in security applications [15], [16].

These limitations position REPLICAWATCHER [15] as a significant advancement in threat detection for containerized microservices. Built on the insight that replicated microservices, adhering to the single-responsibility principle, consistently exhibit analogous behavior under normal operation, REPLICAWATCHER systematically compares the behavior of replicas on a worker node to identify inconsistencies as anomalies, eliminating the need for training and retraining. Yet, this approach has two notable limitations. First, its reliance on replication makes it inoperative in non-replicated environments. Specifically, during low-traffic periods, some replicas remain idle and are flagged by orchestrators (e.g., Kubernetes) as unnecessary resources, triggering automatic downscaling via the horizontal pod autoscaler (HPA) [17]. While this optimizes resource allocation, it can reduce a microservice to a single-instance deployment. Similarly, resource constraints may cause the orchestrator *scheduler* to place a replica on a different node than its peers, effectively isolating it as a standalone instance. In both scenarios, the absence of comparable replicas renders REPLICAWATCHER inactive, leaving microservices vulnerable to undetected threats. Second, its reliance on prolonged monitoring windows (i.e., 30 seconds) – to filter out replicas' background noise – fails to align with modern container attacks that unfold within seconds. In fact, adversaries increasingly leverage the ephemeral nature of containers and AI to execute automated attacks [5], effectively outpacing REPLICAWATCHER's detection capabilities.

In this work, we propose a novel approach to container execution-based anomaly detection that does not mandate training, enables real-time detection, while *supporting non-replicated deployments*. Our solution rests on the key observation that containerized microservices execute a single workload [18], [19], leading to a stable and consistent execution behavior throughout their lifecycle [15]. This stability allows for the immediate and accurate profiling of expected execution behavior at container startup, obviating the need for prior training to establish a baseline of "normal" behavior. Furthermore, it establishes an effective foundation for detecting container attacks. These attacks typically involve running adversarial code within the container [20], such as deploying cryptominers, establishing command-and-control (C2) channels, or planting persistence backdoors—inevitably disrupting the stable execution behavior of microservices. Such disruptions serve as reliable indicators of malicious activity, enabling anomaly detection without requiring specific deployment setups (e.g., replication).

While this approach eliminates the need for training and enables effective attack detection in non-replicated deployments, its effectiveness hinges on the ability to handle the inherent *background noise* introduced by runtime dynamics. This noise introduces complexity, necessitating robust monitoring mechanisms to reliably differentiate benign variations from genuine anomalies.

Background noise. Microservices typically exhibit stable execution behavior throughout their lifecycle, driven by their context-bounded design. However, normal operational dynamics, such as concurrent processing, traffic handling, and resource scaling, can introduce variability into execution activity, potentially increasing the risk of false positives. Therefore, our approach must be resilient to such noise, distinguishing benign transient behavior from adversarial efforts with high fidelity.

To address this challenge, we present PROCATCH, a novel approach to container execution-based anomaly detection. At its core, PROCATCH profiles deterministic execution attributes at container startup to establish a baseline of expected behavior – a task designed to be training-less and lightweight. At runtime, it detects deviations from this baseline as indicators of potential compromise. In summary, our key contributions are as follows:

- We present a novel approach for detecting container execution-based anomalies by leveraging the stable execution behavior inherent to microservices, identifying disruptions from this stability as adversarial behavior.
- We implement this approach in a prototype named PROCATCH, a node-based IDS for execution-based anomalies in containerized microservices.
- We evaluate PROCATCH on four representative microservices-based applications from major cloud vendors, achieving 99.77% precision and 100.00% recall across 10 real-world container attack scenarios.

In the spirit of open science, we make PROCATCH available at https://github.com/asbatel/procatch.

II. PRELIMINARY ASSESSMENT

Detecting anomalies based on execution behavior requires addressing two key challenges: (**A**) identifying execution-level attributes that carry strong security semantics, and (**B**) ensuring their robustness to benign variability to reduce false positives. To this end, we conducted a preliminary assessment to explore candidate attributes and selected those best suited for our detection approach.

A. Execution Attribute Exploration

In Linux, each process is associated with a well-defined set of attributes that describe its execution context and environment [21]. These include the *process name* (i.e., `proc`), which identifies the currently running process; the *executable name* (i.e., `exe`), denoting the binary or script used to spawn the process; the *executable path* (i.e., `exe_path`), representing the absolute location of the binary within the filesystem; the *executable inode* (i.e., `exe_inode`), which refers to the inode number of the executable; and the *current working directory*, indicating the directory context in which the process executes. In our assessment, we focus on all these execution attributes, as they are fundamental to malicious execution and reliably capture artifacts indicative of adversarial activity [15].

TABLE I: MICROSERVICES-BASED APPLICATIONS.

Application	# of microservices	Polyglot	Maintainer
BOOKINFO [22]	4	✓	Istio
SOCK-SHOP [23]	7	✓	Weavework
MU-SHOP [24]	9	✓	Oracle
MARTIAN BANK [25]	8	✓	Cisco

B. Execution Attribute Stability under Normal Behavior

To address the second consideration, we evaluate the stability of selected execution attributes during normal operation. Specifically, we assess how consistently these attributes remain within their baseline over time, quantifying their sensitivity to background noise and runtime variability.

Preliminary setup. We conducted our assessment on four microservices-based applications: BOOKINFO [22], SOCK-SHOP [23], MU-SHOP [26], and MARTIAN-BANK [25]. These applications span different languages and runtimes, reflecting the heterogeneous environments typically found in production-grade clusters, and are widely used in state-of-the-art microservices security research [1]. We deployed these applications on a single-node *Kubeadm* cluster with 16 GB of RAM, running *Ubuntu 24.04.1 LTS* as the node operating system and *containerd* [27] as the container runtime. To collect execution attributes, we employed *Falco* [28] as a kernel tracing tool, providing fine-grained visibility into execution behavior. To simulate realistic workloads, we generated diverse traffic using *Locust* [29], *Selenium* [30], and *Curl* [31], ensuring broad coverage of typical application interactions. We ran each microservice for *three hours*, starting monitoring only after the corresponding pod [1] reaches the `PodReady` state [32]. This state indicates that pod initialization has completed, including entrypoint execution, root filesystem setup, and successful completion of the readiness probe [33], ensuring the microservice is fully operational. This timing also excludes ephemeral binaries invoked during initialization. At the `PodReady` timepoint, we recorded the baseline set of execution attributes for each microservice. We then monitored each microservice at one-second intervals for three hours, capturing observed attributes at each interval. To quantify stability, we computed the ratio of attributes not present in the baseline to the total number of attributes observed at each interval. Table II reports the average stability across all intervals and microservices within each application. A value of 1 indicates complete stability, where no new attributes appeared beyond the baseline.

TABLE II: STABILITY IN EXECUTION ATTRIBUTES

Application	`proc`	`exe`	`exe_path`	`exe_inode`	`cwd`
BOOKINFO [22]	0.7825	1.0000	1.0000	1.0000	1.0000
SOCK-SHOP [23]	0.8914	1.0000	1.0000	1.0000	1.0000
MU-SHOP [26]	0.9088	1.0000	1.0000	1.0000	1.0000
MARTIAN BANK [25]	1.0000	1.0000	1.0000	1.0000	1.0000

Results analysis. As shown in Table II, all execution attributes remain stable across applications, with the sole exception of the `proc` field, which captures process names. Deviations in this field arise from runtime-level behaviors common in managed execution environments, such as thread creation and internal thread renaming. For example, in some BOOKINFO microservices, the Java Virtual Machine (JVM) spawns additional threads—such as `Executor` and `Signal Reporter`—after the `PodReady` timepoint, as part of its background scheduling and signal handling. Similar patterns are observed in other applications: a SOCK-SHOP microservice using the V8 JavaScript engine spawns a dedicated `V8` thread at runtime, while JVM-based microservices in MUSHOP exhibit renamed threads such as `C1` and `HikariPool-1`, reflecting just-in-time compilation and internal resource management. In contrast, all executable-related fields (e.g., `exe`, `exe_path`, and `inode`) remain stable across all applications. These attributes reference the binary invoked by the container's entrypoint, which encapsulates the full execution context of the microservice. Also, the current working directory field exhibits similar stability, as microservices typically run non-interactive workloads and retain the working directory configured during initialization, commonly defined via the image's `WORKDIR` directive.

Armed with these observations, we build our detection approach exclusively on attributes that remain stable across microservices, deliberately excluding the process name attribute (i.e., `proc`) due to its susceptibility to benign thread-level variation. While our preliminary assessment focuses on normal behavior, we show in Section V that the selected attributes generalize well to other microservices and remain robust under adversarial conditions.

[1] A pod is a Kubernetes abstraction that groups one or more containers.

III. THREAT MODEL

We focus on stateless microservices deployed within Kubernetes clusters, each operating within a well-defined and bounded context, consistent with the single-responsibility principle.

We consider an external adversary with access to a pod running as root—a misconfiguration observed in 83% of real-world deployments [34]. This access may result from vulnerabilities or misconfigurations in public-facing microservices. The adversary's goal is to execute code within the compromised pod to abuse computing resources, deploy malware, or exfiltrate data. We assume such actions inevitably introduce abnormal execution artifacts, deviating from the microservice's expected behavior. While stealthy adversaries may attempt to evade such artifacts, their techniques either lack practicality or remain constrained to low-impact activities, as we demonstrate in Section VI. Also, we do not consider attacks that rely on memory corruption, as modern microservices are predominantly written in memory-safe languages such as *Go*, *Java*, and *Python*. We further assume that microservice images are sourced from trusted registries, excluding attacks involving unverified or adversary-controlled images. Finally, we do not consider non-execution-based attacks (e.g., directory traversal), as they do not manifest at the execution level and would unavoidably leave traces that tools orthogonal to ours—such as web application firewalls —can detect via HTTP(S) traffic inspection.

Our system is an anomaly-based IDS designed to monitor microservices for execution-based anomalies. We assume Falco is deployed on all worker nodes to provide execution–level visibility and enforce our detection logic. While attackers may attempt to tamper with our solution, doing so requires escaping the container environment to the underlying node. Although container escape attacks are a valid concern, they are outside the scope of our work.

IV. APPROACH

We propose PROCATCH, a detection system for identifying container execution-based anomalies. Our key observation is that microservices exhibit stable execution behavior due to their narrow operational scope. In contrast, container attacks commonly introduce abnormal executions either through the deployment of malicious

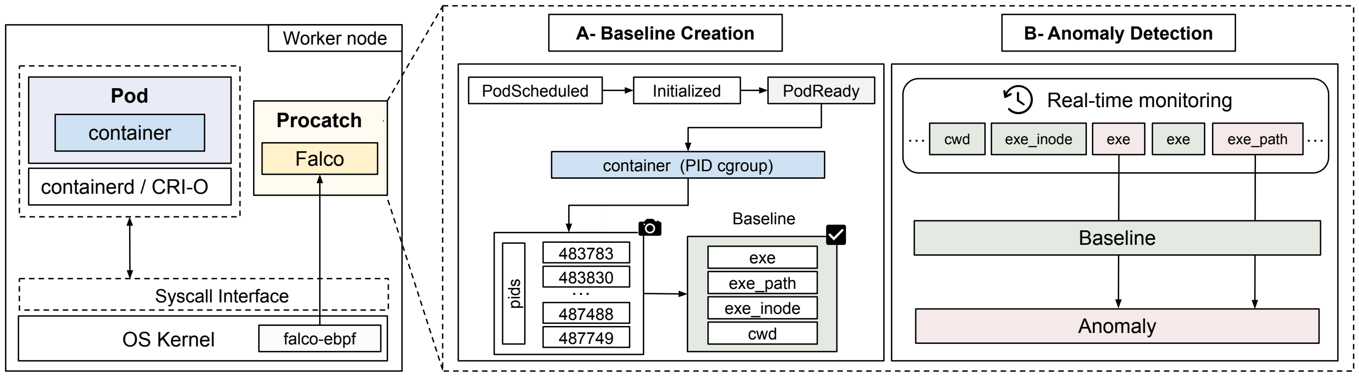

Fig. 1: **PROCATCH Overview.** (A) We establish a baseline by capturing execution attributes immediately after the pod reaches its ready state. (B) We then transition to active monitoring, using this baseline to detect abnormal observables.

payloads or the misuse of system utilities. By establishing a baseline of expected execution behavior at pod startup, PROCATCH continuously monitors for deviations indicative of malicious behavior.

Figure 1 illustrates the design of PROCATCH, which operates in two phases: baseline creation and anomaly detection. In the baseline creation phase (A), PROCATCH records all executables invoked by the pod—along with their absolute paths, inodes, and working directories—immediately after the pod reaches the ready state. In the anomaly detection phase (B), it continuously monitors the pod for deviations across these attributes.

A. Baseline Creation

The first phase of our approach establishes a baseline of execution-related attributes through two key steps:

1) Cgroup Identification.

Control Groups (cgroups) are a fundamental Linux kernel mechanism that enables fine-grained isolation, allocation, and monitoring of system resources across groups of processes [35]. In Kubernetes, each pod is assigned a dedicated *cgroup* hierarchy to manage its resource usage. Once the pod reaches the `PodReady` state, we establish a baseline of legitimate execution attributes. We start by identifying the pod's container and retrieving the host-level PID of its main process from container metadata. This PID is used to resolve the pod's *cgroup* via `/proc/<PID>/cgroup`. We then enumerate all PIDs belonging to that *cgroup*, yielding the complete set of active processes associated with the pod.

2) Execution Attribute Extraction.

For each process (i.e., PID) in the pod's *cgroup*, we extract execution attributes externally from the host without requiring in-pod instrumentation. Specifically, we obtain the *executable name* by parsing the first argument in `/proc/<PID>/cmdline`, which reflects the binary used to launch the process. We retrieve *executable path* by resolving the symbolic link `/proc/<PID>/exe`, yielding the absolute path of the binary as seen from the pod's root filesystem. We obtain *executable inode* via a metadata lookup on `/proc/<PID>/root/<exe_path>`. Finally, we extract the *current working directory* by resolving `/proc/<PID>/cwd`.

It is noteworthy that we establish a new baseline with each pod deployment, avoiding reliance on historical data or previous microservice versions. This ensures alignment with the current microservice image, adapting seamlessly to the evolving nature of microservices, without the need for training or retraining.

B. Anomaly Detection

At runtime, PROCATCH monitors each pod for deviations from its baseline by monitoring the executable name, absolute path, inode, and current working directory attributes. As shown in Section II, these attributes exhibit stable behavior under benign conditions. PROCATCH leverages this stability to flag any deviation as anomalous. This approach is fully training-less and rule-agnostic.

To enable efficient monitoring, we integrate our detection logic into Falco, a lightweight, eBPF-based runtime security engine that inspects execution activity directly from the kernel with minimal overhead [36]. Also, with over 130 million downloads [37], Falco is widely adopted in the security community and offers native integration with Kubernetes. This widespread use ensures that PROCATCH can be deployed seamlessly within existing environments, without introducing additional setup or infrastructure requirements.

V. EVALUATION

We implement PROCATCH as a *shell*-based tool and conduct a comprehensive evaluation of its effectiveness across diverse container attack scenarios. Beyond benchmarking it against REPLICAWATCHER – the only training-less detection system for containers, we also assess its detection lead time and resilience against evasion techniques. Finally, we evaluate PROCATCH's scalability by profiling its performance under varying workload densities, demonstrating its practicality in microservices-based environments.

A. Experimental setup.

Experimental setup. We evaluate our approach using four microservices-based applications: GOOGLE ONLINE BOUTIQUE [38], BANK-OF-ANTHOS [39], RETAIL-STORE [40], and AKS-STORE [41]. These applications are selected for their relevance to recent state-of-the-art research in container security [1], [15] and for their polyglot architectures, which reflect the diversity and scale found in real-world deployments. All applications are deployed on a single-node *Kubeadm* cluster with 16 GB of RAM, running

Ubuntu 20.04.6 LTS and using *containerd* [27] as the container runtime. We conduct our experiments under two operational modes.

Normal mode. We reuse the same traffic generation setup as in the preliminary assessment, employing tools such as *Locust* [29], *Selenium* [30], *curl* [31] to simulate realistic user flows [38]. To reflect dynamic cluster behavior, we enable the Horizontal Pod Autoscaler (HPA) [17], which adjusts microservice replicas in response to load. During the *high-traffic* phase, HPA scales the replicas to two, while in the *low-traffic* phase, it downscales them to one.

Attack mode. We simulate attacks under the assumption that the adversary has root access within the pod, via application-level vulnerabilities or misconfigurations. Instead of reproducing full attack chains, we focus on in-pod techniques commonly observed in real-world container compromises. These include cryptojacking (hijacking resources to validate cryptocurrency transactions), DDoS (deploying binaries to generate high-volume traffic), and proxyjacking (abusing the container's network to reroute traffic through proxy services). These attacks often involve additional steps such as establishing persistence, connecting to C2 infrastructure, disabling security mechanisms, and removing competing programs to maintain control. We replicate these behaviors using publicly available proof-of-concept exploits [42]–[44]. The attack scenarios are detailed in Table IV.

TABLE III: EVALUATED MICROSERVICES.

Name	# microservice	Polyglot	Maintainer
GOOGLE ONLINE BOUTIQUE [38]	10	✓	Google Cloud
BANK-OF-ANTHORS [39]	6	✓	Google Cloud
AKS STORE [41]	7	✓	Microsoft Azure
RETAIL STORE [40]	5	✓	Amazon AWS

TABLE IV: CONTAINER ATTACK SCENARIOS.

Attack Scenario	Microservices-based Application
`Sysrv-Hello` [44]	GOOGLE ONLINE BOUTIQUE
`Kinsing` [42]	GOOGLE ONLINE BOUTIQUE
`RebirthLtd` [45]	GOOGLE ONLINE BOUTIQUE
`Kangaro` [46]	BANK-OF-ANTHOS
`Shellbot` [43]	BANK-OF-ANTHOS
`Mirai` [47]	BANK-OF-ANTHOS
`Scarleteel` [48]	RETAIL-STORE
`Lucifer` [49]	RETAIL-STORE
`LABRAT` [50]	AKS-STORE
`TeamTNT` [51]	AKS-STORE

Evaluation dataset. In normal mode, we monitor each microservice for a continuous ten-hour period. While PROCATCH is designed to function in a "watcher" mode for real-time anomaly detection, for evaluation purposes, we segment this monitoring period into sequential intervals, referred to as *captures*. Each microservice has 1200 captures, each spanning 30 seconds and recording any abnormal execution attribute observed within the pod or its replicas. In attack mode (conducted independently of the ten-hour baseline monitoring period), we perform 20 attack runs for each microservice. While these attacks are intended to run for the lifetime of the container, we consider an attack complete once all preparatory steps (e.g., fetching binaries, disabling defenses, establishing a C2 channel, etc.) have been performed and the final payload is actively running. Each run targets a single application and captures the attack execution phase. Notably, during this time, the microservice continues to handle normal traffic, reflecting realistic conditions where legitimate and malicious activity coexist.

B. Detection Capabilities

Table V presents the performance of PROCATCH across diverse attack scenarios and microservices-based applications. PROCATCH consistently achieves a recall of 100%, supporting the central premise of our approach: container attacks invariably disrupt the stable and narrow execution behavior of microservices. Actions such as downloading payloads, invoking system utilities, or modifying file attributes result in the introduction of previously unseen executables—along with their associated paths and inodes—thereby diverging from the expected execution profile. By flagging these deviations, PROCATCH reliably detects container attacks.

Nevertheless, our solution exhibits minimal false positives, occurring exclusively when replicas scale down in response to load fluctuations. In Kubernetes, a pod consists of two containers: the main application container, which executes the primary workload, and the *pause* container, which runs in an idle sleep loop, preserving shared namespaces such as network and PID throughout the pod's lifecycle [52]. When querying the running container inside a pod during baseline creation, we obtain only the main application container. This is because Kubernetes treats the *pause* container as part of the pod sandbox rather than a user-managed workload, and therefore excludes it from standard container enumeration. As a result, our baseline inherently omits the *pause* container. Later, when a pod is terminated during downscaling, the *pause* container is gracefully stopped, causing its `pause` binary to exit. This results in the binary—along with its path and inode—being incorrectly flagged as anomalous, producing false positives. Yet, these alerts can be suppressed by correlating them with pod termination events from the `kube-apiserver` (e.g., *involvedObject.kind=Pod*, *reason=Killing*), effectively filtering out false positives related to pod shutdown.

TABLE V: DETECTION PERFORMANCE OF PROCATCH COMPARED TO REPLICAWATCHER (PREC: PRECISION, REC: RECALL, F1: F1-SCORE)

	PROCATCH			ReplicaWatcher		
Scenario	Prec.	Rec.	F1.	Prec.	Rec.	F1.
`Sysrv-Hello`	0.9983	1.0000	0.9991	0.9706	1.0000	0.9851
`Kinsing`	0.9983	1.0000	0.9991	0.9706	1.0000	0.9851
`RebirthLtd`	0.9983	1.0000	0.9991	0.9706	1.0000	0.9851
`Kangaro`	0.9972	1.0000	0.9986	0.9833	1.0000	0.9916
`Shellbot`	0.9972	1.0000	0.9986	0.9833	1.0000	0.9916
`Mirai`	0.9972	1.0000	0.9986	0.9833	1.0000	0.9916
`Scarleteel`	0.9967	1.0000	0.9983	0.9856	1.0000	0.9927
`Lucifer`	0.9967	1.0000	0.9983	0.9856	1.0000	0.9927
`LABRAT`	0.9988	1.0000	0.9994	0.9809	1.0000	0.9903
`TeamTNT`	0.9988	1.0000	0.9994	0.9809	1.0000	0.9903

Overall, PROCATCH offers strong detection capabilities with minimal transient false positives, making it an effective solution for execution-based anomaly detection in microservices environments.

TABLE VI: Automated Mirai Attack. ✓ indicates command detection, × indicates failure to detect, and 🔔 signals an alarm. 🟩 marks the first detected attack indicator, 🟥 marks `Mirai`'s execution. Detection lead time: `701ms`.

Timestamp	Attack steps	ProCatch	Artifacts
18:42.865	`cd /tmp`	×	
18:42.866	`wget http://<at-IP>/l4sd4sx64`	✓	
18:42.869 🟩		🔔	`exe=wget, exe_path=/usr/bin/wget, exe_inode=3467593, cwd=/tmp/`
18:43.567	`chmod 777 l4sd4sx64`	✓	
18:43.569		🔔	`exe=chmod, exe_path=/usr/bin/chmod, exe_inode=2593358, cwd=/tmp/`
18:43.570 🟥	`./l4sd4sx64`	✓	
18:43.572		🔔	`exe=./l4sd4sx64, exe_path=/l4sd4sx64, exe_inode=2599981, cwd=/tmp/`

C. *Comparison with* ReplicaWatcher

We evaluate ProCatch against ReplicaWatcher, a training-less anomaly-based IDS, using the dataset structure from the original paper—1000 normal and 100 attack snapshots per microservice. Also, we use a four-replica setup and apply the detection threshold ($\epsilon = 0.3$) and snapshot duration ($\tau = 30s$), in adherence to the original paper's defined parameters.

ReplicaWatcher. As shown in Table V, ReplicaWatcher achieves an average recall of 100%. The evaluated attacks involve extensive adversarial activity, including tool downloads, file permission changes, and malware deployment. These actions introduce abnormal syscalls, executables, process names, and file descriptors within the compromised replica, allowing ReplicaWatcher to detect them reliably. However, this approach yields some false positives. We attribute these to background noise in replica behavior, particularly at the syscall level, where transient variations are occasionally misclassified as anomalies.

ProCatch outperforms ReplicaWatcher. By leveraging the stable execution behavior of microservices, our solution monitors for abnormal execution-related artifacts, effectively detecting container attacks without the need for training, retraining(s) or any specific deployment setup such as replication.

D. *Attack Detection Lead Time*

In this section, we evaluate the detection lead time of ProCatch. As container attacks are increasingly automated and capable of unfolding within seconds [5], we define lead time as the interval between the first detected preparatory action and the execution of the attack payload. Typically, this payload either functions as the primary malware—such as a cryptominer or DDoS bot—or as a script that executes multiple actions before delivering the final payload, such as downloading tools (e.g., *crontab*), eliminating competing mining programs, etc. This definition quantifies ProCatch 's ability to generate timely alerts before the attacker executes the payload and impacts the system.

Lead time analysis. As shown in Figure 2, our solution detects most attacks with significant lead times. Across seven attack scenarios, our approach achieves detection lead times ranging from 700 to 900 ms. These attacks typically follow three steps: downloading the payload, setting its permissions to executable, and executing it. Notably, our approach consistently detects the attack during the download phase. Specifically, when the attacker

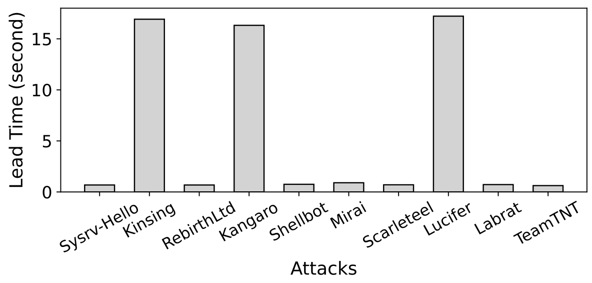

Fig. 2: **Detection lead time on different automated attacks.**

invokes `wget` to retrieve a payload, the action spawns an unseen executable along with its corresponding path and inode, triggering early detection before the execution of the payload itself. However, in the `Lucifer`, `Kangaro`, and `Kinsing` scenarios, we observe longer lead times of 17.22, 16.32, and 16.92 seconds, respectively. The observed delay is due to the attack sequence incorporating the installation of additional dependency tools (e.g., `cron`) before retrieving and executing the payload. Overall, the lead times achieved are sufficient to support timely intervention by incident response teams before the attack can escalate.

E. *Execution Time*

Falco has already demonstrated its efficiency and low overhead as a runtime tracing tool due to its eBPF-based event monitoring [36]. In our evaluation of ProCatch's execution time, we focus exclusively on the baseline creation phase. We measure how long ProCatch takes to generate the baseline for a pod. This involves identifying the container running within the pod, locating its *cgroup*, and extracting its execution attributes. We cover various deployment scenarios, ranging from a single pod to 40 pods, reflecting real-world pod-per-node densities [53]. To scale to 20, 30, and 40 pods, we replicate microservices accordingly. We evaluate the execution time of ProCatch on a *kubeadm* worker node with an Intel Core i7-10850H (Quad-Core, 2.7GHz) and 16GB of RAM.

ProCatch's execution time depends on three key factors: the number of pods on the node, the number of PIDs at the `PodReady` state (varying based on the microservice's base technology), and the overhead from querying pod and container metadata via the Kubernetes API and container runtime. Kubernetes API interactions introduce network and authentication overhead, while container

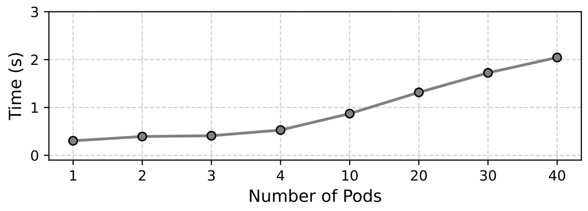

Fig. 3: **Avg. execution time of PROCATCH for baseline creation across different numbers of pods.**

runtime queries add processing overhead via gRPC [54]. As pod count grows, these factors collectively impact execution time. As shown in Figure 3, across 10 baseline creation runs, PROCATCH demonstrates sublinear scaling with respect to the number of pods on the node. It establishes a baseline for an individual pod in 0.4 seconds, scaling efficiently to complete baseline creation for 40 pods in just 2 seconds. This shows that PROCATCH sustains low-latency performance even at peak pod-per-node density, underscoring its computational efficiency and suitability for large-scale deployments.

F. Case Study

This subsection presents a detailed example of PROCATCH detecting an attack.

Kinsing. This attack involves the deployment of the *Kinsing* malware [42]. It begins by updating the package manager and installing auxiliary tools (e.g., *cron*), leading to the execution of system-level binaries for package management (e.g., */usr/bin/dpkg*), cryptographic verification (e.g., *gpgv*), and filesystem operations (e.g., *chmod*). The attacker then retrieves a staging script (`d.sh`) using *wget*, adjusts its permissions (*chmod*), and executes it (*d.sh*). The script coordinates a multi-stage attack: (1) disabling security services such as `Aegis`; (2) terminating competing cryptominers and removing their artifacts; and (3) downloading and executing the final *Kinsing* payload. Across these steps, the attacker invokes previously unseen binaries such as *ps*, *rm*, *chmod*, *grep*, *crontab*, *wget*, and *kinsing*, all observed with new paths and inodes—allowing our solution to detect the attack.

G. Robustness Against Evasion

In this subsection, we assess the robustness of PROCATCH against evasion attempts. To evade detection, an adversary must avoid introducing execution-based artifacts.

1) Evasion Techniques

We identify two techniques that can facilitate execution without leaving detectable observables.

Shell built-ins. These are commands interpreted and executed directly by the shell without spawning new processes or invoking external binaries [55].

In-line execution. This technique leverages existing binaries—such as embedded interpreters (e.g., python)—to execute malicious code within the scope of a legitimate executable.

2) Experiment description

In our experiment, we assume that the shell binary is part of each microservice's baseline, allowing the attacker to obtain shell access without triggering detection. Our analysis focuses exclusively on adversarial activities executed within the pod. We model all attacks in our dataset using a four-stage kill chain, including reconnaissance, environment preparation, payload preparation, and execution. These stages incorporate the evasion techniques introduced earlier (see Table VII for an example). We use the same experimental setup as in our evaluation: 20 attack runs per microservice. For each stage, we report the average detection performance across all attacks.

Reconnaissance. In this stage, attackers gather system information to understand the environment and assess potential execution vectors. This includes identifying the user ID to determine their level of access within the pod, inspecting applied capabilities to check for privilege restrictions (e.g., `/proc/self/status`), querying CPU and memory quotas to assess available resources (e.g., `cpu.cfs_quota_us`), and identifying the active binary running within the pod (e.g., `/proc/1/cmdline`).

Environment preparation. In this phase, the attacker prepares the pod for running the payload. For example, they increase the file descriptor limit to support more connections and open files, disable the NMI watchdog to prevent kernel lockup detection, and turn off security mechanisms such as SELinux to remove access restrictions. Additionally, they modify the shell's execution path to include the current directory, allowing payload execution without requiring explicit relative paths (i.e., omitting `./`).

Payload preparation. In this phase, the attacker prepares the payload using in-line execution to bypass the limitations of shell built-ins, which lack support for tasks such as file retrieval and permission modification. Instead of invoking external tools such as `wget` or `chmod`, the attacker leverages built-in functionality of the legitimately running binary. For example, if the pod runs `python`, they use `urllib.urlretrieve` to download the payload and save it under the name `python`, mimicking a legitimate binary. They then invoke `os.chmod` to adjust its permissions, all within the legitimate executable's context.

Payload execution. In this phase, the attacker executes the payload.

3) Performance Analysis.

As shown in Table VIII, shell built-in commands allow the attacker to operate quietly during the reconnaissance and environment preparation stages, as these commands do not spawn new executables and therefore avoid leaving detectable traces. Yet, this technique does not extend to payload preparation, which requires retrieving a payload and modifying its permissions—operations that typically involve external executables. To maintain stealth, the attacker uses in-line execution via an embedded interpreter. However, this technique depends on interpreter runtimes, which are absent in most microservices. In practice, microservices are typically compiled into self-contained binaries (e.g., `/src/server`, `/app/cart`) during the image build process. Thus, evasion was successful only in Python- and Node.js–based microservices. In all other cases, PROCATCH detects payload preparation, yielding an

TABLE VII: EXAMPLE OF STEALTHY MIRAI ATTACK ON A PYTHON-BASED MICROSERVICE.

Phase	Attack steps	Detection
Recon	`[ "$EUID" -eq 0 ] && echo "root"`	×
	`echo "$(</proc/self/status)"`	×
	`echo "$(</sys/fs/cgroup/cpu/cpu.cfs_quota_us)"`	×
	`echo "$(</sys/fs/cgroup/memory/memory.limit_in_bytes)"`	×
	`echo "$(</proc/1/cmdline)"`	×
Environment Preparation	`ulimit -n 65535`	×
	`echo '0' >/proc/sys/kernel/nmi_watchdog`	×
	`echo SELINUX=disabled >/etc/selinux/config`	×
	`export PATH=$(pwd):$PATH`	×
Payload Preparation	`python -c "import urllib.request; urllib.request.urlretrieve('<mirai-url>', 'python')"`	×
	`python -c "import os; os.chmod('python', 777)"`	×
Execution	`python`	✓

TABLE VIII: DETECTION PERFORMANCE AGAISNT EVASION.

Reconnaissance	Environment Prep.	Payload Prep.	Execution
0%	0%	66.72%	100.00%

overall detection rate of 66.72%. In the final execution phase, PROCATCH achieves a 100% detection rate. Even when the attacker assigns a benign name to the payload (e.g., `python`), the executable remains distinguishable due to differences in its absolute path and inode metadata, both of which are monitored by PROCATCH. Moreover, the execution of certain payloads introduce additional observables. For example, *xmrig*-based cryptominers invoke uncommon executables such as `/sbin/modprobe` to load the MSR kernel module for CPU-specific optimizations, while *Kinsing* executes `/usr/bin/getconf` as part of its environment profiling.

Overall, while techniques such as shell built-ins and in-line execution enable partial evasion in earlier stages, PROCATCH reliably detects the attack during the execution phase. However, our findings are specific to the stealthy attack implementations used in our evaluation. We acknowledge that more evasive variants—such as payloads written entirely in Python or Node.js using only built-in libraries—may evade detection in some microservices. Nonetheless, by monitoring unavoidable execution-level artifacts, PROCATCH imposes meaningful constraints on the attacker, making stealthy execution non-trivial.

VI. DISCUSSION

Despite effectively detecting attacks. PROCATCH is not exempt from limitations.

Rare bookkeeping activities. Microservices are narrowly scoped, typically running a single workload to fulfill their role. However, operations such as DevOps tasks (e.g., *execing* into a pod for debugging) can introduce anomalous artifacts, triggering false positives.

Timing attacks during initialization. As shown in Section V, our approach establishes a baseline for a pod before transitioning to active monitoring within a short timeframe ($\approx$ 0.4s for a single pod). Theoretically, an attacker could exploit the brief latency to execute malicious activities. However, this remains highly impractical as the attacker cannot predict pod scheduling or precisely time the attack during this short window. Yet, this can be mitigated by implementing a restrictive `NetworkPolicy` to block traffic temporarily after the pod is ready, further reducing this risk.

Baseline inclusion of attack-enabling utilities. Since our approach captures a baseline of execution attributes at the `PodReady` timepoint, certain system utilities (e.g., `chmod`) may be included if they remain active within the container's cgroup at that moment. While uncommon in microservices—where initialization typically completes before readiness—this poses a risk: once included in the baseline, these utilities can later be abused without triggering detection.

Non-execution-based attacks. Our approach does not detect non-execution attacks (e.g., directory traversal), where the adversary interacts with the application externally without deploying payloads or invoking binaries. Detecting such attacks falls outside our scope and requires complementary mechanisms, such as a web application firewall (WAF) capable of inspecting HTTP(S) traffic.

VII. RELATED WORK

Anomaly-based solutions. Most container monitoring solutions rely on syscalls to detect anomalies. For example, CDL [13] monitors syscall frequency and flags abnormal spikes. CHIDS [14] enhances detection by combining machine learning with graph-based modeling to analyze syscall frequency and argument patterns among other works. While these solutions capture malicious activity, they require frequent retraining to maintain performance. REPLICAWATCHER eliminates the need for training by comparing replicas for anomalies. However, it is not applicable for single-instance deployments and does not align with the speed of modern container threats.

Rule-based solutions. Several container security tools, including Falco and Trace [6], [28], [56], rely on IoCs to create their detection rulesets. While effective against known threats, they fail to detect novel attacks. Also, translating IoCs into rule syntax requires time and expertise, often lagging behind the evolving threat landscape [18]. Moreover, certain rules are broad, leading to excessive false positives and alert fatigue, which in turn necessitates fine-tuning — a time-consuming process that is difficult to scale [18].

VIII. Conclusion

In this paper, we presented PROCATCH, a training-less, real-time, and setup-agnostic approach to execution-based anomaly detection in containerized microservices. Our solution leverages the inherent design of containerized microservices and continuously monitors their *stable* execution behavior for disruptions indicative of malicious activity. PROCATCH achieves 99.77% precision and 100% recall on average, detecting container attacks with effective lead time, making it practical for microservices environments.

Acknowledgments

We would like to thank our reviewers for their valuable comments to improve our paper. We also thank Dr. Holger Dreger and Dr. Marco Caselli from Siemens AG for their support and for providing the computing infrastructure used in this study. This work is supported by the SeReNity project, Grant No. cs.010, funded by Netherlands Organisation for Scientific Research (NWO). Any views, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for inter-service access control of microservices. In *Proceedings of the USENIX Security Symposium*, 2021.

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

[3] Cloud Native Computing Foundation. Cloud Native 2024. https://www.cncf.io/wp-content/uploads/2025/04/cncf_annual_survey24_031225a.pdf, 2024.

[4] Chierici Stephano. Cloud Lateral Movement: Breaking in through a Vulnerable Container. https://sysdig.com/blog/lateral-movement-cloud-containers/, 2022.

[5] Dougla Nigel. Ephemeral Containers and APTs. https://sysdig.com/blog/ephemeral-containers-and-apts/, 2024.

[6] Sysdig. Falco: container native runtime security. https://falco.org/, 2022.

[7] Aquasec. Aqua tracee: Runtime ebpf threat detection engine. https://www.aquasec.com/products/tracee/.

[8] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[9] Kubernetes. Production-grade container orchestration.

[10] Kubernetes. Restrict a Container's Syscalls with seccomp. https://kubernetes.io/docs/tutorials/security/seccomp/.

[11] Kubernetes. Configure a Security Context for a Pod or Container. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.

[12] Ian Edwards. Compendium of container escapes. In *Black Hat USA*, 2019. Accessed: 2024-10-30.

[13] Yuhang Lin, Olufogorehan Tunde-Onadele, and Xiaohui Gu. CDL: Classified Distributed Learning for Detecting Security Attacks in Containerized Applications. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2020.

[14] Asbat El Khairi, Marco Caselli, Christian Knierim, Andreas Peter, and Andrea Continella. Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2022.

[15] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. Replicawatcher: Training-less anomaly detection in containerized microservices. In *Network and Distributed System Security Symposium, NDSS 2023*. Association for Computing Machinery, 2024.

[16] Dongqi Han, Zhiliang Wang, Wenqi Chen, Kai Wang, Rui Yu, Su Wang, Han Zhang, Zhihua Wang, Minghui Jin, Jiahai Yang, et al. Anomaly detection in the open world: Normality shift detection, explanation, and adaptation.

[17] Kubernetes. Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2024.

[18] Sysdig. Automated falco rule tuning. https://sysdig.com/blog/falco-rule-tuning/.

[19] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, 2021.

[20] Sysdig. Preventing container runtime attacks with Sysdig's Drift Control. https://sysdig.com/blog/preventing-runtime-attacks-drift-control/, 2022.

[21] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[22] Istio. Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/.

[23] Phil Winder Ian Crosby, Alex Giurgiu. Sock Shop : A Microservice Demo Application. https://github.com/microservices-demo/microservices-demo.

[24] Cedric Ziel Steve Waterworth. Sample Microservice Application. https://github.com/instana/robot-shop.

[25] Cisco. Martian Bank. https://github.com/cisco-open/martian-bank-demo.

[26] Oracle Cloud Infrastructure. Mushop. https://github.com/oracle-quickstart/oci-cloudnative.

[27] An industry-standard container runtime with an emphasis on simplicity, robustness and portability.

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

[29] Locust. Locust: An open source load testing tool. https://locust.io/.

[30] Selenium. Selenium automates browsers. That's it https://www.selenium.dev/.

[31] Curl. Curl. https://curl.se/.

[32] Kubernetes. Kubernetes documentation - kubelet. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.

[33] Kubernetes. Pod Lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, 2024.

[34] Sysdig. Sysdig 2023 Cloud-Native Security and Usage Report. https://sysdig.com/2023-cloud-native-security-and-usage-report/, 2023.

[35] Rami Rosen. Namespaces and cgroups, the basis of linux containers. *Seville, Spain, Feb*, 2016.

[36] Douglas Nigel. Falco vs. Sysdig OSS: Choosing the right tool for the job. https://sysdig.com/blog/falco-vs-sysdig-oss/, 2024.

[37] Sysdig. Falco Feeds by Sysdig Empowers Companies to Harness Open Source Security at Enterprise Scale. https://sysdig.com/press-releases/falco-feeds-by-sysdig/.

[38] Google Cloud Platform. Google Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[39] Google Cloud. Bank of Anthos. https://github.com/GoogleCloudPlatform/bank-of-anthos.

[40] Amazon AWS. Retail Demo Store. https://github.com/aws-samples/retail-demo-store.

[41] Azure. AKS Store Demo. https://github.com/Azure-Samples/aks-store-demo.

[42] Sysdig. Zoom into Kinsing. https://sysdig.com/blog/zoom-into-kinsing-kdevtmpfsi/, 2020.

[43] Sysdig. Malware analysis: Hands-On Shellbot malware. https://sysdig.com/blog/malware-analysis-shellbot-sysdig/, 2021.

[44] Sysdig. THREAT ALERT: Crypto miner attack – Sysrv-Hello Botnet targeting WordPress pods. https://sysdig.com/blog/crypto-sysrv-hello-wordpress/, 2021.

[45] NVD. DDoS-as-a-Service: The Rebirth Botnet. https://sysdig.com/blog/ddos-as-a-service-the-rebirth-botnet/, 2024.

[46] Asaf Eitani Assaf Moragn. Threat Alert: New Malware in the Cloud By TeamTNT. https://www.aquasec.com/blog/new-malware-in-the-cloud-by-teamtnt/, 2022.

[47] Nitzan Yaakov. Tomcat Under Attack: Exploring Mirai Malware and Beyond. https://www.aquasec.com/blog/tomcat-under-attack-investigating-the-mirai-malware/, 2023.

[48] Sysdig. SCARLETEEL: Operation leveraging Terraform, Kubernetes, and AWS for data theft. https://sysdig.com/blog/cloud-breach-terraform-data-theft/.

[49] Lucifer DDoS botnet Malware is Targeting Apache Big-Data Stack , author=Nitzan Yaakov, year=2024, howpublished=https://www.aquasec.com/blog/lucifer-ddos-botnet-malware-is-targeting-apache-big-data-stack/.

[50] Miguel, Hernández. LABRAT: Stealthy Cryptojacking and Proxyjacking Campaign Targeting GitLab. https://sysdig.com/blog/labrat-cryptojacking-proxyjacking-campaign/.

[51] Ofek, Itach and Assaf, Morag. TeamTNT Reemerged with New Aggressive Cloud Campaign. https://www.aquasec.com/blog/teamtnt-reemerged-with-new-aggressive-cloud-campaign/.

[52] Kumar Rajesh. What is Pause container in Kubernetes https://www.devopsschool.com/blog/what-is-pause-container-in-kubernetes/.

[53] Sysdig. Sysdig 2023 cloud-native security and usage report. pages 1–29, 2023.

[54] gRPC. A high performance, open source universal RPC framework. https://grpc.io/.

[55] Jason Andres. Exploring syscall evasion – linux shell built-ins.

[56] Aquasec. We Stop Attacks on Cloud Native Applications. https://www.aquasec.com/.