ImmuCheck: Selective Immutability for Container Escape Detection in Microservices

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

Amina Bassit
Mobai
amina.bassit@mobai.bio

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

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

Abstract

Container escape attacks break isolation boundaries, granting threat actors code execution on the underlying host and potentially full control over the entire cluster. Existing runtime defenses exhibit an inherent trade-off. Anomaly- and provenance-based detection mechanisms achieve broad escape detection, yet incur substantial operational costs due to model retraining requirements or system-wide provenance capture. In contrast, industry rule-based detectors avoid these costs but offer limited detection coverage.

We introduce a container escape detection approach that breaks this trade-off, offering broad detection coverage without operational costs. Our key insight is that container escape attacks commonly involve filesystem writes, violating the *immutable* state expected in microservices. While enforcing strict immutability can expose such behavior, microservices sometimes perform legitimate writes for ephemeral operations (e.g., logging), making blanket immutability impractical. Our system, ImmuCheck, instead applies *selective immutability*, automatically inferring legitimate write regions during initialization and flagging subsequent writes outside these regions as potential escape attempts. We evaluate our approach across 12 escape scenarios spanning five microservices-based applications from major cloud providers. ImmuCheck achieves 99.22% precision and 100% recall, with low overhead, no retraining requirements, and timely detection.

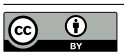
CCS Concepts

• **Security and privacy → Intrusion detection systems**.

Keywords

Container Escape Detection; Microservices Security; Kubernetes Security; Training-less Anomaly Detection

ACM Reference Format:
Asbat El Khairi, Amina Bassit, Andreas Peter, and Andrea Continella. 2026. ImmuCheck: Selective Immutability for Container Escape Detection in Microservices. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 01–05, 2026, Bangalore, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779208.3807484



ASIA CCS '26, Bangalore, India

ACM ISBN 979-8-4007-2356-8/2026/06
https://doi.org/10.1145/3779208.3807484

1 Introduction

As organizations prioritize agility and scalability, microservices have emerged as the dominant architectural paradigm, decomposing applications into small, independently deployable services. Containers, favored for their lightweight virtualization, are the standard unit for running these services [1–3]. The latest CNCF report shows that 91% of organizations now rely on containers for cloud-native applications [4]. However, this ubiquity has also made internet-facing microservices a primary entry point for adversaries targeting cloud infrastructures [5; 6]. Once attackers gain a foothold, they often attempt to escape the container to compromise the host and potentially the cluster. This makes timely and accurate escape detection critical in microservices-based clusters.

Container escape detection has been explored through different solutions in both industry and academia. Falco [7] and Tracee [8], widely adopted container monitoring tools in industry, enforce kernel-level security rules to detect container escape attacks. In addition, container platforms such as Kubernetes [9] offer built-in confinement mechanisms, including *seccomp* default profiles that block syscalls commonly abused in escape attacks (e.g., `finit_module` for kernel module loading) and the *read-only* root filesystem feature, which prevents unauthorized writes and is recommended by MITRE as a mitigation against container escapes [10]. On the academic front, research on container escape detection remains largely unexplored. Most existing solutions [11–14] primarily target application-layer threats while explicitly excluding escape attacks from their threat models. To the best of our knowledge, CHIDS [15] and PACED [16] are the only solutions that specifically target container escape detection in cloud environments. CHIDS detects escapes via syscall analysis using machine learning and graph models, while PACED uses provenance tracking to monitor cross-namespace events between container and host.

While existing solutions address container escape attacks, they expose a fundamental trade-off between detection coverage and practical deployment in microservices-based environments. On one hand, hardening mechanisms and rule-based detectors offer lightweight and readily deployable defenses, but their protection scope is inherently limited — either ineffective in permissive or misconfigured environments, or constrained by static rules that fail to cover all escape vectors and are prone to novel or stealthy techniques. On the other hand, more sophisticated detection systems address this coverage gap but introduce new challenges. Provenance-based monitoring, as in PACED [16], detects container escapes by tracking fine-grained data flows, but incurs high system overhead, limiting both

scalability and real-time use. Similarly, training-based approaches, such as CHIDS [15], improve detection accuracy by modeling normal application behavior and flagging deviations. However, they suffer from quick model aging and require repeated retraining cycles to remain effective in evolving microservices-based environments [13]. In essence, container escape detection remains an open challenge, as existing solutions either achieve limited coverage with low operational overhead or provide broader coverage at the cost of operational complexity, limiting their adoption in practice.

In this work, we challenge this trade-off and show that strong container escape detection can be achieved without incurring prohibitive operational cost in microservices-based environments. Our approach leverages a fundamental property of microservices: their *stateless* execution model, which promotes scalability and replication by externalizing application state to services such as databases, caches, and message queues [17]. As a result, filesystem writes during normal execution are confined to a small, fixed set of locations that are deterministically established during container initialization by startup scripts, runtime environments, or base image defaults, and are repeatedly accessed for auxiliary purposes such as logging, temporary files, or runtime metadata. This yields filesystem activity that is limited in scope and predictable in structure. In contrast, container escape attacks deviate from this execution model by introducing filesystem writes outside these expected locations, a behavior consistently observed in publicly available proof-of-concept (PoC) escape exploits (see Section 3).

This asymmetry implies that effective container escape detection does not require modeling all filesystem behavior. Instead, it is sufficient to identify the small set of filesystem locations expected under normal execution and flag deviations from this footprint. Based on this insight, we adopt *selective immutability* as a detection principle, treating filesystem locations established during container initialization as legitimate write regions and flagging writes outside them as anomalous. We realize this principle in ImmuCheck, a system that infers legitimate filesystem regions at container startup and monitors subsequent filesystem activity relative to this baseline, entirely from the host. Deviations from the expected write footprint provide a precise signal of escape behavior with low operational overhead. To the best of our knowledge, this is the first system to provide complete host-side visibility of container filesystem activity without container instrumentation or kernel modifications, despite storage abstraction and namespace isolation.

In summary, our contributions include:

- We propose a detection system that enforces *selective immutability*, allowing legitimate writes while flagging escape-indicative writes.
- We implement ImmuCheck, a lightweight node-level monitoring approach that operates externally to the container, requiring neither prior training, system-wide monitoring, nor in-container instrumentation.
- We show that ImmuCheck effectively detects container escape attacks, achieving an average precision of 99.22% and a recall of 100% across 12 escape scenarios on five microservices-based applications from major cloud providers.

In the spirit of open science, we make ImmuCheck available at https://github.com/Asbatel/ImmuCheck.

2 Motivating Example

In this section, we present a concrete, realistic container escape attack scenario to motivate ImmuCheck.

2.1 Scenario

Consider an internet-facing pod[1], named cart, running a *php-buster* based microservice vulnerable to remote code execution (RCE). The pod runs with its root filesystem set to *read-only*, restricting write operations. However, it is deployed as *privileged*, a common misconfiguration in cloud environments [18; 19]. The attacker exploits the RCE vulnerability to obtain code execution inside the pod and subsequently attempts to escape to the node[2].

2.2 Pod Escape: Core_dump Abuse

In Linux, the kernel invokes a user-mode helper, defined in */proc/sys/kernel/core_pattern*, to process core dumps when a process crashes [20]. With privileges inside the pod, an attacker can modify *core_pattern* to redirect dumps to a malicious binary, then trigger a segmentation fault to execute code on the node and escape the pod.

```
# check mounts
cart:/# echo "$(</proc/mounts)"
overlay / overlay ro,relatime,lowerdir=/var/lib/..
# check permissions
cart:/# echo "$(</proc/self/status)"
CapPrm: 0000003fffffffff
# mount filesystem as writable
cart:/# mount -o remount,rw /
# mount the proc virtual filesystem
cart:/# mkdir newproc
cart:/# mount -t proc proc newproc
# extract the pod's root filesystem path on the host
cart:/# HP=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /proc/mounts`
# prepare the payload
cart:/# echo '#!/bin/sh' > /pl
cart:/# echo "sh -i >& /dev/tcp/<attacker-IP>/1234 0>&1" >> /pl
cart:/# chmod a+x /pl
# configure core_pattern to execute the payload
cart:/# echo "|$HP/pl" > newproc/sys/kernel/core_pattern
# trigger a segmentation fault
cart:/# sh -c 'kill -11 $$'
```

Figure 1: Pod escape via *coredump* abuse. Commands executed from within the compromised pod after obtaining initial access via the RCE vulnerability. Escape sequence derived from public PoCs [21], excluding the reconnaissance phase.

2.3 Pod Escape Attack Description

As shown in Figure 1, following initial compromise of the pod, the attacker begins with internal reconnaissance, enumerating mounted filesystems and identifying that the root filesystem is mounted as *read-only* (`ro`). Inspecting the pod's applied capabilities, they extract a bitmask of `0000003fffffffff`, which, when decoded offline, reveals an extensive privilege set. Leveraging this, the attacker remounts the filesystem with write permissions, creates a `newproc` directory, and mounts the *proc* filesystem at this location,

[1]A pod is a Kubernetes abstraction that groups one or more containers. Thus, we refer to container escape as pod escape throughout the paper.
[2]We use node and host interchangeably.

establishing an isolated interface for direct kernel parameter manipulation. Next, the attacker extracts the pod's `upperdir` from `/proc/mounts`, corresponding to the pod's writable layer on the host filesystem. They create a custom payload and resolve its host-path by appending its in-pod location to the extracted `upperdir`. The attacker then configures `core_pattern` to redirect core dumps to this host-resolvable path. Finally, they trigger a segmentation fault to invoke the core dump handler, executing the payload and spawning a reverse shell from the node to the attacker's server.

2.4 Limitations of Existing Solutions

2.4.1 *Limited Coverage*

Policy-based confinement solutions. Privileged pods run completely unconfined, making *Seccomp* and *AppArmor* ineffective. Also, a privileged pod enables an adversary to remount the root filesystem as writable, bypassing *read-only* restrictions.

Rule-based solutions. Falco fails to detect this escape technique because its default ruleset lacks a relevant rule. Tracee has a dedicated rule for this escape technique that flags any write to */proc/sys/kernel/core_pattern*. However, the attacker evades this rule by mounting a new *proc* filesystem under a newly created directory (*newproc*), isolating the namespace. Thus, modifications to *core_pattern* do not affect the original */proc*, and the escape proceeds undetected. This shows that even with custom rules, such solutions remain easily evaded by novel or altered escape techniques.

2.4.2 *Operational Overhead*

Anomaly-based solutions. CHIDS [15] detects the escape attack by flagging syscalls not seen during the training phase (e.g., `mount`). Yet, this approach suffers from rapid baseline aging in microservices environments. For example, changing the CART pod's base image from *php-buster* to *php-bullseye* triggers false positives due to low-level *glibc* changes, as shown in [13]. While frequent retraining can maintain performance, it is impractical in microservices-based environments with frequent feature rollouts [13; 22].

Provenance-based solutions. PACED [16] detects the attack by capturing a cross-namespace data flow, where the `pl` payload is created within the pod namespace and later executed in the host namespace by the kernel. Yet, PACED's reliance on provenance systems such as CamFlow [23] adds substantial storage and latency overhead, limiting its use in microservices-based environments.

We bridge this gap with a detection approach that achieves strong coverage without operational overhead. ImmuCheck applies *selective immutability* by identifying legitimate writable locations at pod startup and flagging writes beyond them as escape activity. In Cart, the pod initializes with */var/log/apache2/error.log*, from which ImmuCheck infers */var/log/apache2/* as a valid writable location. During the escape, writes outside this location, such as */run/mount* from *mount* execution, the creation of *newproc* directory, the *pl* payload, or permission changes, are flagged as escape activity.

We stress that this example is not synthetic but reflects a real pod escape attack, reproduced from a public PoC escape [21]. While *privileged* pods constitute a strong adversarial model, such configurations are prevalent in real-world deployments. According to Sysdig's analysis [19], running *privileged* pods is the second most common runtime policy violation. Moreover, equivalent levels of privilege can also be obtained through in-pod privilege escalation via kernel vulnerabilities, even when pods are initially deployed without elevated privileges [24]. Also, this is not an isolated instance: our experiments and case studies, detailed in Section 7, demonstrate that existing solutions fail to detect other escape methods, further underscoring the need for an approach like ours.

3 Pod Escape and Filesystem Writes

The set of pod escape techniques is relatively small, and public techniques consistently show that these attacks interact with the pod filesystem [21; 25]. This interaction typically manifests in three forms: (*i*) direct writes required by the escape, (*ii*) reliance on external tools, or (*iii*) side effects introduced during execution.

Direct writes. Many escape techniques rely directly on writes to the pod's filesystem. User-mode helper–based attacks (e.g., *core_pattern*) drop a binary and overwrite kernel interfaces to redirect execution. Others write payloads before execution or modify the filesystem to disable runtime protections.

Tool dependencies. Some escapes rely on external utilities when custom payloads are unavailable. For example, loading a malicious kernel module requires a custom payload invoking the `finit_module` syscall. Without such a payload, attackers resort to tools such as *kmod*, which are typically absent from common images and must be fetched at runtime, leaving traces in the pod filesystem.

Side effects. Some escape steps implicitly alter the pod's filesystem as a byproduct of execution. For example, executing the `mount` utility (a common step in many escape techniques) often creates a `/run/mount` directory.

It is noteworthy that while filesystem writes are prevalent across escape techniques, we acknowledge a few edge cases that deviate from this pattern, which we examine in §8. Although we focus on container escapes, ImmuCheck generalizes to any container threat that weaponizes filesystem manipulation as part of its execution chain (e.g., cryptojacking campaigns such as Scarleteel [26]).

4 Preliminary Assessment

Our approach builds on selective immutability, which treats deviations from expected filesystem behavior as signals of container escape activity. Applying this principle requires addressing two questions: (❶) which filesystem regions warrant monitoring, and (❷) how filesystem activity relative to these regions should be evaluated for escape detection. We address both through a preliminary assessment.

4.1 Pod Write Exposure

4.1.1 Setup. We analyze the pod filesystem under two settings: (i) the default configuration, and (ii) a highly permissive configuration in which restricted paths are deliberately remounted with write or execute permissions by an adversary. Our analysis is conducted on a *Kubeadm* cluster using *containerd*, the default container runtime in most production Kubernetes deployments.

Table 1: Pod Writable Exposure in Typical Deployments. ● unrestricted access, ◐ restricted access (e.g., format-constrained or specific files), ○ disabled access. *Adversarial* denotes a worst-case setting where restricted paths are remounted with write or execute permissions. □ shows the filesystem regions (i.e., mounts) that define the pod's write-execute surface.

Category	Type	Mounts	By Default		Adversarial	
			Write	Execute	Write	Execute
Root Filesystem	OVERLAY	/	●	●	●	●
Virtual Filesystems	SYSFS	/sys	○	○	◐	○
	PROC	/proc	○	○	◐	○
	CGROUP	/sys/fs/cgroup	○	○	○	○
	DEVPTS	/dev/pts	○	○	○	○
	MQUEUE	/dev/mqueue	○	○	○	○
	TMPFS	/dev	●	●	●	●
Bind Mounts	BIND	/dev/shm	●	○	●	●
	BIND	/etc/hostname	●	●	●	●
	BIND	/etc/hosts	●	●	●	●
	BIND	/dev/termination-log	●	●	●	●
	BIND	/etc/resolv.conf	●	●	●	●
	BIND	/var/run/secret/kubernetes.io/serviceaccount	○	○	●	●

4.1.2 Pod Filesystem Regions. A pod's filesystem consists of multiple mount points, which we refer to as *filesystem regions*, including the *root filesystem*, *virtual filesystems*, and *bind mounts*.

Root Filesystem (RootFS). A pod's root filesystem is backed by OverlayFS [27], which overlays a writable upperdir on top of one or more read-only lower layers (lowerdir). This design permits both file modification and execution within the pod and therefore, represents a primary source of writable and executable exposure.

Virtual Filesystems. A pods has several virtual filesystems mounted by the kernel and the container runtime. The kernel mounts */proc*, */sys*, and */sys/fs/cgroup* to expose system metadata. These filesystems are read-only by default, and execution is blocked. Even when remounted with write permissions, access remains limited to specific control interfaces, such as *uevent_helper* or *cgroup* parameters, and execution is not permitted. In contrast, the container runtime (e.g., containerd) mounts */dev* as a tmpfs-backed filesystem with both write and execute permissions enabled by default. Other virtual filesystems, including */dev/pts* and */dev/mqueue*, are mounted without execute permissions. These restrictions persist even if remounted: */dev/pts* remains non-writable and non-executable, while */dev/mqueue* permits only structured writes via the POSIX message queue interface.

Bind Mounts. The kubelet injects several host files into pods via bind mounts, including *hosts* and *resolv.conf* for networking, *termination-log* for exit status reporting, and the serviceaccount directory for authentication. These mounts allow write and execute access by default, except for serviceaccount, which is read-only unless explicitly remounted with additional permissions. The container runtime additionally mounts */dev/shm* as a tmpfs-backed bind mount that is writable but non-executable unless remounted.

4.1.3 Monitoring Scope. Container escape attacks typically require filesystem regions that permit both write and execute access. As summarized in Table 1, such exposure arises from three sources: the root filesystem, the tmpfs-backed */dev*, and bind mounts injected by the kubelet or container runtime. Although virtual filesystems such as */proc* and */sys* expose writable entries, these interfaces accept only format-constrained values (e.g., flags, tunables) and are non-executable, making them insufficient for completing escape attacks. Accordingly, to address (❶), we focus our monitoring on filesystem regions that are writable and executable, whether by default or through adversarial reconfiguration, namely the OverlayFS `upperdir`, */dev*, and all bind mounts.

4.2 Ephemeral Writes Analysis

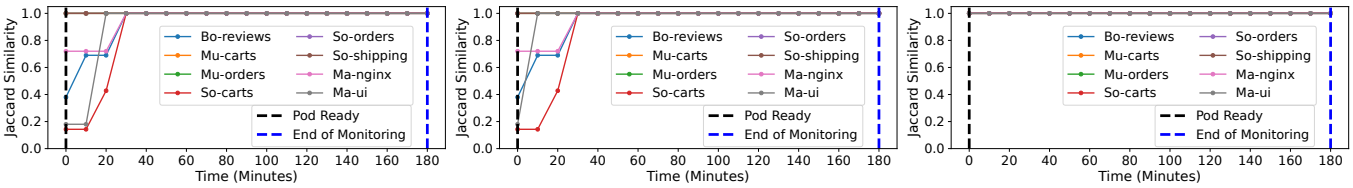
We examine how microservices-based applications interact with the filesystem during normal execution, focusing on the identified regions that permit writes and execution. To isolate legitimate behavior, we exclude adversarial scenarios and, accordingly, filesystem locations that become writable and executable only under adversarial settings.

Microservices-based applications. We conducted our analysis on five representative microservices-based applications (Table 2). These were selected because (i) several are widely used in state-of-the-art container security research [1; 6; 13; 14; 28], (ii) they span diverse application domains, and (iii) together they comprise 33 microservices with heterogeneous runtime environments. This heterogeneity captures runtime-induced filesystem differences (e.g., JIT artifacts, language-specific caches), ensuring write patterns are not ecosystem-specific and reflect production-grade deployments.

Table 2: Microservices-based Applications.

Application	# of svc	Polyglot	Maintainer
Bookinfo [29]	4	✓	Istio
Sock-shop [30]	7	✓	Weavework
Mu-shop [31]	9	✓	Oracle
Martian Bank [32]	8	✓	Cisco
Cinema [33]	5	×	Morejon (individual)

Exercising normal behavior. We exercised each application using realistic client-side workloads that reflect its intended functionality. Specifically, we generated user activity under varying request rates, input sizes, and interaction sequences using *Locust* [34] for load generation, *Selenium* [35] for browser-driven interactions, and *curl* [36] for API-level requests. These workloads covered the core



(a) Full Path. (b) Parent Directory. (c) Longest Shared Prefix.

Figure 2: Temporal convergence of writable paths at varying granularity levels.

application endpoints and usage patterns expected during normal operation. Each application was executed continuously for three hours, during which we recorded all filesystem write activity.

4.2.1 Write Type and Filesystem Scope. As shown in Table 3, not all microservices perform filesystem writes during normal operation. For example, in the Bookinfo application, only the reviews microservice writes to the filesystem. Also, observed writes are limited to two types: (i) file or directory creation (e.g., *nginx* instantiates per-worker subdirectories for buffering), and (ii) content updates to non-executable files (e.g., JVM-generated performance data in `/tmp/hsperfdata_*`). No microservice writes to executables or alters file permissions, and all writes remain confined to a single filesystem region, the pod's root filesystem.

Table 3: Legitimate Writes per Application. Create (file/directory creation), Overwrite (non-executable file modification), Enable-exec (permission change or executable modification).

Application	Write-Svc/Total	Filesystem Regions				
		Overlay RootFS (/)			Tmpfs (/dev)	Bind[1]
		Create	Overwrite	Enable-exec		
Bookinfo	1/4	✓	✓	×	×	×
Sock-shop	3/7	✓	✓	×	×	×
Mu-shop	2/9	✓	✓	×	×	×
Martian Bank	2/8	✓	✓	×	×	×
Cinema	0/5	×	×	×	×	×

[1] In the case of bind mounts, we exclude the service account volume from this experiment, as it is mounted as read-only by default during normal execution and becomes writable only in adversarial settings, which we exclude in this experiment.

4.2.2 Consistency of Writes. The stateless execution model of microservices implies that filesystem writes are confined to locations established during container initialization, as persistent application state is externalized and the filesystem is used only for auxiliary execution support. However, runtime execution may continue to populate these locations with additional files and directories, raising the question of how this write footprint should be represented so that it remains stable under benign runtime activity. To investigate, we track filesystem writes at 10-minute intervals starting from the PodReady phase [37], which marks the end of initialization. Using the final interval as reference, we compute the Jaccard similarity [38] between its cumulative set of written paths and those observed earlier, and compare this evolution at three representations: full paths, parent directories, and longest shared prefixes.

As shown in Figure 2, full paths converge slowly, with new files and directories continuing to appear after the PodReady phase. For example, in *nginx*, */var/cache/nginx/proxy_temp* and */var/cache/nginx/client_temp* are created at initialization, but runtime requests generate deeper paths such as */var/cache/nginx/proxy_temp/1/00*. The same instability is visible at the parent-directory level, where */var/cache/nginx/proxy_temp/1/* emerges only during runtime. In contrast, longest shared prefixes converge at PodReady. Both */var/cache/nginx/proxy_temp* and */var/cache/nginx/client_temp* are already present at PodReady, and their shared prefix */var/cache/nginx* bounds all subsequent write activity.

To address (❷), we determine how write activity within monitored regions can be interpreted to distinguish normal execution from escape behavior. Our analysis shows that representations based on full paths or parent directories are overly sensitive to benign runtime variability and lead to spurious alerts. In contrast, the longest shared-prefix representation captures stable filesystem locations that bound legitimate write activity. Thus, we define *safe locations* as the longest shared prefixes present at PodReady and introduce *selective immutability* as our detection principle.

> ***Selective Immutability*** is governed by two invariants. The *location invariant* is automatically derived from container startup behavior, treating any write outside the safe locations inferred at PodReady as a deviation from normal write activity. The *execution-capability invariant* reflects an inherent property of stateless microservices, where legitimate writes neither alter file permissions nor target executable content, and any such modification constitutes a departure from normal behavior regardless of location.

5 System and Threat Model

We focus on stateless microservices that execute narrow, single-responsibility tasks. Such services constitute the majority of cloud-native deployments [39]. We exclude stateful workloads (e.g., cache services, databases, message queues), which are commonly deployed alongside microservices but require external persistent volumes beyond the pod's filesystem.

We consider external adversaries targeting public-facing pods to escape the pod environment. We assume that the adversary attains privileged execution within the pod, either due to over-permissive configurations or through in-pod privilege escalation via kernel

vulnerabilities (e.g., CVE-2022-0492 [24]). Although Kubernetes best practices discourage excessive privileges, such configurations remain common in production deployments [19]. Moreover, kernel-level privilege escalation enables adversaries to obtain equivalent capabilities even in otherwise correctly configured pods [24]. We assume the adversary does not have access to the Kubernetes API or the *containerd* socket and therefore exclude escape attacks that rely on adversary-created pods. We further assume that microservice images are trusted and sourced from verified repositories, placing supply-chain-based escapes out of scope. Our focus is on escape techniques that leave filesystem artifacts – fully fileless, in-memory escapes are discussed separately in Section 8.

Finally, our system is a node-based IDS, monitoring microservices in Kubernetes[3]. Unlike existing solutions, *ImmuCheck operates without modifying the kernel or relying on pre-deployed syscall-tracing tools on the underlying node.* Instead, it monitors pods externally using the Linux `inotify` [40] subsystem to track pod filesystem activity, which we assume to be available on the host. While an attacker may attempt to tamper with our solution, doing so requires escaping the pod environment to the underlying node. ImmuCheck detects such escape activity and, in many cases, provides sufficient lead time for prevention (Section 7).

6 ImmuCheck: Approach & Design

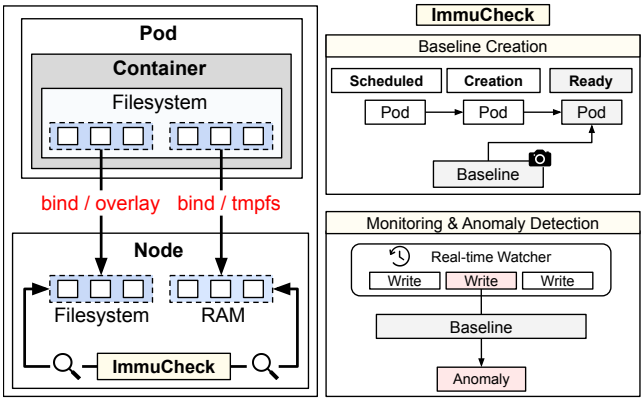


Figure 3: ImmuCheck Overview. (A) Once the pod reaches the PodReady state, ImmuCheck automatically establishes a baseline of safe locations. (B) It then resolves the pod's filesystem to its corresponding node-level paths and initiates monitoring, enforcing selective immutability to detect deviations from the baseline.

We present ImmuCheck, a detection system for pod escape attacks. Our approach builds on the observation that such attacks typically involve filesystem writes. Instead of relying on long-lived filesystem integrity baselines (e.g., Tripwire [41]), ImmuCheck focuses on violations of filesystem invariants established during pod initialization. Specifically, it enforces selective immutability by permitting writes only to safe locations instantiated during startup. Writes outside these locations indicate deviations from the expected execution context and are flagged as potential escape activity.

[3]ImmuCheck also applies to Docker environments, see Appendix A

Figure 3 shows the system design of ImmuCheck, which comprises three phases: baseline creation, monitoring, and anomaly detection. In the baseline creation phase (Ⓐ), ImmuCheck automatically identifies locations instantiated during startup that serve as safe targets for legitimate ephemeral writes. This phase is fast, lightweight, and requires no prior behavioral modeling. In monitoring and anomaly detection (Ⓑ), ImmuCheck maps the pod's filesystem regions (i.e., mount points) to their corresponding host paths and continuously monitors runtime write activity. A selective immutability policy is then enforced to flag escape behavior in real time, without disrupting normal microservice execution.

6.1 Baseline Creation

According to our preliminary assessment (Section 4), microservices consistently write to the root filesystem, and these writes are confined to safe locations established by the time the pod reaches the PodReady state. To construct the baseline, we automatically extract startup writes, defined as writes occurring during initialization up to PodReady, and derive the safe regions. The construction process consists of two steps.

Upperdir identification. To extract startup writes, we leverage OverlayFS [27], the default union filesystem in Kubernetes that combines read-only image layers with a writable layer (i.e., `upperdir`) recording runtime writes (Figure 4). Our approach takes the pod's name or ID as input, resolves the container ID, and locates the container's `upperdir` by querying the node's mounted filesystems.

Safe locations identification. With the `upperdir` identified, we traverse its directory tree at the PodReady timepoint to collect all written paths. We then group structurally related paths and compute the longest shared prefix within each group to derive the safe locations, which together form the baseline.

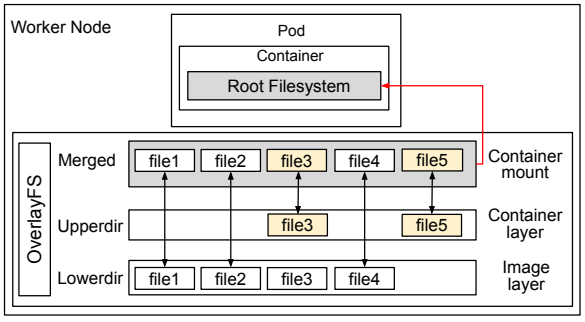


Figure 4: OverlayFS Structure. This structure unifies the lower and upper directories into a merged filesystem in containers.

Overall, our baseline creation process is fully automated, instantaneous, and lightweight, as we demonstrate in our evaluation (Section 7), and entirely external to the pod, requiring no kernel modification. Moreover, ImmuCheck derives a fresh baseline for each pod deployment, ensuring alignment with the running microservice image. This design inherently accommodates evolving microservices, avoiding the retraining costs associated with existing anomaly-based detectors.

6.2 Monitoring & Anomaly Detection

After baseline creation, ImmuCheck automatically transitions to monitoring and anomaly detection in three steps: *filesystem region mapping*, *runtime monitoring*, and *anomaly detection*.

6.2.1 Filesystem Regions Mapping. ImmuCheck resolves the pod's root filesystem, tmpfs-backed `/dev`, and all bind mounts to their corresponding node-level paths. While provisioned by the container runtime and the kernel, observing these regions from the host is non-trivial due to storage abstraction and namespace isolation, with each requiring a distinct resolution strategy.

```
"info": {
  "pid": 4024,
  "runtimeSpec":
      "annotations": {}
      "mounts": [
        {
          "destination": "/etc/hosts",
          "source": "/var/lib/kubelet/pods/<pod-uid>/etc-hosts",
          "type": "bind",
        }
        ...
        {
          "destination": "/dev/shm",
          "source":"/run/containerd/io.containerd.grpc.v1.cri/
          sandboxes/<sandbox-id>/shm",
          "type": "bind",
        }
  "runtimeType": "io.containerd.runc.v2"
}
```

Figure 5: Node-level paths for bind mounts. This data is extracted from the container runtime interface (CRI) metadata.

Root Filesystem. We resolve the root filesystem via the `upperdir` path identified during baseline creation, which records all modifications to the root filesystem.

Bind Mounts. We use container runtime metadata exposed through the container runtime interface (CRI) [42] to resolve the node-level paths of bind mounts. From each mount specification, we extract the `source` field, which records the exact host path mapped into the container's filesystem. For example, as shown in Figure 5, the container bind mount `/etc/hosts` maps to the node-level path `/var/lib/kubelet/pods/<pod-uid>/etc-hosts`.

Tmpfs-backed `/dev`. The kernel mounts `/dev` as a *tmpfs* to provide isolated, writable device files. While this mount resides in the container's private namespace, it is accessible from the host through the *procfs* interface. By retrieving the container's PID from CRI metadata (e.g., 4024 in Figure 5) and resolving `/proc/<pid>/root/dev`, we obtain a node-level view of the container's `/dev` directory.

Overall, baseline creation and filesystem mapping are one-shot operations triggered at `PodReady`, completing within milliseconds (Table 10). While recurring per pod, both remain negligible in high-churn environments, relying on node-level filesystem metadata without pod instrumentation or kernel modification. Furthermore, as the resolved regions are provisioned by the container runtime and the kernel, the monitoring scope remains fixed and independent of microservice complexity, ensuring predictable overhead.

6.2.2 Runtime Monitoring. With container filesystem regions successfully resolved to host-level paths, ImmuCheck leverages *inotify* [40], a Linux kernel subsystem that provides efficient, event-driven filesystem notifications, placing watches on these paths to record write events in real time and enable timely detection of container escape attempts.

6.2.3 Anomaly Detection. ImmuCheck's anomaly detection operationalizes *selective immutability* by configuring *inotify* according to the automatically derived baseline. The *location invariant* is enforced across all monitored regions, flagging any write outside the safe locations established at `PodReady` as a deviation from the expected write footprint. The *execution-capability invariant* is enforced within safe locations, flagging permission changes or modifications to executable content, as such changes are inconsistent with the stateless execution model of microservices. This two-tier design distinguishes routine microservice writes (e.g., logging and caching) from behavior indicative of container escape attempts.

7 Evaluation

We implement ImmuCheck in *Python* [43] and evaluate it across diverse pod escape scenarios. Our evaluation examines detection accuracy and lead time, benchmarks against state-of-the-art detectors, measures runtime overhead at scale, and tests robustness against evasive attacks.

7.1 Experimental Setup & Dataset

Experimental setup. We evaluate ImmuCheck on five representative microservices-based applications (Table 4), distinct from those in our preliminary assessment. These applications, distributed by major cloud providers and adopted in prior research [1; 6; 13; 14; 28], span 34 microservices with diverse runtimes and configurations, enabling us to assess ImmuCheck across architectures and workloads representative of real-world deployments. To the best of our knowledge, this constitutes the largest evaluation composed exclusively of cloud-provider microservices in container security research, substantially extending prior work. In our evaluation, we deploy applications on a single-node *Kubeadm* [44] cluster running Ubuntu 20.04.6 LTS with containerd as the runtime. We select *Kubeadm* because it is the official upstream tool for bootstrapping Kubernetes clusters and is widely used in industry, making it representative of production environments [45].

Table 4: Evaluated Microservices-based Apps.

Name	# service	Polyglot	Maintainer
Online Boutique [46]	10	✓	Google Cloud
Bank-of-Anthos [47]	6	✓	Google Cloud
Robot-shop[31]	8	✓	IBM Instana
AKS Store [48]	5	✓	Microsoft Azure
Retail Store [49]	5	✓	Amazon AWS

Normal behavior simulation. As in our preliminary assessment, we generated user activity under varying traffic volumes and input sizes using *Locust* [34], *Selenium* [35], and *curl* [36], and ran each application continuously for three hours. For applications with native load generators, we used these tools to drive interaction

Table 5: Container Escape Attack Scenarios (Grouped by Escape Vector).

Vector	Vulnerability	Short Description	Application
Usermode Helper	Release_Agent	Release Agent Abuse	Online Boutique
	Core_Pattern	Core Dump Abuse	Bank-of-anthos
	Modprobe	Modprobe Path Abuse	AKS-Store
	Uevent_Helper	Device Handler Abuse	Robot-shop
	Hotplug	Hotplug Handler Abuse	Retail-Store
Kernel & Container Runtime	CVE-2022-0847	Dirty Pipe	Online Boutique
	CVE-2016-5195	Dirty Cow	Bank-of-anthos
	CVE-2019-5736	Runc Overwrite	AKS-Store
Deployment	SYS_MOD	Kernel Module Injection	Robot-shop
	Host_NET	Host Network Abuse	Retail-Store
	PTRACE	Process Debugging	Online Boutique
	Host_FS	Host Filesystem Mount	Bank-of-anthos

Table 6: Performance results of ImmuCheck.

Escape Scenario	Precision	Recall	F1-score
Release_Agent	0.9916	1.0000	0.9958
Core_Pattern	0.9922	1.0000	0.9961
Modprobe	0.9927	1.0000	0.9963
Uevent_Helper	0.9926	1.0000	0.9963
Hotplug	0.9922	1.0000	0.9961
CVE-2022-0847	0.9916	1.0000	0.9958
CVE-2016-5195	0.9922	1.0000	0.9961
CVE-2019-5736	0.9927	1.0000	0.9963
SYS_MOD	0.9926	1.0000	0.9963
Host_NET	0.9922	1.0000	0.9961
PTRACE	0.9916	1.0000	0.9958
Host_FS	0.9922	1.0000	0.9961

flows. For the remaining applications, we reproduced equivalent actions, including page navigation, form submissions, and API calls, under varying traffic levels.

Escape attack simulation. To simulate attacks, we use public pod escape PoCs [21; 25; 50], including scenarios from the container breakout dataset (CB-DS) [15], several of which have been weaponized in real-world container breaches [51]. While some correspond to known CVEs, the majority arise from user-space helpers and pod misconfigurations, reflecting the more prevalent class of escape techniques in practice. Table 5 summarizes the evaluated scenarios, covering the majority of known pod escape techniques.

Evaluation dataset. While ImmuCheck operates in watcher mode, we segment the evaluation into 30-second intervals, referred to as *runs*. Under normal operation, the three-hour monitoring period yields 360 runs per microservice. For the attack evaluation (conducted outside the normal monitoring window but under concurrent user traffic), we execute one escape scenario per application, assigning each technique to the application whose runtime conditions (e.g., base image, available tooling) satisfy the escape's prerequisites. Each run lasts 30 seconds and covers the complete escape execution, including setup and trigger phases, with 20 runs per microservice.

7.2 Detection Capabilities

Table 6 shows the performance of ImmuCheck. Our approach achieves a 100% recall in all escape scenarios. This reinforces our core premise: pod escape attacks commonly trigger filesystem writes, which ImmuCheck reliably detects. While ImmuCheck monitors multiple filesystem regions for comprehensive coverage, malicious activity is confined to the pod's root filesystem (i.e., *upperdir*), consistent with the behavior of publicly available escape PoCs.

Nevertheless, each microservice produces two to three false positives over the three-hour monitoring window, all stemming from Kubernetes' service account token rotation. By default, the kubelet binds a service account volume into every pod, regardless of whether it requires access to the Kubernetes API. Although this bind-mount is *read-only* from the pod's perspective, the kubelet retains write access to its host-level path. It refreshes the credentials approximately every hour by creating a new timestamped subdirectory with the token, certificate authority, and namespace files, and removing the previous one. Since ImmuCheck flags any write in bind mounts, these routine updates are detected as false positives. Yet, this can be mitigated through two practical approaches. First, since microservices commonly perform business logic without interaction with the Kubernetes API, configuring the pod's *automountServiceAccountToken* field to *false*, would prevent the injection of the service account mount, thereby eliminating such periodic writes. Second, such false alerts can be correlated with the kubelet logs, which record these updates, allowing them to be safely filtered without compromising detection accuracy. Overall, despite such minor infrastructure-driven false positives, ImmuCheck exhibits strong detection capabilities against escape attacks. Appendix B presents representative examples of how ImmuCheck detects different escape techniques.

7.3 Comparison with Existing Solutions

Using the same dataset from our evaluation, we benchmark the performance of ImmuCheck against existing solutions.

Falco. With its default ruleset, Falco detects only 20% of usermode-helper escapes, but achieves full coverage of kernel- and container-runtime escapes and 70% coverage of deployment-based escapes, as shown in Figure 6. The detection gap arises from four escapes with no corresponding rules and from one escape that successfully evades an existing rule. Although Falco produces no false positives, its coverage remains incomplete and depends on continuous maintenance of the ruleset to keep pace with evolving attack vectors.

Tracee. With its default ruleset, Tracee detects 20% of usermode-helper escapes, missing three due to absent rules and one that bypasses an existing rule, while achieving full coverage of deployment- and kernel/runtime-based escapes, as shown in Figure 6. Unlike Falco, Tracee produces false positives, primarily due to its *dynamic code loading* rule, which flags transitions of memory pages from writable to executable. Although intended to detect malicious behavior, this rule triggers in Node-based microservices, where such transitions are a normal artifact of *JIT* compilation in the *JavaScript* engine. Rule tuning can mitigate such false alerts, but requires microservice-specific knowledge. Consequently, Tracee's effectiveness depends not only on ongoing rule maintenance but also on

workload-specific tuning, which is impractical in dynamic microservices environments.

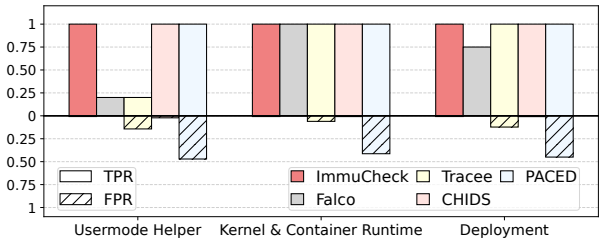


Figure 6: Performance of ImmuCheck compared to existing solutions. Results are averaged per escape category (i.e., vector).

CHIDS. The authors employ a graph-based structure to generate vectors that capture the contextual influence and frequency of unseen syscalls and arguments. An autoencoder is trained to minimize reconstruction error on normal syscall data, and syscall traces are flagged as anomalous when the error exceeds a predefined threshold. Following the original paper, we adopt the same sequence length and threshold in our evaluation. The approach achieves 100% detection across all scenarios, as pod escape steps introduce previously unseen syscalls (e.g., `mount`, `finit_module`) with strong contextual influence. It produces a small number of false positives, mainly from benign but unseen arguments in normal execution (e.g., `tmp` artifacts). Despite strong performance, CHIDS suffers from baseline aging [13], requiring frequent retraining to sustain accuracy, which is misaligned with the pace and scale of dynamic microservices environments.

PACED. This approach relies on CamFlow [23] for whole-system provenance via LSM hooks, but CamFlow requires kernel patching, making it unsuitable for standard Kubernetes environments. For practical evaluation, we reimplemented PACED's detection logic using syscall tracing to capture privileged flows, namely pod writes to *inodes* that are later read by host processes. This preserves PACED's core semantics while remaining compatible with Kubernetes. PACED achieves a 100% detection rate across all escape scenarios, but produces false positives due to its cross-namespace rule. The pattern of a pod writing to an *inode* that is then read by the host also arises in runtime behavior. Specifically, container *stdout* and *stderr* are exposed to the host via pipes or pseudo-terminals shared across namespaces. Host processes such as *containerd-shim* (responsible for managing container I/O) routinely read from these streams to capture logs, resulting in false alerts.

Overall, ImmuCheck achieves detection performance comparable to CHIDS, with a 100% detection rate and minimal false positives. While CHIDS requires frequent retraining to maintain optimal performance (as shown in [13]), ImmuCheck avoids this overhead by exploiting the stateless nature of microservices and the early stability of legitimate write locations. This enables escape detection without retraining, rule update, or provenance collection, eliminating the coverage–overhead trade-off that limits existing solutions.

7.4 Detection Lead Time

The effectiveness of ImmuCheck also lies in its attack detection lead time, defined as the duration from the initial detection of an attack's indicator(s) to the attack's completion [15]. In this section, we evaluate the detection lead time achieved by ImmuCheck compared to Tracee, Falco, and CHIDS. We exclude PACED from this comparison, as its original implementation relies on whole-system provenance collection, which incurs high overhead and is impractical for real-time detection.

Detection lead time analysis. As shown in Figure 7, ImmuCheck detects the majority of escape attacks with a lead time. The highest average lead times are observed in `Sys_Mod` (see Table 7), `CVE-2022-0847`, and `Host_NET`, achieving 1.245s, 0.579s, and 0.576s, respectively. In these escapes, the adversary relies on custom payloads, which must first be downloaded. This download step involves network communication and data transfer, introducing latency that ImmuCheck exploits to detect the escape with a comfortable lead time. Even in usermode-helper-based escapes, where no download occurs, ImmuCheck achieves some lead time by capturing preparatory actions such as creating new directories for mounting pseudo-filesystems (e.g., *proc*) and preparing executables. In contrast, the `Host_FS` escape yields low lead time (0.012s), as it involves a rapid two-step process: directory creation followed by the host filesystem mount. Among existing solutions, Tracee performs similarly to ImmuCheck and often achieves lower lead times by detecting dropped executables before execution. Falco offers minimal lead time, alerting only when non-base binaries are executed, typically the final escape step. CHIDS provides no lead time, as it must first collect and process syscall traces before detecting anomalies.

Detection lead time with automated response. Response measures usually range from blocking egress traffic to labeling compromised pods or terminating affected pods or nodes. To assess the impact of detection lead time, we focus on pod termination, a commonly used response in Kubernetes environments [52; 53]. By default, Kubernetes grants up to 30 seconds for graceful termination [54]. However, in our experiment, we enforce a zero-second grace period upon detection. This allows the *kube-apiserver*[55] to kill the pod instantly [54]. While such a forcible termination may lead to dropped in-flight requests or incomplete cleanup routines, this trade-off is acceptable for stateless pods, which are often replicated, maintaining availability. Using the dataset structure in 4, we evaluate ImmuCheck's lead time in preventing pod escapes.

As shown in Table 8, ImmuCheck provides sufficient detection lead time to fully prevent the escape in 6 out of 12 scenarios, yielding a 100% prevention rate. For usermode-helper based escapes, the available lead time is occasionally too short to guarantee full prevention. While the lead time itself is often below 100 ms, the more critical factor is the non-deterministic latency of the *kube-apiserver* response, which provides no predictable window for enforcement. Yet, our approach consistently achieves over 75% prevention across all such escapes. In the `Host_FS` scenario, prevention occurs post-escape, as the mount completes before any alert is raised. However, since the mount remains within the pod's namespace, terminating the pod still revokes attacker access. Overall, ImmuCheck offers actionable lead time across several automated escape attacks. While it may not fully prevent escapes in certain cases, it still affords

Timestamp	Attack	Detectors				Alarm Description (🔔)
		ImmuCheck	Falco	Tracee	CHIDS	
50:10:000		⊙	⊙	⊙	⊙	
50:10:278	`curl -o load_mod -L https://a-server/load_mod`	✓	×	✓	✓	
50:10.581		🔔				New file created in the *upperdir*
50:10.582				🔔		New executable dropped
50:10.922	`curl -o mod.ko -L https://a-server/mod.ko`		×		✓	
50:11.916	`chmod +x load_mod`		×		✓	
50:11.919	`./load_mod mod.ko`		✓		✓	
50:11.924			🔔			Executing binary not part of base image
50:11.927	Kernel module loaded					
50:13.221					🔔	Malicious syscall trace

Table 7: Automated pod escape via kernel module injection (i.e., SYS_MOD). ⊙ denotes the start of monitoring. ✓ indicates detection of the command. × indicates failure to detect. 🔔 signals an alarm. The red timestamp denotes the escape's completion. The lead time is 1.245s for ImmuCheck, 1.244s for Tracee, 0.003s for Falco and no lead time for CHIDS.

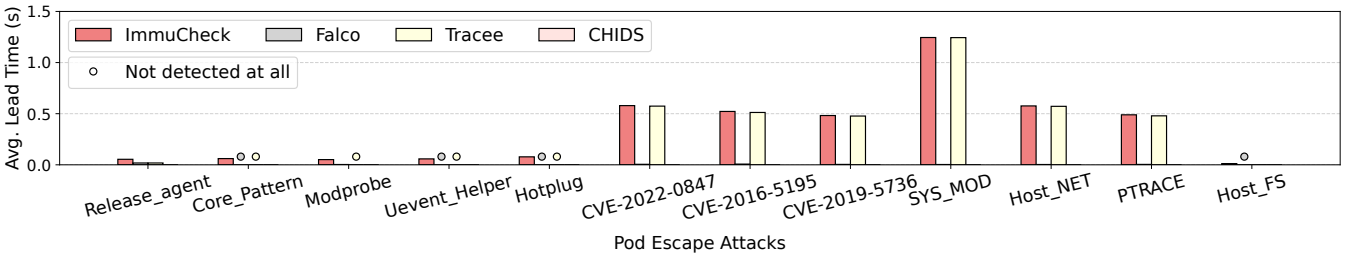


Figure 7: Average detection lead time of ImmuCheck on different automated escapes compared to Falco, Tracee and CHIDS. Longer lead times enable response engines to respond more quickly and effectively.

Table 8: Lead Time and Escape Prevention.

Escape Attack	Prevention Rate
`Release_Agent`	0.8550
`Core_Pattern`	0.9083
`Modprobe`	0.8687
`Uevent_Helper`	0.7700
`Hotplug`	0.9800
`CVE-2022-0847`	1.0000
`CVE-2016-5195`	1.0000
`CVE-2019-5736`	1.0000
`SYS_MOD`	1.0000
`Host_NET`	1.0000
`PTRACE`	1.0000
`Host_FS`	0.0000

incident response teams an ample window to deploy post-escape measures (e.g., node draining to stop lateral movement).

7.5 Runtime Performance and Scalability

The runtime overhead of ImmuCheck depends on the hardware characteristics of the node and the number of concurrently running pods. We evaluate CPU and memory usage across deployment scenarios on two *Kubeadm*-provisioned nodes (Table 9), scaling from 1 to 40 pods to reflect typical densities in production clusters [56]. For each scenario, we run a one-hour monitoring session per application, recording CPU and memory consumption at two-second intervals, and report average usage over all runs and applications.

As shown in Figure 8, ImmuCheck maintains consistently low CPU overhead across pod densities and hardware configurations. On *node01*, CPU usage averages below 0.4% and peaks at 1.35% with six pods. On *node02*, CPU usage remains low, averaging below 0.4% and peaking at 1.92% with 40 pods. This behavior is attributable to the event-driven design of ImmuCheck, where processing occurs only when filesystem activity is present. Under benign conditions, stateless microservices generate sparse writes after initialization, resulting in minimal overhead. Memory usage increases with pod count, reaching 9.6% at six pods, due to the allocation of eight persistent *inotify* watchers per pod, each monitoring a distinct filesystem region. On *node02*, memory usage scales sub-linearly to 9.3% at the highest density. These results show that ImmuCheck imposes negligible CPU cost and bounded memory overhead, making it practical for real-world clusters.

Table 9: Setups of Kubeadm Nodes.

Node	CPU	Memory
`node01`	Intel Xeon E312xx, Single-Core, 3.4GHz	4GB
`node02`	Intel Core i7-10850H, Quad-Core, 2.7GHz	16GB

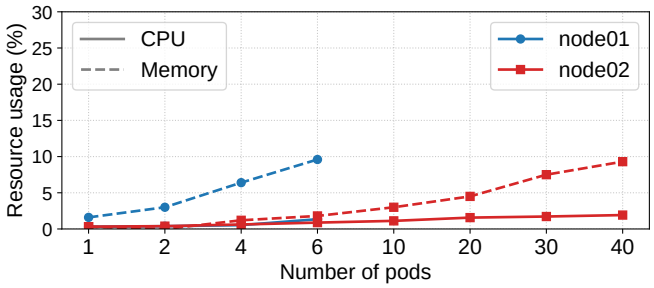


Figure 8: ImmuCheck resource usage across different pod counts. `node01` hosts at most 6 pods due to CPU limits.

We also evaluate the one-time cost of baseline creation and filesystem-region mapping. These steps are performed per microservice, and we report per-application averages in Table 10. Baseline creation completes between 0.015s and 0.088s, while filesystem mapping ranges from 0.056s to 0.110s across all applications. These millisecond-level times demonstrate that the process is instantaneous. The efficiency stems from operating entirely at the node level and relying solely on filesystem metadata, avoiding in-pod instrumentation, syscall interception, and content inspection.

Table 10: Avg Execution Time by Application.

Application	Baseline Creation	Filesystem Mapping
Online Boutique	0.0152s	0.0568s
Bank-of-Anthos	0.0214s	0.0822s
Robot-shop	0.0712s	0.0816s
AKS Store	0.0879s	0.1108s
Retail Store	0.0422s	0.1017s

7.6 Robustness Against Evasion

In this section, we evaluate the resilience of ImmuCheck against evasive escape attempts. We focus on two techniques that directly target its threat model: (1) *Baseline Abuse*, where adversaries exploit safe locations established by the baseline for malicious writes, and (2) *In-memory Execution*, where adversaries invoke low-level interfaces to execute escapes entirely in memory. We do not evaluate evasion techniques based on memory corruption, as such vectors are less practical in microservices environments where applications are predominantly implemented in memory-safe languages (e.g., Go, Python, Java). We evaluate both techniques across all escape scenarios using the same dataset structure as in our evaluation (one scenario per application), and we report the average detection rate (recall) per category.

Baseline abuse. This technique targets safe locations tolerated by ImmuCheck for legitimate writes. We assume the attacker knows these locations, a realistic assumption given the uniformity of containerized microservices and the widespread reuse of base images. For example, an `nginx` pod typically writes error logs to `/var/log/nginx/`.

In-memory execution. This technique avoids file-backed execution by repurposing trusted runtimes already embedded in the microservice image. Instead of dropping new binaries, the attacker leverages pre-installed interpreters to execute payloads directly in memory, evading write-based detection. We distinguish two sub-techniques under this category:

- **Direct syscall invocation.** Some runtimes expose low-level interfaces that allow attackers to invoke syscalls directly. We use this capability in our escape attacks to execute escape-enabling syscalls (e.g., `mount`).
- **In-memory payload injection via `memfd_create`.** For escapes that require introducing custom files or binaries (e.g., kernel modules), we leverage `memfd_create` to create anonymous, memory-backed file descriptors, load content into them, and then execute it through the appropriate syscalls.

Table 11: Detection rates (recall) of ImmuCheck against evasion (averaged by categories).

Escape Scenario Category	Baseline Abuse	In-memory Execution
Usermode Helper	1.0000	1.0000
Kernel & Container Runtime	1.0000	0.7666
Deployment	0.8750	0.8235

7.6.1 *Performance Analysis.*

Baseline abuse. As shown in Table 11, ImmuCheck achieves high detection rates across all categories of escape attempts. While attackers may place payloads in safe locations, making them executable triggers detection. The only partial evasion occurs in the `Host_FS` scenario, evaluated on Bank-of-Anthos. Here, the host filesystem is mounted into a legitimately created directory, bypassing detection in some microservices and yielding an average detection rate of 87.5% for the deployment category. This stems from differences in the base images of Bank-of-Anthos. Its *alpine*-based microservices, where `mount` is provided by *BusyBox* [57], generate no `/run/mount` artifacts. In contrast, microservices using util-linux `mount` (libmount) produce state under `/run/mount`, violating the baseline and triggering detection. Yet, to handle such cases, the gap can be closed by externally monitoring `/proc/<container-PID>/mountinfo` for newly created mount points. Overall, abusing safe locations does not preclude detection. Completing an escape still requires payload preparation, kernel-interface interaction, or mount operations, each of which either violates an execution-capability invariant or produces writes outside the safe locations, substantially constraining attacker maneuverability.

In-memory execution. As shown in Table 11, in-memory execution succeeds only in microservices embedding *Python*, spanning the *deployment* and *kernel & container runtime* categories. In such workloads, the attacker invokes syscalls directly through `ctypes`, calling `libc.syscall` with numeric identifiers to execute escape-enabling syscalls such as `mount`. When the escape requires injecting additional files, the attacker relies on `memfd_create` to instantiate anonymous memory-backed file descriptors, load content into them, and invoke syscalls such as `fchmodat` and `execve` for execution or `finit_module` to load kernel modules, thereby enabling fully fileless execution. Other microservices either lack embedded runtimes

for direct syscall invocation, requiring compiled loaders or external tooling, or they generate detectable artifacts during execution. For example, in *Go*-based microservices, the attacker stages a script in a safe location to invoke `memfd_create`, and its execution triggers on-the-fly compilation, leaving artifacts under `.cache/go-build/`. In *Node* environments, syscall invocation requires installing native extensions such as `ffi-napi`, which produces detectable writes in `.npm/_cacache`. Within the *usermode helper* category, ImmuCheck achieves a full detection rate. Here, escapes rely on the host to execute a crafted payload. For the host to do so, the payload must be placed in a host-visible location within the pod's root filesystem, typically the `upperdir`.

Overall, our results demonstrate that fileless escape attempts which avoid filesystem artifacts are confined to a narrow and well-defined set of execution environments. Across the remaining scenarios, escape activity consistently produces observable filesystem artifacts that ImmuCheck detects with high accuracy, significantly constraining an attacker's ability to evade detection.

8 Discussion

Despite effectively detecting container escape attacks. ImmuCheck is not exempt from limitations.

8.1 Deployment Challenges

Different storage drivers. ImmuCheck relies on *OverlayFS* to monitor the root filesystem via the pod's `upperdir`. Alternative drivers such as *BTRFS* [58] and *ZFS* [59] are incompatible with this design. In practice, this is not restrictive, as *OverlayFS* is the default and widely adopted across modern container runtimes and Kubernetes platforms [27].

Applicability beyond microservices. ImmuCheck assumes that filesystem write locations stabilize after `PodReady`, which holds broadly for stateless microservices. This assumption is validated across the evaluated applications from major cloud providers. Stateful or general-purpose workloads (e.g., plugin-based platforms) may exhibit more diverse filesystem behavior. While such workloads remain susceptible to container escape attacks, applying ImmuCheck in these environments may require redefining the notion of safe regions, which we leave for future work.

8.2 Detection Challenges

Inconsistent artifact generation. Filesystem artifact generation varies with the pod's base image and execution environment. In our evaluation, this manifests in a single case: mounting activity in Alpine-based microservices produces no artifacts, unlike other base images. This can be mitigated by monitoring `/proc/<container-PID>/mountinfo` from the host, where any modification reflects the creation of a new mount point.

Fileless escapes. While most escape techniques generate detectable filesystem artifacts, a narrow class of fileless escapes can occur if the pod embeds the necessary tooling or is misconfigured accordingly. For example, pods with embedded syscall interfaces can enable fully in-memory escapes. Such attacks require a precise combination of misconfigurations and runtimes, limiting practical feasibility.

Reliance on Inotify. ImmuCheck relies on *inotify* for low-latency filesystem notifications. Since *inotify* maintains a bounded per-instance event queue, a sustained burst of watched events can overflow the queue and drop subsequent events. An attacker could exploit this by flooding the filesystem to conceal malicious writes. Yet, when overflow occurs, the kernel emits an `IN_Q_OVERFLOW` event, providing a detectable signal. Queue saturation also induces elevated I/O activity, offering an additional detection signal.

9 Related Work

Anomaly-based solutions. Existing anomaly-based defenses model container behavior using syscall n-grams [12], frequency analysis [11], graph-based reasoning over syscall properties [15], replica comparison [13], executable profiling [14], or provenance graphs [16]. Among these, only CHIDS and PACED target container escapes; the others exclude such attacks from their threat models. However, CHIDS requires frequent retraining, while PACED incurs substantial overhead and false positives, limiting practical deployment.

Rule-based solutions. These systems [7; 60; 61] detect pod escapes using predefined rules. However, such rules cannot cover all escape vectors and must be continually updated as new IoCs emerge. Their broad scope also contributes to false positives [62], while tailoring them to individual microservices further limits scalability [63].

Container isolation mechanisms. Seccomp [64] and AppArmor [65] enforce security through isolation by restricting syscalls and confining program capabilities. While effective at constraining unusual behavior, both are often inapplicable in misconfigured environments (e.g., *privileged* pods), limiting their protection.

Filesystem integrity monitoring. These solutions (e.g., Tripwire [41]) rely on stable paths and checksums, assumptions that break in containerized microservices, where each deployment creates a fresh filesystem with new *inodes*. ImmuCheck instead derives a fresh baseline for each pod at `PodReady` by extracting safe writable regions, eliminating the need for training or retraining while aligning detection with the ephemeral lifecycle of pod filesystems.

10 Conclusion

In this paper, we presented ImmuCheck, a practical system for monitoring microservices to detect pod escape attacks. Unlike existing solutions, ImmuCheck enforces selective immutability, flagging unauthorized filesystem writes while allowing legitimate ones. Our evaluation, covering 12 escape scenarios across five microservices from major cloud providers, shows that ImmuCheck achieves 99.22% precision and 100% recall, outperforming state-of-the-art approaches without training or maintenance. It further incurs low overhead and provides sufficient lead time for timely response, demonstrating real-world practicality.

Acknowledgments

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. This work is supported by the SeReNity project, Grant No. cs.010, funded by Netherlands Organisation for Scientific Research (NWO). Any views, findings, or recommendations communicated in this material do not necessarily reflect the sponsors' standpoints.

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] Asbat El Khairi. *Training-less Anomaly-Based Intrusion Detection in Containerized Microservices*. PhD thesis, University of Twente, The Netherlands, 2025.

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

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

[6] Asbat El Khairi, Andreas Peter, and Andrea Continella. ConLock: Reducing Runtime Attack Surface in Containerized Microservices. In *4th International Workshop on System Security Assurance, SecAssure 2025*, 2025.

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

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

[9] Kubernetes. Production-grade container orchestration.

[10] MITRE. Escape to Host: Mitigations. https://attack.mitre.org/techniques/T1611/.

[11] 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.

[12] 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.

[13] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. ReplicaWatcher: Training-less Anomaly Detection in Containerized Microservices. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2024.

[14] Asbat El Khairi, Andreas Peter, and Andrea Continella. ProCatch: Detecting Execution-based Anomalies in Single-Instance Microservices. In *13th IEEE Conference on Communications and Network Security, CNS 2025*, 2025.

[15] 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.

[16] Mashal Abbas, Shahpar Khan, Abdul Monum, Fareed Zaffar, Rashid Tahir, David Eyers, Hassaan Irshad, Ashish Gehani, Vinod Yegneswaran, and Thomas Pasquier. Paced: Provenance-based automated container escape detection. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 261–272. IEEE, 2022.

[17] Shatanik, Bhattacharjee. Microservices architecture and design: A complete overview. https://vfunction.com/blog/microservices-architecture-guide/, 2024.

[18] Özeren Sıla. The Ten Most Common Kubernetes Security Misconfigurations & How to Address Them. https://zhenzhongxu.com/the-four-innovation-phases-of-netflixs-trillions-scale-real-time-data-infrastructure-2370938d7f01, 2024.

[19] Sysdig. Sysdig 2021 Container Security and Usage Report. https://sysdig.com/content/c/pf-2021-container-security-and-usage-report?x=u_WFRi, 2021.

[20] Linux. core(5) - linux man page. https://man7.org/linux/man-pages/man5/core.5.html.

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

[22] 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.

[23] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017.

[24] Sysdig. CVE-2022-0492: Privilege escalation vulnerability causing container escape. https://sysdig.com/blog/detecting-mitigating-cve-2022-0492-sysdig/.

[25] HackTricks. Docker breakout - privilege escalation, 2024. Accessed: 2024-10-30.

[26] Sysdig. Scarleteel: Operation leveraging terraform, kubernetes, and aws for data theft. https://www.sysdig.com/blog/cloud-breach-terraform-data-theft, 2023.

[27] Docker. OverlayFS Storage Driver. https://docs.docker.com/engine/storage/drivers/overlayfs-driver/.

[28] Bom Kim, Hyeonjun Park, and Seungsoo Lee. Kubeteus: An intelligent network policy generation framework for containers.

[29] Istio. Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/.

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

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

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

[33] Morejón Manuel. Cinema - Example of Microservices in Go with Docker, Kubernetes and MongoDB. https://github.com/mmorejon/microservices-docker-go-mongodb.

[34] Carl Knutsson, Joakim Heyman, et al. Locust: An open source load testing tool. https://locust.io/.

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

[36] Daniel Stenberg. Curl. https://curl.se/.

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

[38] Fatih Karabiber. Jaccard similarity. https://www.learndatasci.com/glossary/jaccard-similarity/, 2023.

[39] Xcube LABS. Differences between stateful and stateless containers.

[40] Linux. inotify(7) — linux manual page. https://man7.org/linux/man-pages/man7/inotify.7.html.

[41] Gene H Kim and Eugene H Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[42] Kubernetes. Container Runtime Interface (CRI). https://kubernetes.io/docs/concepts/architecture/cri/.

[43] Python. Python is a programming language that lets you work quickly and integrate systems more effectively. https://www.python.org/.

[44] The Kubernetes Authors. Kubeadm. https://kubernetes.io/docs/reference/setup-tools/kubeadm/.

[45] Andrew Randall. 7 fun flatcar facts from our community survey. https://www.flatcar.org/blog/2021/09/7-fun-flatcar-facts-from-our-community-survey.

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

[47] 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.

[48] Azure. Aks store demo. https://github.com/Azure-Samples/aks-store-demo.

[49] AWS-Containers. Aws containers retail sample. https://github.com/aws-containers/retail-store-sample-app.

[50] Application Security Cheat Sheet. https://0xn3va.gitbook.io/cheat-sheets/container/escaping/sensitive-mounts.

[51] Asaf Eitani. Threat Actors Using release_agent Container Escape. https://www.aquasec.com/blog/threat-alert-container-escape/.

[52] Ben Melamed. An Automated Response to Malicious Pod Activity. https://www.paloaltonetworks.com/blog/security-operations/an-automated-response-to-malicious-pod-activity/.

[53] Falco Talon. List of Actionners. https://docs.falco-talon.org/docs/actionners/list/.

[54] Kubernetes. Pod lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/.

[55] Kubernetes. kube-apiserver. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

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

[57] BusyBox. Busybox. https://busybox.net/.

[58] Docker. Use the btrfs storage driver. https://docs.docker.com/storage/storagedriver/btrfs-driver/.

[59] Docker. Use the ZFS storage driver. https://docs.docker.com/storage/storagedriver/zfs-driver/.

[60] NeuVector. Full lifecycle container security platform. https://neuvector.com/.

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

[62] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[63] Sysdig. Automated falco rule tuning. https://sysdig.com/blog/falco-rule-tuning/, 2021.

[64] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop (CCSW)*, 2021.

[65] Andreas Gruenbacher and Seth Arnold. AppArmor Technical Documentation, 2007.

[66] NVD. CVE-2022-0847 Detail. https://nvd.nist.gov/vuln/detail/cve-2022-0847.

[67] NVD. CVE-2022-0492 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-0492.

A ImmuCheck in Docker Environments

While our evaluation targets Kubernetes clusters, ImmuCheck is equally applicable to Docker-based deployments, including microservices orchestrated with Docker Compose or Docker Swarm. In this section, we describe how ImmuCheck can be adapted to such environments, focusing on three core aspects: (1) the deployment location, (2) the monitored regions, and (3) the initialization boundary used for baseline construction.

Deployment location. In Kubernetes, ImmuCheck operates at the node level, where the node corresponds to the physical or virtual machine hosting the Kubernetes pods. Similarly, in Docker environments, ImmuCheck functions at the host level, as the host directly manages container lifecycles without an intermediary orchestration layer.

Monitored regions. As in Kubernetes, Docker employs `OverlayFS` to construct container filesystems from layered images. The container's root filesystem (`upperdir`) associated with each container is located on the host under */var/lib/docker/overlay2/container-hash /diff*, representing all changes made during the container's lifetime. Docker also attaches host configuration files to containers via bind mounts. These include `resolv.conf`, `hostname`, and `hosts`, which reside under */var/lib/docker/containers/container-id/* on the host. Each of these files is injected into the container's namespace through explicit bind mounts. In addition, tmpfs-backed locations such as */dev* and */dev/shm* are mounted per container to provide memory-resident writable storage. While these directories are container-specific, they are accessible from the host via the container's process namespace, using paths such as */proc/container-pid/root/dev*, which also involves */dev/shm*.

Initialization boundary. Since Docker does not provide an explicit readiness signal such as Kubernetes' *PodReady* condition, ImmuCheck can adopt a practical boundary for baseline construction. For example, it may allow a brief stability period after the container enters the running state—during which filesystem activity is observed to stabilize—before capturing the baseline. This offers a lightweight yet effective approximation of readiness in Docker-based environments.

In short, ImmuCheck can also be deployed in Docker environments, where it remains essential to detect and mitigate escape attempts, particularly in cases of over-permissive containers.

B Case Study

In this section, we delve into detailed examples on how ImmuCheck detects attacks. We include one example from each category (i.e., escape vector).

Dirty Pipe (CVE-2022-0847). Dirty Pipe is a Linux kernel vulnerability that enables unauthorized modification of page-cache-backed files [66]. In the available PoC exploit, the attacker downloads a pre-compiled payload into the pod, changes its permissions, and executes it to read the host's */etc/passwd* file. The transfer of the payload, the permission modification, and its execution introduce filesystem writes that ImmuCheck flags as escape activity.

Kernel Module Injection (SYS_MOD). The attacker performs the escape in two ways: first, by downloading a custom payload that invokes `finit_module` directly, and second, by retrieving `kmod` from a remote source to handle the loading. In both cases, the kernel module itself must be introduced into the pod's root filesystem, along with permission changes and downloading-related artifacts. These actions leave detectable writes, which ImmuCheck flags as anomalous activity.

Release Agent Abuse (CVE-2022-0492). When a process ends in *cgroups* with the *notify_on_release* flag set, the kernel runs the *release_agent* file with elevated privileges [67]. In the available PoC exploit of this vulnerability, the attacker creates a directory, mounts a new *cgroup* filesystem, and enables release notifications. Next, they extract the pod's *upperdir* path from */proc/mounts*, craft a payload, and modify the *release_agent* file to execute it upon process termination in *cgroup*. ImmuCheck detects this escape early by flagging the creation of the directory used for mounting the *cgroup*, and both the creation and permission modification of the payload.