Forensic Analysis of Container Snapshot Chains for Post-Event Reconstruction

Radostin Stoyanov[a,*], Lorena Goldoni[b], Adrian Reber[c], Christopher Hargreaves[a], Rodrigo Bruno[d]

[a] *University of Oxford, Oxford, United Kingdom*
[b] *University of Modena and Reggio Emilia, Modena, Italy*
[c] *Red Hat, Stuttgart, Germany*
[d] *INESC-ID, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal*

Abstract

Container orchestration platforms have become a crucial part of the cloud-native infrastructure for deploying modern applications. The highly dynamic and ephemeral nature of these environments, however, introduces new challenges for digital forensics: malicious code often runs entirely in memory and vanishes when the container terminates, leaving no traces. The absence of forensic data can be just as dangerous as the malicious activity itself, preventing post-incident investigation and adequate response. In this paper, we propose Forensic Snapshot Chains (FSC) – a framework that transparently captures and preserves the state, configurations, and metadata of running containers. These snapshot artifacts allow investigators to accurately reconstruct and analyze the events during a security incident without impacting the running cluster. To achieve this, FSC leverages memory-tracking mechanisms inspired by live-migration optimization techniques that enable high-frequency snapshot capture when a security alert is triggered, while minimizing performance and storage overhead. Our evaluation with real-world cloud-native workloads demonstrates that FSC, with minimal performance overhead, enables accurate temporal reconstruction of memory-resident malicious activity derived from container snapshot chains under both stealthy execution and active attack scenarios.

Keywords: Container Checkpointing, Kubernetes, Forensic Readiness, Event Reconstruction, Digital Investigation

Introduction

Over the past decade, container orchestration platforms have become widely adopted to support the backbone infrastructure in many Fortune 500 companies, providing automated deployments and scaling for modern cloud-native applications. Some prominent examples include Google's search engine (Burns et al., 2016), large-scale data analytics at Goldman Sachs (Uzcategui and Hinrichs, 2019), PayPal's financial services (Qadeer and Wang, 2021), Uber's ride-hailing (Bhave and Krishnan, 2025), Tesla's and OpenAI's data pipelines (Madabushini, 2025), among many others (Wong et al., 2023). These platforms manage complex meshes of applications and services running in lightweight, ephemeral containers. The ephemeral nature of containers ensures consistency and reproducibility across development, testing, and production environments, while reducing the complexity of application scaling and orchestration. However, this same property introduces significant challenges for forensic readiness and incident response. In particular, modern attacks increasingly rely on malware that executes entirely in memory and leverages encrypted network communications that render traditional forensic techniques insufficient for post-incident investigation (Schmidt et al., 2025; Ali et al., 2025).

The Kubernetes Hardening Guide published by the U.S. National Security Agency (NSA) and the Cybersecurity and Infrastructure Security Agency (CISA) (NSA and CISA, 2022) highlights the importance of forensic readiness in such containerized environments. These recommendations reflect a proactive approach to ensure container orchestration systems and networks are prepared to efficiently collect, preserve, and analyze evidence when security incidents occur (Sachowski, 2016). This guidance was developed in response to the rapid adoption of cloud-native applications across enterprise and critical infrastructure environments, along with the growing number of security incidents and the limited ability of organizations to conduct effective post-incident investigations (Wallace and Baer, 2019; Kaczorowski and Wallace, 2019; Angel et al., 2021; Kubernetes Security Response Committee, 2026).

Prior research has largely focused on the technical feasibility of container checkpointing, including the acquisition of volatile memory and container filesystem state, as well as the interpretation of memory-resident artifacts (Gharaibeh et al., 2024). However, a single container snapshot offers limited forensic value, as it provides a static view of the container state at a specific point in time and fails to capture the temporal evolution of processes, memory contents, and service interactions. In addition, during the *post-event period* (i.e., the period between the incident event and snapshot creation), crucial forensic data may be lost or overwritten, further reducing the effectiveness of event reconstruction and the ability to infer the temporal progression of the attacker's activity (Breitinger et al., 2025). In particular, these limitations reduce the utility of individual container snapshots in real-world investigations, where understand-

*Corresponding author
Email address: radostin@stoyanov.io (Radostin Stoyanov)

ing *how* and *when* state changes occurred is vital to infer events from the past and draw evidence-informed conclusions (Gladyshev and Patel, 2004).

In this work, we present a novel forensic container checkpointing mechanism that captures and preserves a chronological sequence of snapshots over time, enabling temporal forensic analysis and event reconstruction. Our approach addresses the above limitations by supporting continuous state capture and allows investigators to correlate changes across successive snapshots. This capability enables the reconstruction of execution timelines and the relationships between events within containerized environments. We implement these mechanisms as a framework, FSC, that leverages iterative container checkpointing with OS-level memory tracking to capture successive snapshots of container execution state at short intervals. We further extend the `checkpointctl` utility with capabilities for in-depth forensic analysis of successive snapshots, enabling event reconstruction by examining the evolution of system state across snapshots.

From a forensic readiness perspective, FSC is designed to automate the capture of snapshot chains and establish temporal provenance, while supporting key principles of forensic admissibility such as authenticity, integrity, and reproducibility. The proposed snapshot chains model aligns with established memory forensics practices that enable investigators to reason about past system behavior by interpreting preserved execution state, correlate volatile artifacts across time, and infer causally related events from residual evidence, even in the absence of complete audit logs or persistent storage artifacts. This is particularly important in legal and regulatory contexts, where investigators must justify not only *what* evidence was collected but also *when* and *how* it evolved over the course of an incident.

From an investigative perspective, `checkpointctl` integrates into investigator workflows by automating the data parsing, extraction, decoding, and interpretation of the low-level execution state across container snapshots, including process metadata, memory mappings, filesystem changes, and network sockets. This approach significantly reduces the manual effort required to understand and correlate the state of isolated snapshots when analyzing memory-resident and fileless attacks in container orchestration platforms such as Kubernetes.

Through these new automated forensic container checkpointing capabilities, we not only enable investigators to analyze memory-resident and fileless attack activity in containerized environments but also establish a foundation for standardized container forensics methodology. For tool developers and researchers, our approach serves as a reference architecture and evaluation framework to validate forensic tools designed for ephemeral, cloud-native systems. More broadly, the FSC framework aims to contribute toward emerging best practices in forensic readiness for container orchestration platforms, complementing Kubernetes hardening guidance (NSA and CISA, 2022) by providing concrete mechanisms for post-incident analysis.

In summary, this paper makes the following contributions:

- We introduce a method for transparently capturing high-frequency, temporally ordered sequences of container snapshots that enable post-event reconstruction of execution timelines derived through analysis of the evolving container state.
- We design and implement FSC, a framework that realizes this methodology via chained incremental forensic snapshot acquisition based on OS-level memory tracking (soft-dirty bit) to efficiently capture only the memory pages modified since the previous snapshot.
- We extend `checkpointctl` with post-incident forensic analysis capabilities for container snapshot chains, enabling parsing, extraction, and interpretation of the captured container execution-state, including process tree, memory, filesystem changes, and network sockets.
- We evaluate FSC and `checkpointctl` through controlled case studies with real-world Kubernetes workloads, demonstrating that snapshot chains enable recovery of in-memory attack payloads, decrypted application-layer data, and transient processes that are not observable via logs or single snapshots, while remaining practical for production deployment by minimizing checkpoint latency, application frozen time, and storage overhead.

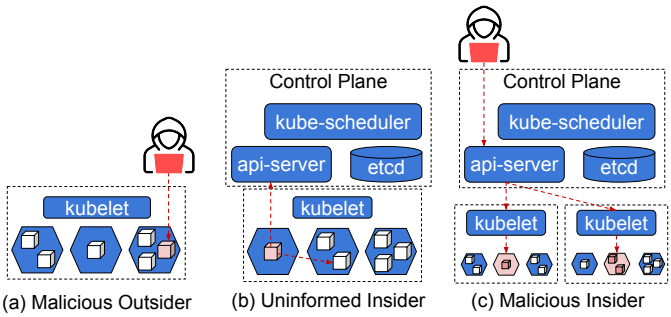

Figure 1: Threat model overview: A Kubernetes cluster with (a) an external attacker targeting a containerized application, (b) a malicious container attempting to interact with other containers and the kube-apiserver, and (c) a malicious user with control-plane access capable of deploying containers.

Background and Motivation

This section introduces container snapshots, focusing on the main limitations and challenges of using these snapshots for forensic analysis of production-grade container deployments.

Forensic Container Snapshots

Container orchestration systems for distributed applications such as Kubernetes automate many infrastructure-level operations, including scheduling, scaling, load balancing, deployment rollouts and rollbacks. These systems typically monitor running containers through an agent running on each node (e.g., kubelet), and automatically restart or reschedule failed containers to maintain the desired state. As increasingly many long-running GPU-accelerated workloads such as AI/ML model

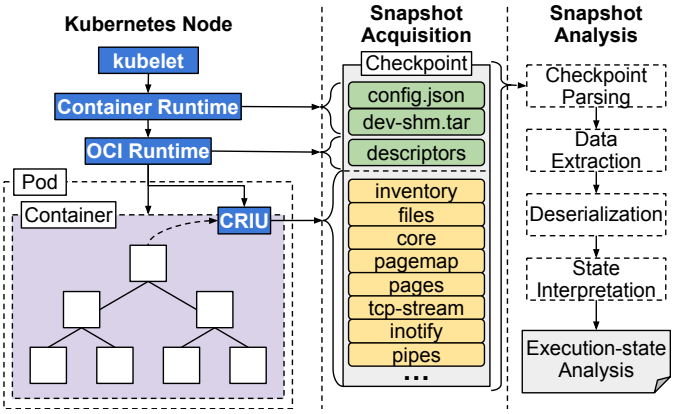

Figure 2: Overview of the forensic container snapshot analysis methodology, where the Kubernetes components are highlighted in blue, the container environment in purple, snapshot data representing the runtime state of the container in yellow, and snapshot metadata in green.

training and inference with long initialization times are being deployed with these systems, restarting containers from scratch becomes prohibitively expensive. This has led to the increased adoption of transparent container checkpointing as a fault-tolerance and live-migration mechanism in recent years.

Transparent checkpointing not only allows preserving the runtime state of applications running in the container, but also capturing any suspicious activity when anomaly detection rules have been triggered. However, container checkpoints capture comprehensive low-level information about processes (e.g., memory pages, file descriptors, sockets) that is difficult to interpret in order to extract and analyze forensic evidence.

Container checkpointing was recently introduced as a beta feature in Kubernetes to enable forensic analysis capabilities. As a result, several tools have been developed to analyze container snapshots (Kubernetes SIG Node, 2020). An initial prototype demonstrated these capabilities using `crit` (CRIU Image Tool). However, this tool was originally designed for debugging and testing the CRIU project, providing encoding and decoding functionality for protocol buffer–based images. As such, it does not support other formats, such as raw memory pages.

To extend the analysis capabilities for container checkpoints, we developed the `checkpointctl` utility [1]. This utility allows in-depth analysis of container checkpoints created by Podman as well as container runtimes for Kubernetes, such as `CRI-O` and `containerd`. It shows an overview of the snapshot, inspects its contents (e.g., open files, mount points, network sockets), and displays the content of memory pages. This tool was subsequently adopted by the research community (Gharaibeh et al., 2024) and the industry (Pellitteri and Chierici, 2024; Weaversoft.io, 2024). However, prior to this work, `crit` and `checkpointctl` were limited to analyzing individual snapshots in isolation and offered no checkpoint orchestration logic (i.e., deciding when to checkpoint), and were not able to connect multiple snapshots to form a snapshot chain. In this work, we extend the forensic analysis capabilities with cross-snapshot temporal correlation, differential state analysis, and event reconstruction of snapshot chains.

Memory Forensics. Container snapshots inherently capture process memory, placing this work within the broader field of memory forensics. Foundational work on memory acquisition and analysis (Case and Richard III, 2017; Vömel and Freiling, 2011) established methodologies for extracting forensic artifacts from volatile memory, and tools such as Volatility (Volatility Foundation, 2024) remain widely used for analyzing memory images from physical and virtual machines. Recent research has expanded memory forensics to application-specific runtime environments, including .NET (Manna et al., 2022), JavaScript engines (Wang et al., 2022), and Python (Ali et al., 2025), while others have investigated acquisition reliability, including snapshot consistency (Ottmann et al., 2023) and data remanence in virtualized environments (Savchenko et al., 2024). Our work extends these principles to containerized environments, where process memory is captured through kernel-level checkpointing rather than traditional acquisition, and where the ephemeral nature of containers makes memory-resident evidence particularly critical.

Limitations and Challenges

Forensic Data Extraction and Evidence-Driven Checkpointing. Much of the existing research on container technologies focuses on proactive security measures rather than forensic investigations. Existing checkpointing mechanisms lack systematic strategies for deciding when checkpoints should be captured, what components of container state are forensically relevant, and how evidence should be preserved to maintain verifiable integrity. Without evidence-driven checkpointing, practitioners are forced to rely on ad-hoc approaches that may miss critical transient state or fail to establish verifiable evidence provenance.

Sequential Snapshot Analysis and Checkpoint Continuity. In other contexts, e.g., the Volume Shadow Copy Service in Windows (Hargreaves et al., 2008), snapshots are supported by forensic tools that can aggregate versioned data, and versioning file systems such as APFS have been explored from a forensic recovery perspective (Plum and Dewald, 2018). However, container checkpointing tools treat checkpoints as isolated, point-in-time artifacts. There are no existing mechanisms for linking successive checkpoints to preserve temporal ordering, provenance, and integrity. This prevents reliable reconstruction of execution timelines and correlation of state changes across checkpoint iterations.

Performance and Storage Overhead. Creating complete snapshots at high frequency incurs significant overhead, both in compute resources and application frozen time. While full-state capture is important for fault tolerance and live migration, the amount of memory state that changes between successive checkpoints over short intervals is typically small. Efficiently tracking and capturing only modified state is necessary to en-

[1] https://github.com/checkpoint-restore/checkpointctl

able high-frequency forensic checkpointing in production environments.

Protecting Sensitive Data. Container checkpoints often contain sensitive data such as passwords, tokens, and session keys that can lead to leaks across trust boundaries (Stoyanov et al., 2024). Existing systems assume a trusted execution environment and leave encryption to the user. A naive encryption approach is impractical due to performance overheads for large snapshots, leaving an important research gap for encryption compatible with forensic analysis and incremental checkpointing.

Threat Model and Assumptions

As illustrated in Figure 1, our threat model is concerned with the following three primary types of actors:

- *Malicious outsider:* an actor without authorization, able to launch attacks via the internet, supply chain, or physical perimeter to compromise containerized applications.
- *Uninformed insider:* an authorized actor who unintentionally introduces security risks, e.g., by deploying a container image containing malicious code without realizing it has been compromised.
- *Malicious insider:* an authorized actor (e.g., with Kubernetes API access) who intentionally deploys arbitrary containers and Pods within the cluster.

While there are other actors that may interact with the modeled system (e.g., uninformed outsiders), the controls for their actions are a subset of those for the primary actors listed above.

Threat scenarios. The first scenario corresponds to a malicious outsider, who exploits vulnerabilities to compromise the confidentiality or integrity of a container's private data or to escape the container's isolation boundaries. In the second scenario, aligned with an uninformed insider, the attacker relies on a compromised or intentionally malicious container image to obtain unauthorized behavior at runtime, such as establishing a persistent backdoor or attempting to escalate privileges within the cluster. In the third scenario, corresponding to a malicious insider, the attacker has authorized access to the container platform but does not initially control the underlying host operating system and instead attempts to exploit OS-level or container runtime vulnerabilities to access or tamper with data belonging to other containers.

Assumptions and trust boundary. We assume that the legitimate application running in a container does not expose its own private data. However, attacks originating from other compromised containers, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised operating system, as well as physical or side-channel attacks, are out of scope. The trust boundary of FSC is the OS system call API, which enables adversaries to observe certain aspects of OS interactions, such as sizes and offsets. Consequently, the host kernel, CRIU, and the container runtime (e.g., CRI-O, containerd) are within the trusted computing base. If an attacker compromises any of these components (e.g., through a container escape vulnerability leading to host-level access) the integrity of the evidence chain cannot be guaranteed.

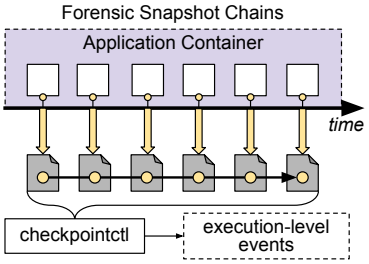

Figure 3: Overview of FSC snapshot acquisition and `checkpointctl` forensic analysis workflow. A sequence of immutable container checkpoints preserves runtime state at well-defined points in time, enabling post-incident forensic reconstruction.

Malicious container images. Adversaries, particularly uninformed or malicious insiders, may further inject malicious code (e.g., backdoors) into container images and circumvent container registry checking mechanisms. While the creation of such container images is outside the scope of this paper, prior work has shown that existing registries may contain many malicious or vulnerable images (Shu et al., 2017). To make such attacks more stealthy, an attacker may repackage a container image with malicious code while preserving the original functionality, making the compromise difficult for users to detect.

Forensic Container Snapshot Chains

FSC captures a sequence of container snapshots at high-frequency that preserve a complete view of the runtime state of all processes running in a container at specific points in time. Each snapshot is immutable and cryptographically protected, providing forensic soundness by ensuring integrity and resistance to post-acquisition tampering. Specifically, FSC computes a SHA-256 hash over the contents of each snapshot and includes the hash of the preceding snapshot in the chain, forming a tamper-evident linked sequence. Any modification to a snapshot invalidates all subsequent hashes, enabling investigators to verify chain integrity and detect post-acquisition tampering. Complementary to FSC, our extensions to `checkpointctl` support post hoc forensic examination of snapshot chains without altering the original evidence. Figure 3 illustrates how FSC and `checkpointctl` work together to create snapshot chains and process them to identify execution-level events.

Snapshot chains enable investigators to reason about past system behavior by analyzing both individual snapshots and their evolution over time. Because snapshots are continuously written to persistent storage during execution, the forensic chain remains available for retrospective analysis even after the original container has been terminated, restarted, or deleted. This

approach supports reconstruction of execution timelines and identification of malicious activity even in environments where traditional execution logs are unavailable or incomplete.

Forensic Artifacts Preserved in Container Snapshots

Each snapshot captures a comprehensive set of system-level artifacts that collectively reflect the execution state of the container at the time of acquisition. These artifacts constitute digital evidence from which prior actions may be inferred.

Process Tree. Container snapshots preserve the complete process tree, including parent–child relationships, process identifiers, and process lifetimes. Across a snapshot chain, this information enables inference of process creation order and termination. For example, Figure 6 demonstrates how consecutive snapshots reveal the emergence and expansion of attacker-controlled processes following a remote code execution compromise.

Container Filesystem. Snapshots capture the writable layer of the container filesystem together with references to the underlying OCI image. Newly created or modified files, open file descriptors, and memory-mapped files provide evidence of filesystem-level activity.

Containers commonly employ OverlayFS, where only the top layer is writable. Deletions or replacements of files originating from read-only layers are recorded using *whiteout* files, enabling investigators to determine how the filesystem was altered during execution.

Memory State. Snapshots preserve the memory state of running processes, reflecting the effects of executed instructions and data processing prior to acquisition. Because memory images may be large and contain substantial irrelevant data, FSC employs CRIU's incremental checkpointing mechanisms based on the Linux *soft-dirty* and *page-map* interfaces (see Figure 4).

Only memory pages modified since the previous snapshot are captured, reducing snapshot size and acquisition latency. Changes in memory content across snapshots provide temporal evidence of execution behavior, including code execution paths, data handling, and attacker interactions.

Network Sockets. Snapshots record the state of all network sockets, including *listening* and *established* connections. When analyzed across snapshot chains, socket state and creation timestamps provide evidence of when services began listening and when external clients connected.

Although network communications may be encrypted in transit, snapshots preserve decrypted application-layer data as it exists in process memory after transport-layer decryption. This enables recovery of request contents, responses, credentials, and other sensitive data processed by the application. Figure 5 illustrates this capability in the context of an SQL injection attack.

Snapshot Metadata. Each snapshot includes metadata associated with processes, files, and network objects, such as timestamps, ownership, permissions, and configuration parameters.

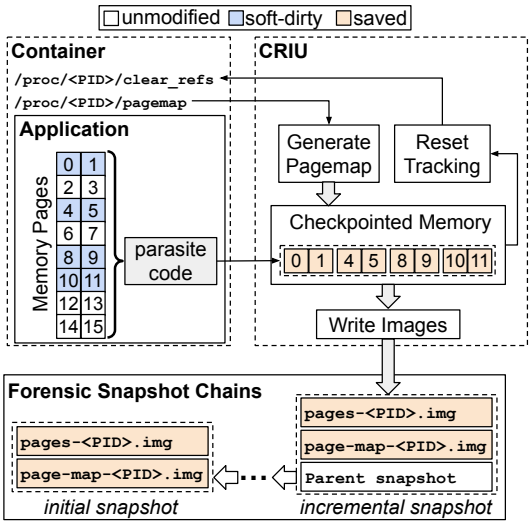

Figure 4: Incremental memory tracking used by FSC to construct forensic snapshot chains while minimizing snapshot size and acquisition overhead.

When interpreted collectively, these attributes provide contextual evidence that supports temporal inference and reconstruction of higher-level activities.

Temporal Analysis Across Snapshot Chains

While individual snapshots capture system state at a single point in time, forensic reconstruction relies on analysis across snapshot chains. FSC performs temporal analysis by comparing inferred artifact sets between successive snapshots to identify newly introduced, modified, or removed artifacts.

Artifacts observed at time t_{i+1} but not at t_i imply actions occurring after t_i, while artifacts present at t_i but absent at t_{i+1} imply actions occurring before t_{i+1}. Applying this reasoning iteratively across a snapshot chain yields a partial ordering of inferred events consistent with acquisition times and artifact dependencies.

Correlation and Causal Linking. Temporal ordering is augmented through correlation across subsystems and abstraction layers. FSC correlates events using shared identifiers (e.g., process identifiers, file descriptors, socket tuples), temporal proximity, and established system semantics. These correlations enable reconstruction of higher-level activities—such as request handling, command execution, or data exfiltration—from low-level forensic artifacts, even when evidence is fragmented across snapshots.

Handling Gaps and Inconsistencies. Snapshot-based forensic analysis inevitably encounters gaps where no direct evidence exists for certain actions. Rather than treating these gaps as analytical failures, FSC models them as bounded intervals between known events. Inconsistencies, such as overlapping artifact lifetimes or conflicting timestamps, are explicitly identified and attributed to factors including snapshot delays or container runtime behavior. Explicitly modeling uncertainty strengthens the defensibility and transparency of the reconstructed timeline.

FSC transforms temporal analysis of static container snapshots into a structured forensic chronology. By inferring events

from preserved state, ordering them across snapshot chains, and correlating them into higher-level activities, FSC supports post-incident forensic reasoning in containerized environments without reliance on continuous monitoring or complete audit logs.

Experimental Evaluation

In this section, we present the results of our experiments. To carry out our evaluation and to validate FSC, we deployed a Kubernetes cluster on a two-node setup representative of a typical production environment configured with CRI-O v1.32.12 and CRIU v4.2, running Ubuntu 22.04. Both machines provide 52 CPU cores, 62 GB of memory, and 3 TB of disk space.

As described previously, the forensic snapshot chains combine CRIU images capturing the complete runtime state of all Linux processes within the checkpointed container, along with the container's filesystem writable layer, and associated container metadata and configurations. This metadata includes the container image, container configuration, container creation and checkpoint timestamps, network configuration, as well as Kubernetes Pod name, namespace, annotations, and volumes. FSC automates the snapshot creation mechanism at pre-defined intervals, allowing investigators to capture continuous forensic evidence over time while minimizing the disruption to running workloads. `checkpointctl` enables the forensic analysis of these snapshots by interpreting the captured execution state and extracting human-readable representations of the runtime state of all processes, snapshot metadata, and facilitating in-depth memory analysis.

Evaluation Criteria

Our evaluation is designed to assess FSC and `checkpointctl` along two dimensions: (i) extraction accuracy, evaluating the ability to recover memory-resident artifacts and reconstruct events in a manner consistent with forensic soundness and ground truth timelines; and (ii) runtime overhead, measuring the performance impact of periodic snapshotting on running containers, including checkpoint latency, application frozen time, and storage footprint. To this end, we deployed a Kubernetes cluster as described above and executed a set of controlled container workloads representative of common cloud-native applications, including interactive applications, short-lived jobs, and long-running services. Each workload was deployed as one or more Pods, and exercised with scripted activity to induce observable process creation, termination, filesystem modifications, network interactions, and in-memory state changes. Deterministic ground-truth event timelines were recorded externally to enable validation of the reconstructed events.

Across all case studies, existing forensic methods in Kubernetes—such as audit logs, container logs, and single-snapshot analysis (Gharaibeh et al., 2024)—provide limited visibility into application-level attack semantics and lack the temporal context required to reconstruct how an attack unfolded over time.

FSC was configured to automatically create forensic snapshots at fixed time intervals throughout each experiment. Each

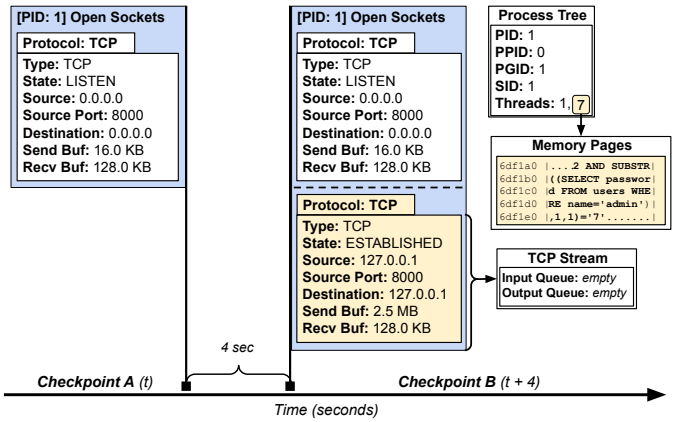

Figure 5: Temporal analysis of two sequential snapshots capturing an SQL injection attack. The checkpoint data reveals a newly created process thread and an established TCP connection with empty input/output queues. We identify and extract the attacker's payload from the memory of the newly created thread, and show that the extracted payload performs a test to determine whether the first character of the administrator's password is the digit 7.

snapshot captures the complete runtime state of all processes within the target container using CRIU, along with the container's writable filesystem layer and associated container and Kubernetes metadata. No manual intervention was required during snapshot acquisition, and containers continued executing without being stopped or restarted.

For offline static analysis, we applied `checkpointctl` to the collected snapshots, without restoring the containers or replaying execution. Using `checkpointctl`, we extracted process tree hierarchies, open files, network sockets, command line arguments, environment variables, plaintext content of memory pages, and snapshot metadata, and correlated artifacts across consecutive snapshots to reconstruct temporal process behavior and higher-level events. Memory analysis was performed directly on checkpointed process memory to recover in-memory artifacts relevant to the forensic investigation.

Performance measurements were collected by the container runtime and checkpointing engine to measure impact on application execution. Snapshot sizes and associated storage requirements were obtained using standard Linux filesystem tools. All experiments were repeated ten times to ensure reproducibility, and the reported results reflect representative values observed across independent runs.

Case Study: SQL Injection Attack

Many web applications and databases are deployed as services running in container orchestration platforms such as Kubernetes, where SQL injection remains a prevalent attack vector. To demonstrate the efficacy of FSC and `checkpointctl` in collecting, preserving and analyzing digital evidence, this case study explores classic SQL injection flaws. We employed a Damn Small Vulnerable Web (DSVW) container image featuring popular web application vulnerabilities and attacks[2], where

[2] https://github.com/stamparm/DSVW

the injection occurs when user input is directly concatenated into an SQL query string. As a result, the attacker can change the logic of the query by specifying SQL code as input data. In particular, the attacker changes the condition logic and appends additional SQL clauses that force the query to return extra rows. As a side-effect, the attacker can observe output changes and reconstruct secrets (e.g., session keys, tokens, passwords).

As illustrated in Figure 5, by capturing a sequence of snapshots at high frequency and correlating these artifacts with audit log analysis, FSC enables investigators to examine and reconstruct a fine-grained execution timeline of the malicious activity across multiple layers of the orchestration system. Checkpoint chains allow investigators to analyze and correlate incoming requests with the corresponding application code paths and subsequent database service interactions. As these chained checkpoints capture the memory pages, open network sockets, and TCP send/receive buffer data for all processes running in the container, it becomes possible to extract the exact malicious input, even when end-to-end encryption has been used by the attacker. In such cases, the captured memory page contents reveal the content of the attacker's query or payload after decryption, allowing extraction of the original attacker request or payload.

The key benefit of forensic analysis with snapshot chains is their ability to preserve the temporal context across the full containerized execution of an application-level attack. By maintaining a continuous record of this system state, snapshot chains enable traceability of malicious inputs from the point of ingress through vulnerable application logic and into backend database interactions. This enables not only identifying the affected processes and code paths, but also determining which sensitive data was accessed or exposed, without requiring prior application instrumentation or additional logging.

Case Study: Remote Code Execution

Remote code execution represents one of the most critical security risks in cloud-native environments, where an attacker is able to execute arbitrary commands inside a target container. To evaluate the capability of FSC to capture and reconstruct the set of events and actions performed in this context, we deployed a web application susceptible to command injection. This vulnerability occurs because user-supplied input is directly concatenated into a shell command being executed, allowing special characters to be interpreted by the operating system shell. In particular, an attacker can inject additional commands (e.g., using semicolon or ampersand symbols) and cause the application to execute arbitrary commands with the privileges of the web server process.

By analyzing snapshot chains generated by FSC, `checkpointctl` enables detailed reconstruction of the remote code execution attack. As shown in Figure 6, consecutive snapshots reveal the creation of new processes that were not part of the original container image, including the attacker-initiated reverse shell with auxiliary utilities such as Netcat. By correlating the process trees and memory pages across snapshots, investigators can identify the parent-child relationships between the vulnerable application and subsequently spawned attacker-controlled processes. A forensic memory analysis of the snapshots allows extraction of command strings, environment variables, and network parameters used to establish reverse shells, while captured socket state reveals outbound connections initiated by the attacker. This temporal correlation enables reconstruction of the exact sequence of injected commands, process executions, and network interactions during the attack.

The key benefit of using snapshot chains in a remote code execution scenario is their ability to preserve the full temporal evolution of attacker activity inside a compromised container. This enables attribution and impact analysis without application-level instrumentation or additional logging.

Case Study: Fileless Persistence in Containers

Adversaries often leverage the `at` utility to establish recurring execution of malicious code within containerized environments (MITRE ATT&CK, 2025). In this case study, we explore how the three primary actors (malicious outsider, uninformed insider, malicious insider) possess the capability to inject code into a target container through a command injection attack, remote shell backdoor, and `kubectl exec`, respectively. Following initial code injection, the attacker installs a scheduled (malicious) task that persists beyond the initial access.

As defined in the threat model, adversaries are assumed to possess the capability to inject code into the target container via exploitation of a remote code execution vulnerability, interactive access through `kubectl exec`, or the introduction of malicious artifacts via a mounted volume. Following initial code injection, scheduled execution can provide a comparatively stealthy persistence mechanism, as such activity may resemble legitimate administrative behavior, produce limited forensic artifacts, and evade detection approaches that focus primarily on container entrypoints, runtime processes, or externally triggered execution events.

In addition, adversaries can further reduce forensic visibility by executing obfuscated, memory-resident code where the payloads are decoded and executed at runtime. In such cases, scheduled jobs can be leveraged to exfiltrate sensitive data from the target application's memory address space (e.g., a web server process) without persisting artifacts to the container filesystem. This approach limits opportunities for static analysis and reduces the availability of durable forensic evidence, increasing reliance on volatile data sources such as memory captures, host-level telemetry, and runtime monitoring for detection and reconstruction.

To evaluate FSC in this scenario, we created a controlled containerized workload based on an NGINX web server and injected a malicious task using the `at` utility to periodically execute a short-lived payload that attaches to the running web server process and extracts sensitive data from its address space, simulating data exfiltration without writing artifacts to disk. Concretely, the scheduled task is registered via `echo "python3 -c $(echo <base64-encoded-payload> | base64 -d)" | at now`, which decodes and executes the payload at the scheduled time without writing it to the filesystem. The Python payload uses `/proc/<pid>/mem` to directly read memory regions of the target NGINX worker

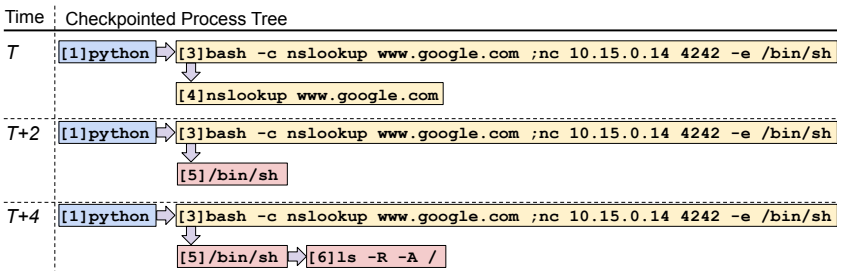

Figure 6: Temporal analysis of the container process-tree across three sequential snapshots, generated at 2-second intervals, capturing a remote code execution attack that establishes a reverse shell (using Netcat) and enables arbitrary code execution. The main web application process (PID 1) is shown in blue, request-handling processes (PID 3 and 4) in yellow, and attacker-initiated processes (PID 5 and 6) in red.

process, extracting sensitive data such as request headers and session tokens from the worker's heap. The payload operates entirely in memory, terminates after execution, and leaves no persistent filesystem artifacts.

We then assessed whether FSC and `checkpointctl` were able to (i) identify the presence of the scheduled task, (ii) recover the in-memory execution artifacts associated with the payload, and (iii) accurately reconstruct the temporal sequence of events across successive snapshots, despite the absence of persistent filesystem indicators.

FSC captured a chain of 19 incremental snapshots over a 43-second window (approximately 2.4-second intervals). Analysis of the snapshot chain revealed the scheduled task registered with the `at` daemon and the emergence of a short-lived `python3` process executing the decoded payload. By correlating process trees and memory contents across successive snapshots, `checkpointctl` recovered the decoded Python payload from process memory, revealing the `/proc/<pid>/mem` read operations targeting NGINX worker heap regions. The snapshot chain further captured the outbound HTTP connection used to exfiltrate the extracted memory contents, including the destination address and transmitted data. These artifacts were not observable through container logs or filesystem inspection, as the payload was decoded at runtime, operated entirely in memory, and terminated after execution.

Overhead Analysis

We evaluate the runtime and storage overhead introduced by forensic snapshot acquisition to assess the practicality of deploying FSC in production Kubernetes environments. Our analysis focuses on checkpoint latency, application frozen (paused) time during snapshot creation, and the resulting snapshot storage footprint.

Checkpoint latency. A critical factor in determining the feasibility of forensic snapshotting in production environments is checkpoint latency, as it directly affects application responsiveness. As shown in Table 1, the freezing time introduced by FSC remains consistently low across both workloads. For the lightweight DSVW application, freezing completes in under 0.5 ms and remains below 9 ms for NGINX. Consequently, this initial phase of the snapshot operation, required to create a consistent snapshot, introduces negligible delay for long-running services.

Table 1: Performance and storage overhead of snapshot acquisition, comparing a single full checkpoint (Baseline) with incremental snapshot chains (FSC).

Workload	Method	Freezing (ms)	Frozen (ms)	Memdump (ms)	Storage
DSVW	Baseline	0.43	255.7	15.5	8.6 MB
	FSC	0.37	240.3	6.6	553 KB
NGINX	Baseline	8.38	1271.6	384.3	217 MB
	FSC	8.08	1194.9	189.6	15 MB

Once frozen, container processes remain paused for the duration of the checkpoint operation. The frozen time in Table 1 reflects the resulting application downtime, including the time spent persisting memory pages (memdump) to storage. Leveraging memory tracking and incremental checkpointing, FSC reduces the amount of memory that must be written while processes are paused, leading to an approximately 6% reduction in frozen time and a consistent decrease in memdump time compared to baseline full checkpoints across workloads.

Storage overhead. Another key consideration for the practical deployment of forensic snapshot chains is the storage overhead. As shown in Table 1, preserving full snapshots (baselines) incurs a significant amount of disk space. This overhead is particularly important when snapshotting memory-intensive services and can quickly become prohibitive when snapshots are taken very frequently. In contrast, the incremental snapshot chains generated by FSC substantially reduce storage requirements by saving only the modified memory pages for each snapshot. For NGINX, snapshot size is reduced from hundreds of megabytes to tens of megabytes, while for DSVW it decreases by an order of magnitude. These reductions enable efficient high-frequency snapshot acquisition of the forensic evidence without incurring excessive storage costs. At 3-second intervals, incremental snapshot chains accumulate approximately 663 MB per hour for the DSVW workload and 18 GB per hour for NGINX, making retention policies and tiered storage essential for sustained deployment.

Discussion

While the experimental results demonstrate that incremental forensic snapshot chains enable accurate event reconstruction

with low performance overhead, real-world investigations often encounter additional challenges that require consideration.

Resilience Against Anti-Forensic Techniques

A key concern in memory and container forensics is the increasing use of anti-forensic techniques designed to evade detection, obscure attacker activity, or destroy evidence. In containerized environments, such techniques commonly include fileless malware execution, rapid process creation and termination, encrypted network communications, and deliberate manipulation of application-level logs.

FSC mitigates several of these techniques by design. Because checkpoints capture raw process memory, open file descriptors, and socket state, attackers cannot rely solely on log deletion, short-lived processes, or encrypted transport protocols to conceal activity. As demonstrated in the case studies, decrypted application-layer payloads and in-memory command execution traces remain recoverable even when network traffic is encrypted and filesystem artifacts are absent.

However, FSC does not render all anti-forensic techniques ineffective. An attacker could exploit snapshot granularity by executing malicious actions entirely between checkpoint intervals, and deliberate memory spraying could introduce noise into incremental memory diffs. While increasing checkpoint frequency and using random intervals reduce these risks, residual windows of opportunity remain.

In addition, FSC assumes the integrity of the host kernel and checkpointing subsystem. Kernel-level rootkits, compromised CRIU components, or manipulation of checkpoint metadata could undermine the trustworthiness of collected evidence. While snapshot chaining enables investigation within the snapshot sequence, preventing or detecting active interference during snapshot creation remains an open challenge. Addressing such adversaries would require complementary host-based attestation, trusted execution environments, or hardware-backed integrity mechanisms, which are beyond the scope of this work.

Error Handling During Checkpointing

In practice, checkpointing is not a failure-free operation. Errors may arise due to unsupported kernel features (e.g., system calls) or interactions with external device drivers (e.g., GPUs). These failures can lead to partially captured state and incomplete snapshots.

FSC tolerates such imperfections by modeling gaps as bounded intervals during which unobserved actions may have occurred, aligning with the forensic principle that absence of evidence constrains inference rather than constituting evidence of absence. Cryptographic snapshot chaining allows investigators to verify which snapshots are present and reason about the implications of missing data. In our evaluation, isolated checkpoint failures did not prevent reconstruction of higher-level attacker activity. Nevertheless, frequent failures reduce temporal resolution, underscoring the importance of checkpoint scheduling and health monitoring as part of a forensic readiness strategy.

Implications for Forensics in Cloud-Native Systems

FSC illustrates a shift from artifact-centric forensics toward state-centric and time-aware analysis in cloud-native systems. Traditional forensic methods rely heavily on persistent storage artifacts and centralized logs, both of which are increasingly unreliable in ephemeral containerized workloads. By contrast, snapshot chains allow investigators to reconstruct behavior from state transitions, even when explicit audit trails are missing or intentionally suppressed.

This capability has important implications for incident response workflows. First, it complements real-time detection and logging by enabling meaningful retrospective analysis when such mechanisms are unavailable or incomplete. Second, it supports defensible expert testimony by preserving temporal provenance and explicitly representing uncertainty. Finally, it enables forensic analysis of workloads that are otherwise difficult to investigate after termination.

At the same time, the approach raises practical considerations around data volume, privacy, and multi-tenant attribution. Forensic snapshots may contain sensitive user data unrelated to an incident, complicating legal and regulatory compliance. While these issues are not unique to FSC, the richness of captured memory state amplifies their importance and motivates future work on selective capture, access control, and privacy-preserving analysis.

Limitations and Future Directions

Despite its advantages, FSC has several limitations. The framework currently focuses on single-container analysis and does not natively correlate events across multiple pods, nodes, or clusters. In real-world Kubernetes incidents, attacks frequently span multiple containers within a Pod—including sidecars and init containers—as well as across nodes. Temporally aligning and correlating snapshot chains from different containers requires shared clock references and cross-container event linking, which remain open research challenges. Additionally, while `checkpointctl` automates much of the low-level analysis, expert interpretation is still required to construct high-level narratives and assess intent. Application-specific memory deserialization (e.g., language runtime structures) is not yet supported; `checkpointctl` currently relies on raw string and byte extraction.

Our evaluation uses lightweight workloads (DSVW and NGINX) on a two-node cluster. Memory-intensive workloads such as databases, JVM-based applications, or ML training containers would increase both checkpoint latency and storage overhead. In large-scale deployments with hundreds of containers, storage growth from high-frequency snapshot chains requires careful retention policies and tiered storage strategies to remain operationally feasible. Future evaluation should characterize these scaling properties across diverse workload profiles.

The selection of checkpoint intervals involves a fundamental tradeoff between temporal resolution and performance overhead. Our evaluation uses intervals of 2 to 3 seconds, which reliably capture transient processes and short-lived payloads; longer intervals would reduce overhead but risk missing

ephemeral artifacts. The optimal frequency depends on workload characteristics and the expected pace of attacker activity. Adaptive strategies that dynamically adjust checkpoint frequency based on anomaly detection signals could improve this tradeoff, though such mechanisms introduce additional complexity.

Furthermore, containers that have been terminated or deleted before checkpoint acquisition cannot be analyzed through snapshot chains. In highly dynamic environments with autoscaling and rolling updates, proactive integration with orchestration-level telemetry could help identify and prioritize containers for checkpointing before they are removed.

Future work will explore adaptive checkpointing strategies driven by anomaly signals, integration with cluster-wide telemetry, and stronger integrity guarantees using hardware-backed trust. Investigating defenses against kernel-level anti-forensics and developing standardized representations for snapshot-based timelines are also promising directions.

Overall, FSC demonstrates that forensic checkpoint chains can substantially improve post-incident visibility in Kubernetes environments, but it should be viewed as a complementary capability rather than a standalone solution. Its greatest value emerges when combined with broader forensic readiness practices, including secure configuration, monitoring, and incident response planning.

Conclusions

Container orchestration platforms such as Kubernetes are crucial for deploying and scaling modern cloud-native applications. However, the highly dynamic and ephemeral nature of these container environments introduces significant challenges for digital forensics by rapidly creating, rescheduling, and terminating containers, often leaving little persistent state behind for post-incident analysis. As a result, traditional forensic techniques are often insufficient to capture and preserve relevant evidence in cloud-native environments. Volatile artifacts such as in-memory data, ephemeral storage layers, and short-lived network flows may disappear within seconds, while orchestration-level operations can further obscure provenance and event timelines.

In this work, we presented FSC, a framework for forensic snapshot chain analysis that enables retrospective investigation of containerized workloads through transparent, incremental checkpointing. By leveraging kernel-level memory tracking, FSC captures the evolving container execution state over time with low performance and storage overhead, preserving volatile evidence that would otherwise be lost. We further extend `checkpointctl` with forensic analysis capabilities for snapshot chains that enable the reconstruction of execution timelines and memory-resident artifacts.

Our evaluation on Kubernetes demonstrates that FSC and `checkpointctl` can accurately recover process activity, filesystem changes, network interactions, and in-memory attack payloads under malicious workloads, while maintaining low performance and storage overhead. The results show that even fileless attacks and encrypted network communications leave recoverable traces in checkpointed memory, enabling temporal reconstruction of attacker behavior when traditional logs are unavailable. FSC and `checkpointctl` do not replace existing detection or monitoring mechanisms; instead, they provide complementary forensic capabilities that support forensic readiness in dynamic containerized environments. We believe forensic checkpoint chains represent a practical foundation for future research and operational tooling in Kubernetes forensics.

Acknowledgments

We acknowledge Kouamé Behouba Manassé and Prajwal Nadig for their contributions to the implementation of checkpoint inspection and memory parsing capabilities in the `checkpointctl` tool. This work was supported in part by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/PRR/50021/2025, LISBOA2030-FEDER-00748300, UID/50021/2025, and by the EU's Horizon Europe research and innovation programme under Grant Agreement No. 101189689.

CRediT authorship contribution statement

Radostin Stoyanov: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing, Visualization. **Lorena Goldoni:** Conceptualization, Methodology, Investigation, Software. **Adrian Reber:** Conceptualization, Methodology, Software. **Christopher Hargreaves:** Conceptualization, Methodology, Resources, Validation, Writing – review & editing. **Rodrigo Bruno:** Conceptualization, Methodology, Resources, Validation, Writing – original draft, Writing – review & editing, Supervision.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT and Claude in order to revise, condense text, and correct grammatical errors, typos, and awkward phrasing. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Ali, H., Case, A., Ahmed, I., 2025. Memory analysis of the python runtime environment. Forensic Science International: Digital Investigation 53, 301920. URL: https://www.sciencedirect.com/scienc

e/article/pii/S2666281725000599, doi:https://doi.org/10.1016/j.fsidi.2025.301920. dFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA.

Angel, J., Joglekar, P., Raghunathan, S., 2021. A closer look at nsa/cisa kubernetes hardening guidance. https://kubernetes.io/blog/2021/10/05/nsa-cisa-kubernetes-hardening-guidance/. Accessed: 2025-01-24.

Bhave, A., Krishnan, A., 2025. Migrating uber's compute platform to kubernetes: A technical journey. https://www.uber.com/blog/migrating-ubers-compute-platform-to-kubernetes-a-technical-journey/. Accessed: 2026-01-20.

Breitinger, F., Studiawan, H., Hargreaves, C., 2025. Sok: Timeline based event reconstruction for digital forensics: Terminology, methodology, and current challenges. Forensic Science International: Digital Investigation 53, 301932. URL: https://www.sciencedirect.com/science/article/pii/S266628172500071X, doi:https://doi.org/10.1016/j.fsidi.2025.301932. dFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J., 2016. Borg, omega, and kubernetes. Commun. ACM 59, 50–57. URL: https://doi.org/10.1145/2890784, doi:10.1145/2890784.

Case, A., Richard III, G.G., 2017. Memory forensics: The path forward. Digital Investigation 20, 23–33. doi:10.1016/j.diin.2016.12.004.

Gharaibeh, T., Seiden, S., Abouelsaoud, M., Bou-Harb, E., Baggili, I., 2024. Don't, stop, drop, pause: Forensics of container checkpoints (conpoint), in: Proceedings of the 19th International Conference on Availability, Reliability and Security, Association for Computing Machinery, New York, NY, USA. pp. 1–11. URL: https://doi.org/10.1145/3664476.3670895, doi:10.1145/3664476.3670895.

Gladyshev, P., Patel, A., 2004. Finite state machine approach to digital event reconstruction. Digital Investigation 1, 130–149.

Hargreaves, C., Chivers, H., Titheridge, D., 2008. Windows vista and digital investigations. Digital Investigation 5, 34–48.

Kaczorowski, M., Wallace, A., 2019. Container forensics: What to do when your cluster is a cluster, in: Proceedings of KubeCon & CloudNativeCon Europe 2019, Cloud Native Computing Foundation. Talk slides / video available online, Accessed: 2026-01-24.

Kubernetes Security Response Committee, 2026. Official kubernetes cve feed. https://kubernetes.io/docs/reference/issues-security/official-cve-feed/. Accessed: 2026-01-24.

Kubernetes SIG Node, 2020. Kep 2008: Forensic container checkpointing. https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/2008-forensic-container-checkpointing. Accessed 2025-12-30.

Madabushini, P., 2025. Harnessing kubernetes for scalable ai/ml workloads: Insights from tesla and openai. Conf42 Machine Learning 2025 Online Presentation. Online conference talk.

Manna, M., Case, A., Ali-Gombe, A., III, G.G.R., 2022. Memory analysis of .net and .net core applications. Forensic Science International: Digital Investigation 42, 301404. URL: https://www.sciencedirect.com/science/article/pii/S2666281722000853. dFRWS USA 2022.

MITRE ATT&CK, 2025. Scheduled task/job: At (T1053.002). https://attack.mitre.org/techniques/T1053/002/. Accessed: 2026-01-24.

NSA, CISA, 2022. Kubernetes Hardening Guide. Technical Report U/OO/168286-21; PP-22-0324. National Security Agency and Cybersecurity and Infrastructure Security Agency. URL: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF. version 1.2.

Ottmann, J., Üsame Cengiz, Breitinger, F., Freiling, F., 2023. As if time had stopped – checking memory dumps for quasi-instantaneous consistency. Forensic Science International: Digital Investigation 46, 301611. DFRWS USA 2023.

Pellitteri, A., Chierici, S., 2024. Csi forensics: Unraveling kubernetes crime scenes. Talk at CloudNativeSecurityCon North America 2024.

Plum, J., Dewald, A., 2018. Forensic apfs file recovery, in: Proceedings of the 13th International Conference on Availability, Reliability and Security, pp. 1–10.

Qadeer, A., Wang, C., 2021. Trimaran: Real load aware scheduling in kubernetes. Presentation at KubeCon + CloudNativeCon North America 2021.

Sachowski, J., 2016. Implementing Digital Forensic Readiness. Syngress, Boston. URL: https://www.sciencedirect.com/book/9780128044544/implementing-digital-forensic-readiness, doi:10.1016/B978-0-12-804454-4.

Savchenko, E., Ottmann, J., Freiling, F., 2024. In the time loop: Data remanence in main memory of virtual machines. Forensic Science International: Digital Investigation 49, 301758. URL: https://www.sciencedirect.com/science/article/pii/S2666281724000775, doi:10.1016/j.fsidi.2024.301758. dFRWS USA 2024.

Schmidt, L., Strasda, S., Schinzel, S., 2025. Uncovering linux desktop espionage. Forensic Science International: Digital Investigation 53, 301921. URL: https://www.sciencedirect.com/science/article/pii/S2666281725000605, doi:https://doi.org/10.1016/j.fsidi.2025.301921. dFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA.

Shu, R., Gu, X., Enck, W., 2017. A study of security vulnerabilities on docker hub, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Association for Computing Machinery, New York, NY, USA. p. 269–280. URL: https://doi.org/10.1145/3029806.3029832, doi:10.1145/3029806.3029832.

Stoyanov, R., Reber, A., Ueno, D., Clapiński, M., Vagin, A., Bruno, R., 2024. Towards efficient end-to-end encryption for container checkpointing systems, in: Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems, Association for Computing Machinery, New York, NY, USA. p. 60–66. URL: https://doi.org/10.1145/3678015.3680477, doi:10.1145/3678015.3680477.

Uzcategui, M., Hinrichs, T., 2019. Kubernetes policy enforcement using OPA at goldman sachs, in: Proceedings of KubeCon + CloudNativeCon North America. URL: https://confs.space/conf/kubecon-cloudnativecon-north-america/kubernetes-policy-enforcement-using-opa-at-goldman-sachs/. conference presentation.

Volatility Foundation, 2024. Volatility 3: The volatile memory extraction framework. https://github.com/volatilityfoundation/volatility3. Accessed: 2026-01-24.

Vömel, S., Freiling, F.C., 2011. A survey of main memory acquisition and analysis techniques for the windows operating system. Digital Investigation 8, 3–22. doi:10.1016/j.diin.2011.06.002.

Wallace, A., Baer, C., 2019. Exploring container security: Performing forensics on your gke environment. https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-performing-forensics-on-containers. Accessed: 2026-01-24.

Wang, E., Zurowski, S., Duffy, O., Thomas, T., Baggili, I., 2022. Juicing V8: A primary account for the memory forensics of the V8 JavaScript engine. Forensic Science International: Digital Investigation 42, 301400. URL: https://www.sciencedirect.com/science/article/pii/S2666281722000816. dFRWS USA 2022.

Weaversoft.io, 2024. Snap container checkpointing and state management platform. https://weaversoftio.github.io/Snap/.

Wong, A.Y., Chekole, E.G., Ochoa, M., Zhou, J., 2023. On the security of containers: Threat modeling, attack analysis, and mitigation strategies. Computers & Security 128, 103140.