ConLock: Reducing Runtime Attack Surface in Containerized Microservices

Asbat El Khairi[1], Andreas Peter[2], and Andrea Continella[1]

[1] University of Twente, Netherlands
a.elkhairi@utwente.nl, a.continella@utwente.nl
[2] University of Oldenburg, Germany
andreas.peter@uni-oldenburg.de

Abstract. While detection and response are essential components of container runtime security, this reactive approach depends on the timely and accurate classification of threats to trigger mitigation. This dependency introduces unavoidable delays in the response path, creating a window of opportunity for adversaries to escalate privileges or pivot laterally across the cloud environment. These limitations underscore the need for a proactive hardening mechanism that reduces the container's execution surface before it is compromised. While existing hardening mechanisms enforce security-relevant constraints, their effectiveness remains inherently tied to specific deployment configurations and does not generalize across diverse setups. Moreover, their dependence on prior knowledge of container behavior makes them ill-suited for dynamic microservices, where frequent rollouts invalidate baselines and introduce substantial maintenance overhead.

In this work, we present ConLock, a hardening mechanism that generalizes across diverse deployment setups and requires no prior knowledge of container behavior. Our key insight is that containerized microservices, consistent with the single-concern principle, execute a single, task-specific binary throughout their lifetime. In contrast, adversaries violate this model by invoking additional executables—typically pre-packaged within the container image—to carry out malicious actions. At its core, ConLock identifies the main binary at startup and purges all non-essential executables from the container's runtime filesystem, thereby preventing unauthorized code execution. We evaluate ConLock on five microservices-based applications maintained by major cloud vendors and 21 container attack scenarios derived from publicly available exploits. ConLock achieves a 99.57% attack prevention rate with zero false positives. Moreover, it operates without in-container instrumentation or system call hooking, incurring minimal performance overhead.

1 Introduction

Microservices are widely adopted for developing scalable and maintainable cloud applications. By decomposing functionality into isolated services, they enable modular development, fault tolerance, and flexible scaling [1]. Containers serve

as the preferred runtime for microservices, offering lightweight execution, rapid startup, and consistency across environments [2]. Their integration with orchestration platforms such as Kubernetes[3] has led to widespread adoption across cloud infrastructure. Thus, containerized microservices now underpin a majority of production systems, forming a critical part of cloud-native applications [3].

As containerized microservices become popular, they have also become a focal point for adversaries. The inherent uniformity and predictability of container environments, combined with advancements in AI, enable attackers to automate and scale their operations with ease. Notably, container-based attacks can complete key phases such as initial access, payload delivery, lateral movement, and even container escape within minutes [4], often outpacing existing detection and response mechanisms. This growing asymmetry highlights the need for security measures that act earlier in the attack chain, before malicious code can execute.

Container security efforts across both industry and academia have predominantly focused on detection. Industry tools such as Falco [5] and Tracee [6] rely on kernel-level heuristics to identify suspicious activity at runtime. In the academic domain, a variety of approaches have explored anomaly detection through system call (syscall) analysis. CDL [7] monitors syscall frequencies and flags abnormal spikes. Stide-BoSC [8] uses sliding n-gram windows to model short syscall sequences, detecting attacks via sequence mismatches. CHIDS [9] enhances expressiveness by constructing graph-based representations of container behavior, incorporating syscall arguments and execution context. More recently, ReplicaWatcher [10] moves away from static baselines, instead comparing replicas at runtime and flagging behavioral inconsistencies as potential compromises.

While existing solutions have advanced threat detection in containerized environments, their ability to mitigate attacks remains fundamentally limited. Many detection systems generate alerts without actionable context, leaving security teams with limited guidance during triage. Even with access to enriched contextual data, response mechanisms—whether automated or operator-driven—are inherently reactive, initiated only after adversarial activity has been observed. This reactive posture introduces operational latency that conflicts with the fast-paced execution model of containerized environments, where attacks can advance through critical stages in seconds. These limitations necessitate proactive runtime hardening that defines strict execution boundaries and blocks unauthorized executions at runtime.

Container runtime hardening is commonly enforced through three complementary mechanisms: mandatory access control (MAC), syscall filtering, and filesystem immutability. MAC frameworks such as AppArmor [11] enforce path-based policies that constrain access to files, network resources, and Linux capabilities. Seccomp[12] enables syscall filtering by defining a whitelist of allowed syscalls, thereby limiting the container's interaction with kernel-level functionality. Filesystem immutability, typically enforced via a read-only root filesystem[13], prevents write operations within the container's root filesystem during runtime.

[3] https://kubernetes.io/

While these mechanisms operate at different layers of the container stack and reduce the attack surface, they suffer from three fundamental limitations. First, they are prone to *deployment fragility* under over-permissive configurations. Containers running in privileged mode execute in an unconfined context, rendering mechanisms such as Seccomp and AppArmor ineffective. Moreover, privileged access allows remounting the root filesystem as writable, bypassing read-only restrictions. Second, they exhibit *semantic gaps*. Even without elevated privileges, attackers can abuse memory-backed filesystems (e.g., *tmpfs*) to execute payloads entirely in memory, evading read-only filesystem protections. Moreover, Seccomp and AppArmor profiles often permit syscalls and capabilities required for benign functionality, some of which can be repurposed for malicious use. Third, they depend on *prior knowledge of legitimate behavior*. Seccomp and AppArmor require workload-specific profiles (e.g., whitelisted syscalls), while enforcing a read-only filesystem requires pre-identifying application-specific writable paths and mounting ephemeral volumes accordingly. These requirements introduce non-trivial maintenance overhead that becomes increasingly burdensome in fast-evolving microservice environments.

In contrast to these solutions, we propose a novel hardening mechanism that reduces the execution surface within containers. The approach builds on the observation that microservices typically execute a single long-lived binary throughout their lifecycle, consistent with the single-responsibility principle [14]. Conversely, container-based attacks often rely on auxiliary binaries pre-installed in the image (e.g., shells, download tools, and file manipulation utilities) to perform malicious actions [15]. By permitting only the long-lived binary identified at startup and purging all auxiliary binaries, our mechanism eliminates a primary avenue for post-startup compromise.

While this approach offers the advantage of remaining effective in privileged setups, being resilient to evasion, and requiring no prior knowledge of container behavior, its effectiveness hinges on limiting the execution surface without disrupting essential initialization logic—commonly referred to as *startup noise*.

Startup noise. While containerized microservices are designed to run a single, long-lived service binary, they often invoke short-lived auxiliary utilities during initialization. These transient binaries are executed by entrypoint logic to perform necessary environment setup tasks (e.g., configuring permissions, initializing filesystems, loading environment variables) before launching the main application. Hence, enforcing execution-surface reduction prematurely can remove required binaries, causing startup failures and repeated container restarts.

To address this, we introduce ConLock, a hardening mechanism that enforces *execution surface reduction* by retaining only the main microservice binary and removing access to all other executables. Our key observation is that, despite transient startup noise, containerized microservices reliably converge to a steady state in which only the long-lived binary remains active. ConLock identifies this convergence point at runtime and systematically purges all remaining filesystem-resident executables, thereby eliminating auxiliary execution paths

and preventing post-initialization binary invocation—a prevalent technique in container-based attacks.
In summary, we make the following contributions:

- We introduce a novel container hardening technique based on *execution surface reduction*, leveraging the observation that microservices consistently execute a single, stable binary throughout their lifecycle.
- We implement this technique in ConLock, a lightweight prototype that reduces the execution surface at startup, without prior knowledge of container behavior, syscall hooking, or in-container instrumentation.
- We evaluate ConLock across five representative microservices-based applications and 21 attack scenarios, showing that it consistently blocks attacks with an average prevention rate of 99.57% without any false positives.

We make ConLock available at https://github.com/asbatel/conlock.

2 Motivation

While reliance on prior knowledge of container behavior is a well-known limitation of existing hardening mechanisms, this section presents two realistic container attack scenarios that highlight their *deployment fragility* and *semantic gaps*. We focus on the two most widely used hardening techniques in practice: Seccomp and read-only filesystems.

2.1 Deployment Fragility

Scenario. Consider a pod[4] named Shipping, running a Java-based microservice vulnerable to remote code execution (RCE). The pod is configured with a *read-only* root filesystem and has *Seccomp* enabled, but operates in *privileged* mode, a misconfiguration widely observed in real-world deployments [16,17].

Attack description. Figure 1 illustrates the attack scenario. The attacker begins by verifying that the pod's root filesystem is mounted as *read-only* and confirms the presence of elevated privileges by inspecting the effective capability bitmask. Exploiting the privileged mode, the attacker remounts the root filesystem with write permissions, fetches a Dirty Pipe exploit from a remote server, modifies its permissions to enable execution, and executes the payload, resulting in unauthorized access to the underlying host's `/etc/passwd` file.

Limitations of existing controls. Despite Seccomp being enabled and the root filesystem mounted as *read-only*, the pod operates in privileged mode—a configuration that disables Seccomp confinement and permits remounting the root filesystem as writable, thereby bypassing immutability enforcement. Consequently, arbitrary code execution remains feasible even in the presence of preventive security mechanisms.

[4] In Kubernetes, a pod may consist of one or more containers.

```
# check mounts
shipping:/# cat /proc/mounts
overlay / overlay ro,relatime,lowerdir=/var/lib/..
# list capabilities
shipping:/# cat /proc/self/status
CapPrm: 0000003fffffffff
# mount filesystem as writable
shipping:/# mount -o remount,rw /
# download diretypipe payload and execute it
shipping:/# wget http://<attacker-IP>/dirtypipe
shipping:/# chmod +x dirtypipe
shipping:/# ./dirtypipe /etc/passwd
```

Fig. 1: **Pod to node escape via Dirty Pipe [18].**

2.2 Semantic Gaps

Scenario. Consider a pod named Cart, running a PHP-based application vulnerable to remote code execution (RCE). The pod is configured as non-privileged, with a read-only root filesystem and Seccomp enabled. However, the pod runs as root (i.e., UID 0)—a common default inherited from base images and frequently seen in production deployments [19].

Attack description. As illustrated in Figure 2, given that both Seccomp and the read-only filesystem security feature remain enforced in this setup, the attacker changes the working directory to `/dev`, and uses it to stage and execute the `Scarleteel` malware. This allows scanning for exposed credentials and secrets, ultimately enabling lateral movement within the cluster.

```
# change directory to /dev and execute malware
cart:/# cd /dev
# download scarleteel payload and execute
cart:/dev# wget http://<attacker-IP>/scarleteel
cart:/dev# chmod +x scarleteel
cart:/dev# ./scarleteel
```

Fig. 2: **Scarleteel [20] execution.**

Limitations of existing controls. Even in non-privileged configurations, attackers can bypass existing hardening controls. Although the root filesystem is mounted as *read-only*, writable paths remain, such as `/dev`, which is a `tmpfs`

mount provisioned by the container runtime. This memory-backed region enables the attacker to download and execute malicious payloads without altering the immutable root filesystem. While fine-grained Seccomp profiles can reduce the syscall attack surface, they remain susceptible to abuse through legitimate application behavior. Specifically, during initialization, the CART microservice executes syscalls such as `execve` to launch *apache2-foreground*, `chmod` to adjust file permissions, and network-related syscalls such as `socket`, `connect`, and `recvmsg` to retrieve configuration and initialize services. While these syscalls are legitimate and permitted by the active Seccomp profile, they are later repurposed by the attacker to fetch, prepare, and execute malware. Consequently, the malicious activity proceeds without triggering any policy violations.

We address this gap with a hardening mechanism that remains effective under elevated privileges, withstands evasion techniques, and operates without prior knowledge of container behavior. CONLOCK identifies the primary microservice binary during startup and determines when the container reaches a steady execution state. At that point, it purges all remaining filesystem-backed executables. In our attack scenarios, CONLOCK identifies `/usr/sbin/php` and `/usr/sbin/java` as the active binaries for the CART and SHIPPING microservices, respectively, and eliminates all others from the pod's filesystem, rendering the execution of auxiliary binaries at runtime infeasible.

We emphasize that the attack scenarios discussed in this section are not synthetic, but rather reflect realistic container compromises, reproduced from publicly available proof-of-concept (PoC) exploits and grounded in common misconfigurations observed in real-world deployments [17]. Moreover, our experiments in Section 6 show that existing hardening mechanisms fail to block additional post-compromise techniques, underscoring the need for a more robust, deployment-agnostic hardening approach like ours.

3 Preliminary Assessment

To assess the feasibility of execution surface reduction, we conduct a two-part preliminary study. First, we analyze the execution characteristics of pod attacks to determine whether adversaries invoke binaries beyond the main microservice binary. Second, we examine pod startup behavior to identify a convergence point in the lifecycle at which the execution surface stabilizes, enabling the safe enforcement of binary purging without disrupting initialization activity.

3.1 Understanding Pod Attacks

Pod-based attacks commonly involve code execution within the pod to carry out malicious actions [21]. We classify these attacks into three categories: *initial access*, where the adversary establishes a foothold inside the pod; *in-pod attacks*, which remain confined to the pod while abusing internal resources or exfiltrating data; and *escape-based attacks*, which attempt to break isolation and access the underlying host system. Based on publicly available PoC exploits observed in

the wild, we select five attacks: one representative of initial access and two from each of the remaining categories. We analyze how each attack interacts with the pod runtime, with particular emphasis on the executed binaries.

As shown in Table 1, pod attacks routinely depend on auxiliary binaries prepackaged in the container image. Initial access is typically achieved through shell-based vectors, resulting in the execution of shell interpreters (e.g., `/bin/sh`). In subsequent stages, attacks such as `Lucifer` and `Kinsing`, known for cryptojacking and DDoS activity, invoke binaries such as `wget` to retrieve remote payloads and `chmod` to enable execution. Some rely on binaries such as `tar` or `apt` to unpack archives or install additional components. Escape-based attacks, in addition to reusing the same primitives, frequently invoke system-level utilities such as `mount` to manipulate namespaces and break isolation. These examples demonstrate that pod attacks consistently rely on auxiliary binaries present in the container environment, reinforcing the effectiveness of binary purging as a defense. While our analysis covers a few attacks, the evaluation in Section 6 confirms that this reliance persists across a broader range of attacks.

Table 1: Binaries Invoked in Pod Attacks. `Download`: fetch tools (e.g., `curl`); `Permission`: access control utilities (e.g., `chmod`); `System`: general-purpose tools (e.g., `ps`, `mount`); `Filesystem`: file and directory manipulation tools (e.g., `mkdir`); `Shell Interpreter`: command-line shells (e.g., `sh`, `bash`).

Category	Attack	Types of Binaries				
		Download	Permission	System	Filesystem	Shell Interpreter
Initial Access	`Shell Access`	○	○	○	○	●
In-pod	`Lucifer`	●	●	○	●	●
	`Kinsing`	●	●	●	●	●
Escape	`Dirty Pipe`	●	●	●	○	●
	`CVE-2022-0492`	○	●	●	●	●

3.2 Analysis of Pod Startup Behavior

Although microservices typically execute a single long-lived binary, pods often invoke short-lived setup utilities during initialization to perform essential tasks such as permission adjustments, environment configuration, and health checks. These transient executions are critical to correct startup behavior and must not be disrupted. We analyze startup behavior across several representative microservices-based applications to identify a reliable enforcement point for safely applying execution surface reduction.

Experiment setup. We conduct our preliminary analysis on three representative microservices-based applications: BookInfo[22], Sock-shop[23], and Mu-shop[24]. Collectively, these applications comprise over 20 microservices built with diverse languages, frameworks, and execution environments, forming a representative basis for our analysis. All applications are deployed on a single-node

Kubeadm[25] cluster running *Ubuntu 24.04.1 LTS*, with *containerd* [26] as the underlying container runtime. We ran each application continuously for six hours under realistic workload conditions. To simulate user interactions and drive typical microservice behavior, we automated traffic generation using `Selenium` [27], `Locust` [28], and `Curl` [29]. Throughout execution, we monitored each microservice from initialization to steady state, capturing all executable invocations by tracing the `execve` and `execveat` syscalls. We collected these execution traces using `Sysdig` [30], a widely used syscall tracing tool.

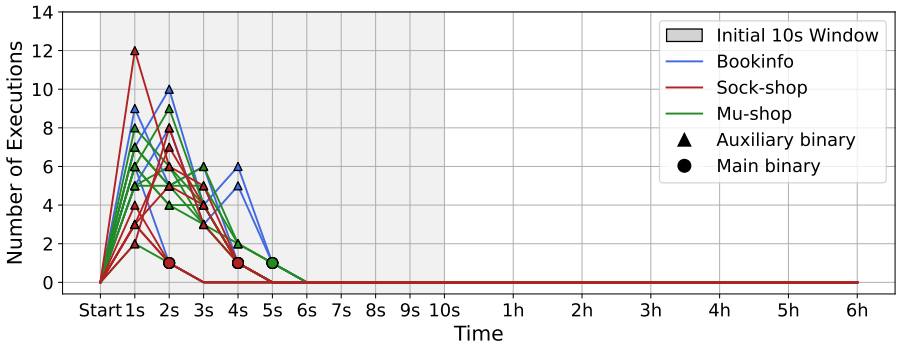

Fig. 3: **Execution Patterns in Microservices Over Time.**

Table 2: Time Gap Between Main Binary Execution and PodReady. (Min–Max per Application).

Application	**Min Duration**	**Max Duration**
Bookinfo	0.062s	0.171s
Sock-shop	0.092s	179.141s
Mu-shop	0.147s	28.185s

As shown in Figure 3, all microservices exhibit a consistent and structured startup sequence. Execution begins with the container runtime, typically *runc*[31], which creates and manages the container process, followed by the launch of the pause binary—a minimal, long-running process deployed by Kubernetes to hold the pod's network and IPC namespaces[32]. After this initialization, a short burst of auxiliary binaries is invoked, typically by image-level entrypoint scripts to prepare the pod runtime environment. These include shell interpreters and core utilities used for environment setup, filesystem configuration, and system checks. Notably, the main microservice binary is launched at the end of this transient phase, after which no further binaries are invoked, indicating that the pod has reached a stable execution state.

While the main binary's execution signals the end of initialization, detecting this boundary via syscall tracing is intrusive and unsuitable for production. To provide lightweight and externally observable enforcement points, we instead rely on Kubernetes lifecycle signals. Specifically, we use the `PodReady` condition [33], which is set once the pod passes its readiness checks. As shown in Table 2, the `PodReady` timestamp consistently follows the launch of the main microservice binary, reflecting the expected ordering where initialization completes before the pod is marked ready to serve traffic. Notably, the delay between binary execution and readiness assertion varies considerably—from milliseconds to several minutes—depending on readiness probe configurations such as the `initialDelaySeconds` parameter in the pod specification. To address cases with minimal delay, security teams may optionally introduce a short enforcement grace period, ensuring execution-surface reduction does not interfere with residual initialization activity.

4 Threat Model

We target microservices-based pods deployed in Kubernetes, where each microservice adheres to the single-responsibility principle and executes a single, well-defined task. We exclude microservices instrumented with monitoring agents, sidecars, or runtime hooks, as these introduce auxiliary executables that violate the assumption of a minimal and stable execution context central to the microservice model.

We aim to prevent unauthorized code execution within the pod, whether triggered during initial access (e.g., spawning a shell via a remote code execution vulnerability) or during post-compromise phases involving auxiliary binaries for malware deployment or pod escape. Although attackers could theoretically exploit memory corruption to issue malicious syscalls without invoking binaries, we exclude such cases from our threat model. Contemporary microservices are predominantly implemented in memory-safe languages (e.g., *Go*, *Java*, *Python*) that mitigate low-level vulnerabilities, making memory corruption-based attacks uncommon in practice. We further assume trusted container images at deployment, excluding supply chain attacks with pre-injected malicious binaries.

Our system is a startup-time hardening mechanism that prevents unauthorized execution by purging all executables other than the main microservice binary in containerized microservices. It operates entirely outside the container, requiring no in-pod instrumentation, syscall hooking, or kernel-level modifications. We enforce execution-surface reduction at the `PodReady` state, before the application serves traffic, thereby ensuring adversaries inside the pod cannot interfere with the surface reduction process.

5 Approach

We present ConLock, a startup-phase hardening mechanism that prevents unauthorized code execution in containerized microservices. The approach is

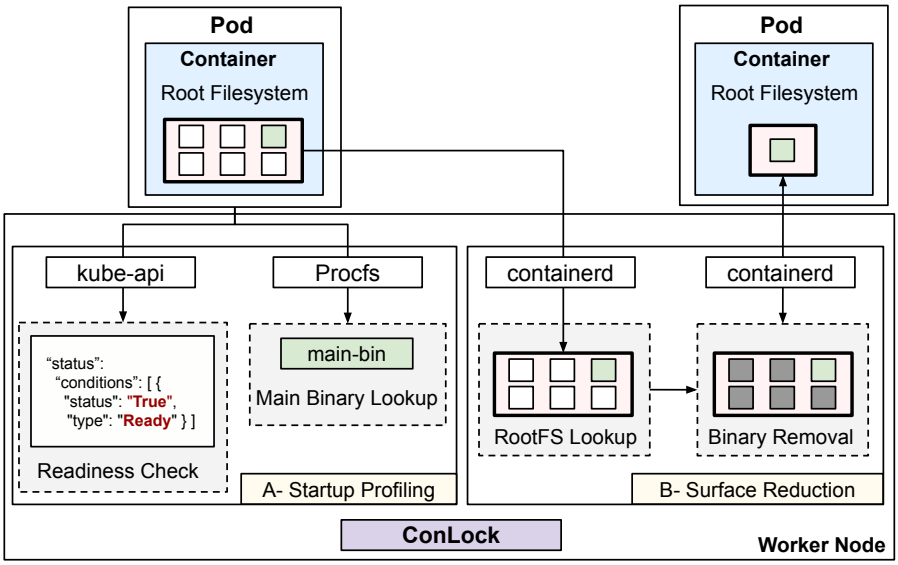

Fig. 4: **ConLock Overview.** At the *PodReady* state, we removes all filesystem-based binaries except the identified main microservice binary, making it infeasible to execute any auxiliary binaries.

motivated by two key observations: (1) microservices typically execute a single long-running binary as their core application logic [34,14], and (2) pod-based attacks commonly involve code execution, relying on auxiliary binaries prepackaged in the pod environment [21,15]. Motivated by these observations, ConLock enforces execution surface reduction as a startup-phase hardening mechanism.

Figure 4 shows the architecture of ConLock, which operates in two phases: (Ⓐ) startup profiling and (Ⓑ) execution-surface reduction. In the profiling phase, ConLock tracks the pod lifecycle to detect readiness and identify the primary long-lived binary. Once the pod reaches the `PodReady` state, ConLock removes all remaining filesystem-resident binaries, retaining only the identified main binary. This eliminates auxiliary execution paths and prevents unauthorized code execution.

5.1 Startup Profiling

Informed by our preliminary assessment, the `PodReady` signal consistently marks the end of the initialization phase, after which no auxiliary binaries are executed. This transition provides a reliable and externally observable enforcement point for execution surface reduction. In its first phase, ConLock performs two key tasks: verifying pod readiness and identifying the main microservice binary.

Readiness Check. ConLock determines pod readiness by querying the Kubernetes control plane and inspecting pod status conditions via the API server.

Specifically, it checks whether the `PodReady` condition is set to `True`, indicating that the pod has passed its readiness checks and is ready to serve traffic.

Main Binary Lookup. Once readiness is confirmed, ConLock identifies the pod's primary executable through container-level introspection. It first extracts the container ID, then queries the container runtime via the Container Runtime Interface (CRI) [35], which exposes metadata over gRPC [36]. Using this metadata, ConLock retrieves the container's main process ID (PID) and resolves the absolute path of the running executable by dereferencing the symbolic link at `/proc/<PID>/exe`. This path reliably corresponds to the long-lived binary that remains active throughout steady-state execution.

5.2 Surface Reduction

After confirming pod readiness and identifying the main binary, ConLock transitions to the surface reduction phase. In this stage, ConLock removes auxiliary executables from the pod's environment, retaining only the primary binary required for core functionality. This process involves two steps: (1) resolving the container's root filesystem path from the host, and (2) pruning non-essential binaries to minimize the pod's execution surface.

Root Filesystem Lookup. ConLock begins by resolving the container ID of the target pod and locating its merged root filesystem under `/run/containerd/io.containerd.runtime.v2.task/k8s.io/<containerid>/rootfs`. This path is exposed by *containerd* as part of its task supervision infrastructure: for each container, *containerd* maintains a mount namespace and projects the container's merged root filesystem to the host to support lifecycle operations such as checkpointing and debugging [37]. ConLock leverages this host-visible path to access the container's complete runtime view, enabling lightweight and accurate surface execution reduction without relying on in-pod instrumentation.

Binary Removal. With access to the pod's full root filesystem, ConLock performs targeted pruning of the execution surface by removing unnecessary binaries. It begins by parsing shell configuration files (e.g., `/etc/profile`) to extract directories listed in the command search path (`$PATH`), which typically contain auxiliary binaries bundled with the base image. ConLock then traverses each of these directories and removes all executable files, excluding the previously resolved main binary. This selective removal eliminates latent execution vectors while preserving the pod's intended functionality.

6 Evaluation

We developed ConLock as a lightweight shell-based tool and evaluated its effectiveness across a range of pod attack scenarios. Our analysis includes comparisons with existing security mechanisms such as Seccomp and the read-only filesystem security feature, as well as measurements of execution time under realistic workloads to demonstrate its practicality in microservices environments.

6.1 Experimental setup.

Table 3: Pod Attack Scenarios.

Category	Attack Scenario	Application
Initial Access	`Shell Access`	Google Online Boutique Bank-of-Anthors AKS Store Retail Store Martian Bank
In-pod Attacks	`Sysrv-Hello` [38]	Google Online Boutique
	`Kinsing` [39]	Google Online Boutique
	`RebirthLtd` [40]	Bank-of-Anthors
	`Kangaro` [41]	Bank-of-Anthors
	`Shellbot` [42]	AKS Store
	`Mirai` [43]	AKS Store
	`SCARLETEEL` [20]	Retail Store
	`Lucifer` [44]	Retail Store
	`LABRAT` [45]	Martian Bank
	`TeamTNT` [46]	Martian Bank
Pod Escape Attacks	`CVE-2022-0492`	Google Online Boutique
	`Core_Pattern Abuse`	Google Online Boutique
	`Kernel Module Injection`	Bank-of-anthors
	`Device Handler Abuse`	Bank-of-Anthors
	`Host_PID Abuse`	AKS Store
	`Host_NET Abuse`	AKS-Store
	`Host Process Debugging`	Retail-Store
	`Host_FS Mount`	Retail Store
	`CVE-2016-5195`	Martian Bank
	`CVE-2019-5736`	Martian Bank

Experimental setup. We evaluate ConLock on five representative microservices applications maintained by major cloud vendors and used in recent research [10,52]. These applications feature polyglot architectures spanning multiple languages and frameworks (e.g., Node.js, Java, Python, Go), and collectively comprise 33 microservices, as detailed in Table 4. This diversity allows us to evaluate ConLock across heterogeneous runtime environments. All applications are deployed on a single-node *kubeadm* cluster with 16 GB of RAM, running *Ubuntu 20.04.6 LTS* and *containerd*.

We evaluate ConLock under two operational modes: normal and adversarial. In normal mode, we simulate realistic user activity to drive typical microservice behavior. For applications with built-in workload drivers such as Google Online Boutique's `loadgenerator` [47], we use them directly. For the remaining applications, we employ a combination of *Locust* and a custom Selenium-

Table 4: Evaluated Microservices-based Applications.

Name	# of microservices	Polyglot	Maintainer
Google Online Boutique [47]	10	✓	Google Cloud
Bank-of-Anthors [48]	6	✓	Google Cloud
AKS Store [49]	5	✓	Microsoft Azure
Retail Store [50]	5	✓	Amazon AWS
Martian Bank [51]	8	✓	Cisco

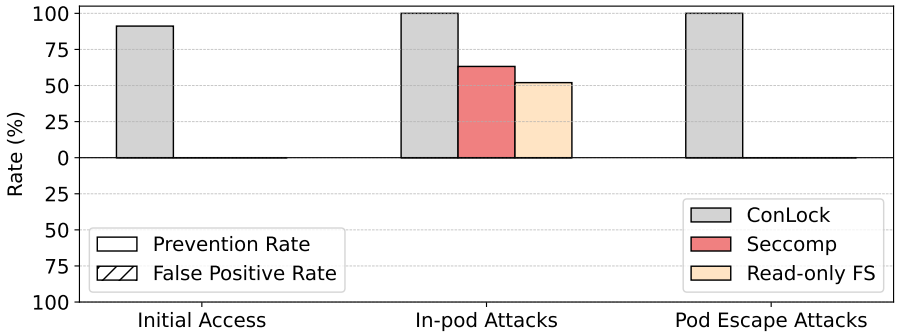

Fig. 5: **Assessment of ConLock Performance.**

based autoclicker to generate continuous, user-like traffic. This ensures representative and sustained workloads across all evaluated microservices. In adversarial mode, we simulate pod attacks spanning initial access, in-pod execution, and pod escape. We leverage publicly available PoC exploits [39,42] to reproduce known techniques observed in real-world container intrusions [20]. These attack scenarios are summarized in Table 3.

Evaluation dataset. In normal mode, each microservice is monitored continuously over a ten-hour period. Since ConLock enforces execution restrictions, we define false positives as any unintended disruption to benign behavior, measured in terms of pod crashes and restarts during this period. In attack mode, conducted separately from normal monitoring, we execute 20 attack runs per microservice, with one distinct attack scenario selected per application.

6.2 Performance Evaluation

Figure 5 presents the performance of ConLock across diverse attack scenarios and microservices-based applications. Our approach achieves a 100% prevention rate against both in-pod and pod escape attacks. These attacks consistently depend on auxiliary binaries such as downloaders, permission modifiers, and

system utilities to execute malicious behavior beyond the scope of the main microservice binary. By removing these binaries, ConLock effectively eliminates such attack vectors. In contrast, ConLock achieves a 91.11% prevention rate for initial access scenarios involving shell-based entry. The remaining 8.89% corresponds to three microservices in the Bank of Anthos application—`frontend`, `contacts`, and `userservice`—where the main binary is the shell interpreter `/usr/bin/dash`. Because ConLock preserves this as the primary executable, shell access remains possible post-initialization. Yet, the attack surface is significantly reduced: without auxiliary binaries, the shell environment lacks the tooling required for post-compromise operations. As a result, attackers are constrained to shell built-in commands, which offer limited capabilities [10], significantly reducing the impact of the compromise.

Importantly, ConLock incurs no false positives. Throughout ten hours of continuous monitoring, all microservices executed without disruption, and no container restarts were observed.

6.3 Comparison to Existing Hardening Mechanims

We compare ConLock against two widely adopted hardening techniques: read-only filesystems and Seccomp. For the former, we configure pods to mount container filesystems as read-only via the pod specification. For Seccomp, we generate a custom profile for each microservice by tracing syscalls over a three-hour period under realistic workloads, collecting a whitelist of syscalls.

Read-only Filesystem. Figure 5 shows that read-only filesystem enforcement provides only limited protection: it blocks write operations but does not restrict code execution. In initial access scenarios, shell-based interaction remains unaffected since it requires no writes. In escape attacks, adversaries leverage excessive capabilities to remount the root filesystem as writable, nullifying the restriction. For in-pod attacks, 52.16% are partially blocked, not because of execution control but because attackers attempt to install auxiliary tooling via package managers, which fail under read-only constraints due to their reliance on write operations to internal state and cache directories. Notably, no false positives occur during normal application execution.

Seccomp. As shown in Figure 5, Seccomp exhibits limited prevention capabilities across the evaluated attack scenarios. In initial access cases, it fails to block shell-based entry, as the `execve` syscall—responsible for spawning shells—is typically permitted to support pod startup. In escape-based attacks, enforcement remains largely ineffective due to the privileged access that allows unrestricted syscall execution. For in-pod attacks, Seccomp blocks 63.12% of in-pod attacks due to their invocation of non-whitelisted syscalls. Although no false positives were observed during evaluation, this outcome required a dedicated three-hour training period per microservice to capture representative syscall activity, highlighting the significant overhead involved in constructing reliable and non-disruptive profiles.

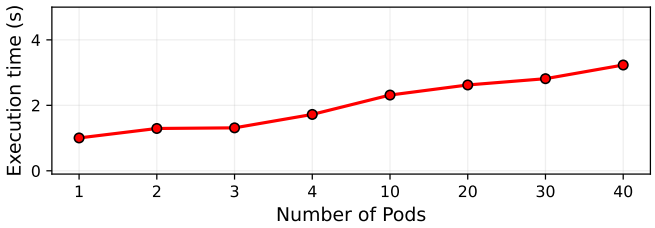

Fig. 6: **Avg. execution time of ConLock across varying pod densities.**

ConLock outperforms existing hardening mechanisms by leveraging a key property of microservices: they execute on a single, long-lived binary. By preserving only this binary post-initialization, ConLock enforces execution surface reduction without training, in-pod instrumentation, or syscall hooking. Moreover, its external enforcement at startup time ensures resilience against tampering while preserving normal microservice functionality.

6.4 Execution Time

We assess ConLock's execution time by measuring the duration required to carry out its core stages: main binary lookup and execution surface reduction. This includes container-level introspection to resolve the container's process ID and dereference the symbolic link `/proc/<PID>/exe`, followed by host-level access to the container's merged root filesystem and selective removal of auxiliary binaries. To reflect realistic deployment densities, we evaluate ConLock across varying scales, from a single pod up to 40 pods, by replicating microservices accordingly [53]. All experiments are conducted on a *kubeadm* worker node equipped with an Intel Core i7-10850H CPU (4 cores, 2.7 GHz) and 16GB RAM.

ConLock 's execution time is shaped by two key factors: the number of active pods and the cost of per-pod introspection and filesystem modification. For each pod, ConLock queries the Kubernetes API to assess readiness and retrieve container metadata, identifies the main binary through process-level introspection, and performs execution surface reduction by accessing the container's merged root filesystem and removing auxiliary executables. Although these operations introduce network and processing overhead, ConLock maintains low latency even at high pod densities. Over 10 measurement runs, ConLock completes the full process—spanning pod readiness assessment, main binary identification, and execution surface reduction—in almost one second for a single pod and approximately three seconds for 40 pods, as shown in Figure 6. These results reflect ConLock 's sublinear scaling behavior, underscoring its efficiency even as pod count increases. Moreover, because all operations are confined to the startup phase, ConLock introduces no overhead during steady-state execution, making it highly practical for production environments.

7 Discussion

Despite its effectiveness in preventing pod attacks, ConLock is not without limitations.

Incompatibility with Instrumented Microservices. ConLock assumes that microservices follow a single-responsibility execution model, where only the main binary remains active post-initialization. However, some production pods embed auxiliary executables (e.g., sidecar agents, configuration reloaders, or custom health probes) that periodically execute binaries as part of their normal operation. Enforcing strict execution immutability in these cases may disrupt expected behavior, leading to functionality loss or instability.

Debugging and Incident Response. By design, ConLock restricts binary execution once the pod reaches its ready state, effectively preventing ad-hoc code execution. While this enforces a strong security posture, it can hinder operational tasks such as debugging, live forensics, and incident remediation. As a workaround, instead of permanently removing binaries for execution surface reduction, security teams can mount an empty directory over executable paths during startup, masking access. If runtime access is later required for debugging or incident response, the original directories can be remounted on demand. This approach preserves the security benefits while remaining reversible, yet it introduces additional operational complexity.

Evasion Attacks. ConLock reduces the container's attack surface by eliminating auxiliary binaries, based on the assumption that malicious activity entails spawning new executables. Yet, attacks that exploit memory corruption or trigger remote code execution (RCE) within the address space of the main binary can evade this restriction. We acknowledge that such in-memory evasions constitute an inherent limitation of our approach.

8 Related Work

Runtime Hardening. Mechanisms such as Seccomp[12], AppArmor[11], and the read-only filesystem security feature are commonly used to restrict pod behavior at runtime. However, they often break down in over-permissive configurations such as privileged pods or those granted excessive capabilities, where enforcement becomes ineffective or is bypassed entirely. Even in constrained environments, these mechanisms rely on prior knowledge of container behavior. Enforcing a read-only filesystem requires developers to anticipate all legitimate write operations and explicitly configure ephemeral volumes. Similarly, applying Seccomp and AppArmor demands that DevOps teams identify benign syscalls and file access paths to avoid disrupting application functionality. While solutions such as Confine [54] automate Seccomp policy generation, they still depend on developer input, such as `exec` calls. In contrast, ConLock requires no training or DevOps involvement. It infers enforcement boundaries directly from observed startup behavior, enabling robust and adaptive execution surface reduction across diverse deployment scenarios.

Static Hardening. These techniques reduce the attack surface at build time by minimizing dependencies and limiting available tooling. Common practices include using minimal base images (e.g., Alpine), adopting multi-stage builds to exclude development artifacts, and explicitly removing unnecessary system utilities in the Dockerfile. At the orchestration level, YAML-based configurations such as setting restrictive `securityContext` fields (e.g., dropping Linux capabilities, enforcing non-root execution, or disabling privilege escalation) further constrain runtime behavior [55]. While these methods are effective when applied rigorously, they rely on DevOps discipline and consistent security hygiene throughout the CI/CD pipeline. In contrast, ConLock performs runtime-driven hardening without requiring any DevOps involvement. It enforces execution surface reduction based on observed behavior after initialization, independent of image composition or configuration settings.

Distroless Images. These images [56] reduce the attack surface by omitting general-purpose tools such as shells, package managers, and other auxiliary binaries from the base image. This static hardening strategy limits post-compromise capabilities by depriving adversaries of commonly exploited utilities. However, distroless images remain uncommon in production microservices due to operational limitations. In particular, the absence of a package manager complicates dependency management, making it difficult to adapt images to evolving application requirements [57]. In contrast, ConLock applies execution surface reduction dynamically at runtime, without requiring changes to image composition or interfering with microservice logic.

9 Conclusion

We presented ConLock, a pod hardening mechanism that requires no prior behavioral knowledge and generalizes across diverse deployment environments. ConLock identifies the main microservice binary at startup and purges all non-essential executables from the pod's filesystem, depriving adversaries of auxiliary tooling commonly leveraged in post-compromise stages. Our evaluation across representative microservices-based applications and diverse attack scenarios shows that ConLock prevents 99.57% of attacks with zero false positives, all without requiring prior knowledge of the pod or introducing significant overhead, offering a practical and effective defense for containerized microservices.

References

1. Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and software technology*, 131:106449, 2021.
2. Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. Microservices: architecture, container, and challenges. In *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)*, pages 629–635. IEEE, 2020.

3. Vitor Goncalves da Silva, Marite Kirikova, and Gundars Alksnis. Containers for virtualization: An overview. *Appl. Comput. Syst.*, 23(1):21–27, 2018.
4. Dougla Nigel. Ephemeral Containers and APTs. `https://sysdig.com/blog/ephemeral-containers-and-apts/`, 2024.
5. Sysdig. Falco: container native runtime security. `https://falco.org/`, 2022.
6. Aquasec. Aqua tracee: Runtime ebpf threat detection engine. `https://www.aquasec.com/products/tracee/`.
7. 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.
8. Amr S Abed, Charles Clancy, and David S Levy. Intrusion detection system for applications using linux containers. In *Proceedings of Security and Trust Management (STM)*, 2015.
9. 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.
10. 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.
11. Kubernetes. Restrict a Container's Access to Resources with AppArmor. `https://kubernetes.io/docs/tutorials/security/apparmor/`.
12. Kubernetes. Restrict a Container's Syscalls with seccomp. `https://kubernetes.io/docs/tutorials/security/seccomp/`.
13. Kubernetes. Configure a Security Context for a Pod or Container. `https://kubernetes.io/docs/tasks/configure-pod-container/security-context/`.
14. Sysdig. Automated falco rule tuning. `https://sysdig.com/blog/falco-rule-tuning/`.
15. Surana Nitesh. How Malicious Actors Abuse Native Linux Tools in Attacks. `https://www.trendmicro.com/en_us/research/22/i/how-malicious-actors-abuse-native-linux-tools-in-their-attacks.html`.
16. Ozeren Sila. The Ten Most Common Kubernetes Security Misconfigurations & How to Address Them. `https://www.picussecurity.com/resource/author/s%C4%B1la-ozeren`, 2024.
17. Sysdig. Sysdig 2021 Container Security and Usage Report. `https://sysdig.com/content/c/pf-2021-container-security-and-usage-report?x=u_WFRi`, 2021.
18. NVD. CVE-2022-0847 Detail. `https://nvd.nist.gov/vuln/detail/cve-2022-0847`.
19. Sysdig. Sysdig 2024 Cloud-Native Security and Usage Report. `https://sysdig.com/2024-cloud-native-security-and-usage-report/`, 2024.
20. Sysdig. SCARLETEEL: Operation leveraging Terraform, Kubernetes, and AWS for data theft. `https://sysdig.com/blog/cloud-breach-terraform-data-theft/`.
21. Sysdig. Preventing container runtime attacks with Sysdig's Drift Control. `https://sysdig.com/blog/preventing-runtime-attacks-drift-control/`, 2022.
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. Mushop. Oracle cloud infrastructure. https://github.com/oracle-quickstart/oci-cloudnative.
25. Kubernetes. Kubeadm. https://kubernetes.io/docs/reference/setup-tools/kubeadm/.
26. Containerd. An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.
27. Selenium. Selenium automates browsers. That's it! https://www.selenium.dev/.
28. Locust. Locust: An open source load testing tool. https://locust.io/.
29. Curl. Curl. https://curl.se/.
30. Sysdig. Secure DevOps Platform. https://github.com/draios/sysdig.
31. Hykes Solomon. Introducing runC: a lightweight universal container runtime. IntroducingrunC:alightweightuniversalcontainerruntime.
32. Kumar Rajesh. What is Pause container in Kubernetes? https://www.devopsschool.com/blog/what-is-pause-container-in-kubernetes/.
33. Kubernetes. Pod Lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, 2024.
34. Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, 2021.
35. Kubernetes. Container Runtime Interface (CRI). https://kubernetes.io/docs/concepts/architecture/cri/.
36. gRPC. gRPC: A high performance, open source universal RPC framework. https://grpc.io/.
37. Suda Akihiro. The internals and the latest trends of container runtimes (2023). https://medium.com/nttlabs/the-internals-and-the-latest-trends-of-container-runtimes-2023-22aa111d7a93.
38. Sysdig. THREAT ALERT: Crypto miner attack – Sysrv-Hello Botnet targeting WordPress pods. https://sysdig.com/blog/crypto-sysrv-hello-wordpress/, 2021.
39. Sysdig. Zoom into Kinsing. https://sysdig.com/blog/zoom-into-kinsing-kdevtmpfsi/, 2020.
40. NVD. DDoS-as-a-Service: The Rebirth Botnet. https://sysdig.com/blog/ddos-as-a-service-the-rebirth-botnet/, 2024.
41. 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.
42. Sysdig. Malware analysis: Hands-On Shellbot malware. https://sysdig.com/blog/malware-analysis-shellbot-sysdig/, 2021.
43. Nitzan Yaakov. Tomcat Under Attack: Exploring Mirai Malware and Beyond. https://www.aquasec.com/blog/tomcat-under-attack-investigating-the-mirai-malware/, 2023.
44. 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/.
45. Miguel, Hernández. LABRAT: Stealthy Cryptojacking and Proxyjacking Campaign Targeting GitLab. https://sysdig.com/blog/labrat-cryptojacking-proxyjacking-campaign/.
46. Ofek, Itach and Assaf, Morag. TeamTNT Reemerged with New Aggressive Cloud Campaign. https://www.aquasec.com/blog/teamtnt-reemerged-with-new-aggressive-cloud-campaign/.

47. Google Cloud Platform. Google Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.
48. Google Cloud. Running distributed services on GKE private clusters using Cloud Service Mesh. https://cloud.google.com/service-mesh/docs/distributed-services-private-clusters, 2019.
49. Azure. Aks store demo. https://github.com/Azure-Samples/aks-store-demo.
50. AWS-Containers. Aws containers retail sample. https://github.com/aws-containers/retail-store-sample-app.
51. Cisco. MARTIAN BANK. https://github.com/cisco-open/martian-bank-demo.
52. Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3971–3988, 2021.
53. Sysdig. Sysdig 2023 cloud-native security and usage report. pages 1–29, 2023.
54. Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis 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.
55. Kubernetes. Configure a Security Context for a Pod or Container. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.
56. Goderre Laurent. Is your container image really distroless? https://www.docker.com/blog/is-your-container-image-really-distroless/.
57. Inoio. Less but better? Distroless container images. https://inoio.de/blog/2024/06/14/distroless/.