CLARION: Sound and Clear Provenance Tracking for Microservice Deployments

Xutong Chen
Northwestern University

Hassaan Irshad
SRI International

Yan Chen
Northwestern University

Ashish Gehani
SRI International

Vinod Yegneswaran
SRI International

Abstract

Linux container-based microservices have emerged as an attractive alternative to virtualization as they reduce application footprints and facilitate more efficient resource utilization. Their popularity has also led to increased scrutiny of the underlying security properties and attack surface of container technology. Provenance-based analysis techniques have been proposed as an effective means toward comprehensive and high-assurance security control as they provide fine-grained mechanisms to track data flows across the system and detect unwanted or unexpected changes to data objects. However, existing provenance tracking techniques are limited in their ability to build sound and clear provenance in container network environments due to complexities introduced by *namespace virtualization*.

We describe a namespace- and container-aware provenance tracking solution, called CLARION, that addresses the unique *soundness* and *clarity* challenges introduced by traditional provenance tracking solutions. Specifically, we first describe *fragmentation* and *ambiguities* introduced in provenance analysis tools by each of the Linux namespaces and propose solutions to address analysis soundness. Then we discuss the design of specialized semantics-summarization techniques that improve the clarity of provenance analysis. We have developed a prototype implementation of CLARION and evaluate its performance against a spectrum of container-specific attacks. The results demonstrate the utility of our system and how it outperforms the state-of-the-art provenance tracking systems by providing an accurate and concise view of data provenance in container environments.

1 Introduction

Linux container technology has seen a rapid rise in adoption due to the miniaturized application footprints and improved resource utilization that are crucial in contemporary microservice architectures [20] and serverless computing environments [2]. The performance boost realized in containerized environments stems from their use of light-weight virtualization techniques whereby a single Linux operating system (OS) kernel is used to manage an array of virtualized containers. However, a side effect of this design choice is that an attack initiated inside a container may affect the shared host Linux OS kernel. Compared to the traditional virtual machine (VM) model, in which the guest VM OS is completely isolated from the host, this provides a much greater target surface to the attacker. Hence, comprehensive security tracking and analysis are vital in container networks.

The application of data provenance analysis techniques [22, 24, 27–30, 32, 34, 36, 39] for host and enterprise security monitoring has been well studied. However, extending such capabilities to container-based microservice environments raises some unique research challenges. At a cursory glance, the shared-host OS kernel substrate provides a centralized monitoring platform for observing events across containers and implementing security policy. In fact, the use of Linux namespaces introduces *fragmentation* and *ambiguities* in data streams used by provenance tracking systems, such as those based on the Linux Audit subsystem. Here, fragmentation refers to abnormal vertex splitting leading to false disconnections in the provenance graph. Conversely, ambiguity refers to vertex merging where a single vertex incorrectly represents multiple distinct objects in the correct provenance graph. Both fragmentation and ambiguities lead to false or missing dependencies. We refer to these as *soundness* challenges for container provenance analysis.

Namespaces [15] are a fundamental feature in the Linux kernel that facilitate efficient partitioning of kernel resources across process groups. This is the key feature exploited by popular containerization technologies such as Docker [7]. While processes within the same namespaces will share OS resources, those in different namespaces have isolated instances of corresponding operating system resources. For example, files in the same mount namespace have the same root directory so they must have different path names. Conversely, two files in different mount namespaces can appear to have exactly the same path names within but can still be distinguished by the root directory of their respective mount namespaces – i.e., their path names are virtualized (*containerized)* by the mount namespace. Unfortunately, it is the virtualized path names that will be recorded and reported by the kernel's audit subsystems, making those two files indistinguishable, which leads to falsely conflated elements in inferred provenance graphs.

Furthermore, mishandling the effect of namespaces can prevent a provenance tracking system from correctly characterizing essential aspects, such as the boundary of containers. Here the boundary of containers refer to the delineation of a provenance subgraph that represents the behavior within a container. It includes the processes running inside the container, the files manipulated by them, the sockets they create, etc. Without a proper understanding of container semantics

(i.e., ability to define boundary of containers and activity patterns of container engines corresponding to initialization, termination etc.), it will be impossible for security analysts to reason about *how*, *when*, and *what* containers are affected by attacks. We refer to these as *clarity* challenges for container provenance analysis.

CLARION Solution. To resolve the aforementioned soundness and clarity challenges, we propose CLARION, a namespace- and container-aware provenance tracking solution for Linux microservice environments. For soundness, we first provide an in-depth analysis of how the virtualization provided by each relevant namespace causes fragmentation and ambiguity in the inferred provenance. For each relevant namespace, we then propose a corresponding technical solution to resolve both issues. To improve clarity, we first define essential container-specific semantics including boundary of containers and initialization of containers. Next, we propose summarization techniques for each semantics to automatically mark the corresponding provenance subgraphs.

We show that soundness and clarity challenges are not specific – i.e., they exist in a range of monitoring approaches, including Linux Audit [25], Sysdig [21] and LTTng [16]. We describe a prototype implementation based on SPADE [23], an open source state-of-the-art provenance tracking system and comprehensively evaluate the effectiveness, efficiency, and generality of our solution. We studied the effectiveness and utility of our system using container-specific attacks. We also empirically evaluated system efficiency by running our solution on desktop computers as well as in an enterprise-level microservice environment. To assess generality, we collected provenance graphs for various state-of-the-art container engines including Docker, rkt [3], LXC [17] and Mesos [1]. These results show our solution works across container engines and outperforms the traditional provenance tracking technique by producing superior provenance graphs with an acceptable increase in system overhead ($< 5\%$).

Contributions. In summary, our paper makes the following contributions:

- We thoroughly analyze the ways namespace virtualization can affect provenance tracking. To the best of our knowledge, this is the first in-depth analysis of the implications of namespaces on microservice provenance tracking.
- Based on these insights, we designed and implemented a namespace- and container-aware provenance tracking solution – i.e., CLARION– that holistically addresses the soundness and clarity challenges.
- We conducted a comprehensive evaluation of the effectiveness, efficiency, and generality of our solution. The results show our solution produces sound and clear provenance in container-based microservice environments with low system overhead.

Table 1: Supported Linux Namespaces

Namespace	Isolated System Resource
Cgroup	Cgroup root directory
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
Time	Boot and monotonic clocks
User	User and group IDs
UTS	Hostname and NIS domain name

2 Background and Motivation

We provide basic background information on Linux containers and namespaces. We then use a motivating example to highlight the limitations of existing provenance tracking techniques and also describe our threat model.

2.1 Linux Namespaces

Linux namespaces [15] provide a foundational mechanism leveraged by containerization technologies to enable system-level virtualization. They are advertised as a Linux kernel feature that supports isolating instances of critical operating system resources including process identifiers, filesystem, and network stack across groups of processes. Internally, namespaces are implemented as an attribute of each process, such that only those processes with the same namespaces attribute value can access corresponding instances of containerized system resources. Currently, eight namespaces are supported by the Linux kernel as listed in Table 1.

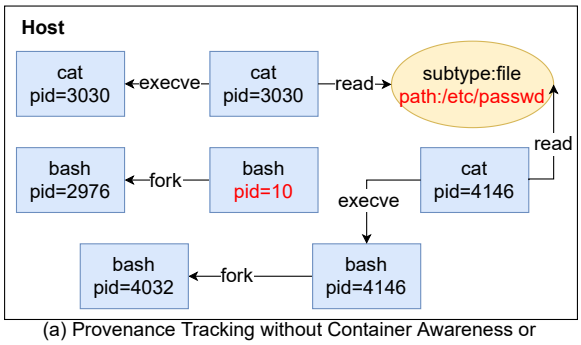
Consider the mount namespace as an example. On a Linux operating system that has just been booted, every process runs in an initial mount namespace, accesses the same set of mount points, and has the same view of the filesystem. Once a new mount namespace is created, the processes inside the new mount namespace can mount and alter the filesystems on its mount points without affecting the filesystem in other mount namespaces.

2.2 Linux Containers

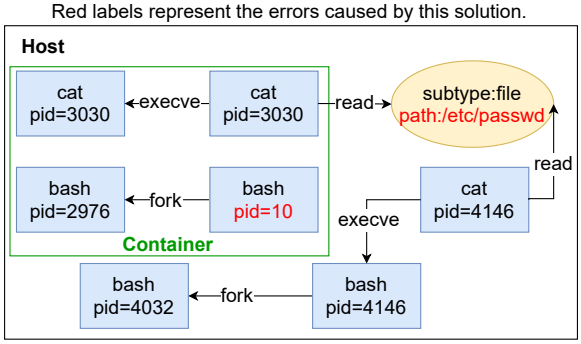
Linux containers may be viewed as a set of running processes that collectively share common namespaces and system setup. In practice, containers are usually created by a container engine using its container runtime. The container runtime will specify the namespace to be shared among processes running inside the container. As a concrete example, the Docker container engine specifies five namespaces (PID, Mount, Network, IPC and UTS) to be shared, initializes several system components including rootfs `/`, hostname, `/proc` pseudo-filesystem, and finally executes the target application as the first process inside the container.

2.3 Motivating Example

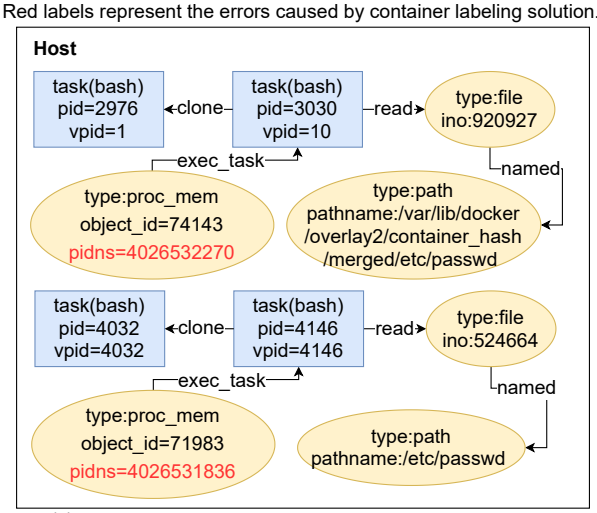
We motivate our solution by investigating the performance of three classes of state-of-the-art provenance tracking solutions



(a) Provenance Tracking without Container Awareness or Namespace Awareness
Red labels represent the errors caused by this solution.



(b) Provenance Tracking with only Container Awareness
Red labels represent the errors caused by container labeling solution.



(c) Provenance Tracking with only Namespace Awareness
Red labels represent the information which could be used for identifying containers.

Figure 1: Comparison between different provenance tracking solutions. The traditional provenance solution graph illustrated in (a) lacks namespace awareness and container awareness. The container-aware graph shown in (b), produced by systems such as Winnower, performs slightly better because it can distinguish processes from different containers, but lacks namespace awareness, leading to disconnected intra-container provenance graphs. The namespace-aware graph, illustrated in (c), produced by CamFlow lacks container-awareness. While this graph does not suffer from the soundness issue, it cannot effectively capture essential container semantics (e.g., the boundary of containers).

against a trivial credential theft insider attack[1]. Notably, during this attack, the attacker touches the `/etc/passwd` file in both a container and the host system.

First, as shown in Figure 1(a), traditional solutions that lack both container and namespace awareness, e.g., SPADE, are unable to deliver a sound and clear illustration of this attack step. To illustrate soundness challenges, we explain how *fragmentation* and *ambiguity* occur in the figure. For fragmentation, when `bash (2976)` forks a child process `bash (10)` with the global PID 3030 to execute the `cat` command, the virtualized PID 10 will be reported and used in building this process creation provenance so `bash (3030)` splits into two vertices, `bash (10)` and `cat (3030)`, which build incorrect `fork` and `execve` edges correspondingly. For ambiguity, consider the file `/etc/passwd`. Since the file path is virtualized, *ambiguity* occurs on the vertex representing two `/etc/passwd` files (inside and outside the container respectively) simultaneously. The correct graph should contain two separate `/etc/passwd` file artifact vertices. With respect to clarity, it is not intuitive which processes are inside the container because *container boundaries* are not marked in the graph.

Second, solutions that only provide container awareness, e.g., Winnower, also suffer from the soundness challenge. Though they can distinguish the processes inside the container in Figure 1(b), the ambiguity and graph fragmentation issues persist. This is also the case for other simple container labeling solutions, e.g., using a cgroup prefix or a SELinux label for every provenance artifact.

Third, solutions that only provide namespace awareness, e.g., CamFlow, still suffer from the clarity challenge. As we can see in Figure 1(c), they do not capture essential container semantics, such as the boundary of containers, complicating security analysis. As CamFlow provides a more fine-grained and complex provenance graph[2], non-trivial additional graph analysis will be required to design and apply similar semantic patterns in CamFlow to provide clarity. For instance, to support the boundary of containers, it is necessary in CamFlow to (1) put the PID namespace identifier on every task vertex to group processes inside a container by aggregating PID namespace information; (2) get the namespace-virtualized pathname and the mount namespace identifier for each file to reveal whether the file is inside a container by complementing mount namespace information.

For (1), we need to find the process memory vertex assigned to each task vertex and use its PID namespace identifier. Figure 1(c) illustrates a simple case. In practice, the graph analysis required is more complex. Because CamFlow uses versions to avoid cycles or to record any object state change for a provenance artifact, a path traversal is needed to find the correct version of the task vertex, i.e., where a clone tries to

[1] A complete attack description can be found in the Appendix, but is not required to illustrate the challenges faced by container provenance systems.

[2] The provenance graph of CamFlow is framed over fine-grained kernel objects, e.g., task, process memory, inode, path, packet.

```
type=SYSCALL msg=audit(1567029444.851:431219): arch=c000003e syscall=56
success=yes exit=2 a0=3d0f00 a1=7f81aa8f8fb0 a2=7f81aa8f99d0
a3=7f81aa8f99d0 items=0 ppid=5880 pid=5903 auid=4294967295 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295
comm="runc:[2:INIT]" exe="/" key=(null)
```

Figure 2: Problematic Linux Audit Record (PID)

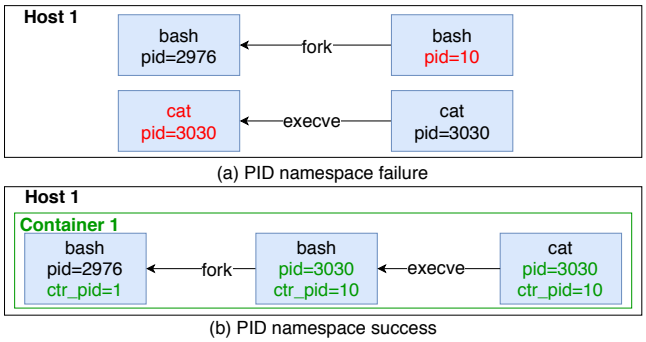


Figure 3: PID Namespace: Failure and Success

assign the process memory. For (2), CamFlow does not provide virtualized paths and mount namespace identifier for file vertices natively. The same state management implemented on CLARION (See Section 4.1.2) to track `pivot_root` and `chroot` calls and path traversal analogous to what was described above for (1) will need to be implemented within CamFlow.

2.4 Threat Model

We consider the OS kernel and audit subsystem, i.e., Linux Audit, to be part of the trust computing base (TCB). We assume that the OS kernel is well protected by existing kernel-protection techniques [13, 38]. The integrity of Linux Audit assures the ability to observe all system calls associated with malicious activity in user space. If the attackers succeed in compromising the kernel, they can disrupt the normal operation of Linux Audit and the kernel module used by CLARION. To address such attacks, the security of the TCB can be bolstered using TPM-based approaches as used by prior provenance-tracking systems [22, 35].

We further assume adversaries can only have limited or no *a priori* privileges. Thus, we only consider a threat model where attacks are launched from user space. This threat model is based on what was used in prior provenance tracking systems, and it is reasonable because the container virtualization does not mitigate the effort required for attackers to compromise the kernel. Implementing provenance tracking for containers and addressing namespace virtualization problems shown in Section 3 do not require additional information beyond what is provided in the kernel, as described in Section 4. Finally, the system may be subject to resource exhaustion attacks, leading to missed events. We believe that the defense against such attacks is outside the scope of this paper.

```
type=SYSCALL msg=audit(1573775822.523:18757): arch=c000003e syscall=257
success=yes exit=3 a0=ffffff9c a1=7fff09576970 a2=0 a3=0 items=1 ppid=22352
pid=22422 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=pts0 ses=4294967295 comm="cat" exe="/bin/cat" key=(null)
type=CWD msg=audit(1573775822.523:18757): cwd="/"
type=PATH msg=audit(1573775822.523:18757): item=0 name="/etc/passwd"
inode=67535 dev=00:2e mode=0100644 ouid=0 ogid=0 rdev=00:00
nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
cap_fe=0 cap_fver=0
```

Figure 4: Problematic Linux Audit Record (Mount)

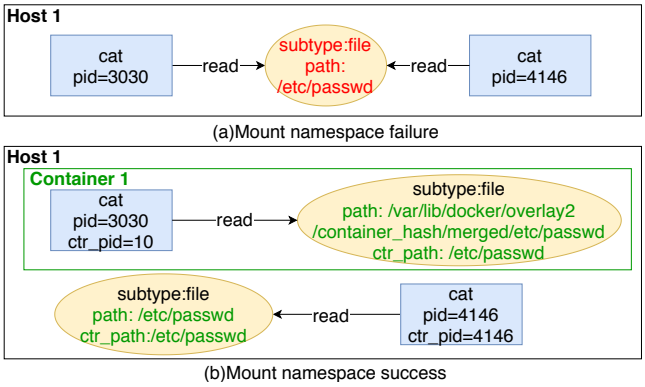


Figure 5: Mount Namespace: Failure and Success

3 Container Provenance Challenges

We elaborate on the soundness and clarity challenges introduced by mishandling the effect of container virtualization in each namespace. Through the analysis in this section, we also ensure that the technique we propose in Section 4 covers all the needed namespace interactions.

3.1 Soundness: Namespace Virtualization

As illustrated in the motivating example, *fragmentation* and *ambiguity* are soundness issues caused by namespace virtualization in provenance tracking. However, not every namespace triggers either or both issues. In Table 2, we provide a deeper analysis about *how* each namespace impacts provenance tracking and *what* events will be affected. In addition, we use audit records from Linux Audit to demonstrate the problem and show how the soundness challenge can extend to other monitoring techniques such as Sysdig and LTTng.

3.1.1 PID Namespace

Figure 2 shows a problematic audit record. It is created by a `runC` container runtime process inside a container and trying to finish the initialization. Syscall value 56 means that it is a `clone` system call, and its return value is the PID of the cloned child process. Here, we can see that exit value is 2, but the process 2 is usually a kernel-related process generated when the system is booted. It suggests process 2 cannot be the cloned child process of this `runC` runtime process, which is confirmed by our further investigation. So 2 cannot be the global PID for the cloned child process. It can only be a virtualized PID.

Figure 3 illustrates the subgraph exposing the fragmentation caused by PID namespace virtualization in the motivating

Table 2: Namespace Virtualization: What / How Provenance?

Namespace	What events will be affected?	How each namespace impacts provenance tracking?
PID	Audit records related to process manipulation system calls (e.g., `clone`, `fork`) will be affected. In those records, the argument fields and return value field with PID semantics are virtualized but the PID fields themselves are virtualized. This leads to a semantic inconsistency.	The aforementioned inconsistency leads to fragmentation in the provenance graph when process creation happens, so the provenance tracking system fails to produce sound provenance information.
Mount	Audit records related to file operation system calls, e.g., `open`, `close` and `read`, will be affected. Just like the PID namespace, argument fields with file path semantics will be virtualized. In addition, the file path in CWD and PATH records will also be virtualized.	Two different files, accessed within two different containers, may have the same name which leads to ambiguity. Thus the provenance tracking system fails to produce sound provenance information.
Network	Audit records containing local IP addresses and local ports of a socket will be affected. Examples include the `bind` system call, which is the only system call directly providing local IP and local ports of a socket in its arguments, and other system calls like the `listen` system call providing socket file descriptors where local IP addresses and ports can be indirectly extracted.	Two sockets in two containers can have the same local IP address and local port leading to ambiguity. Furthermore, sockets inside the container are connected to a host port through port-mapping rules. Without explicit understanding of this mapping information, the provenance system fails to connect the incoming connection to the correct sockets, leading to fragmentation.
IPC	Audit records related to system calls handling SYSV IPC objects, i.e., message queue, semaphore and shared memory segmentation, and the POSIX message queue will be affected, e.g., `msgget`, `mq_open`, `shm_open`. The effect is that argument fields with the semantics of the ID/name of a SYSV IPC object or a POSIX message queue are virtualized.	Two IPC objects of the same type can have the same ID/name, and this will lead to ambiguity in the provenance graph.
User	The only affected data elements are UIDs and GIDs. They do not lead to fragmentation in the provenance graph. As for ambiguity, Linux Audit records can report the host view of UID and GID in the corresponding fields of every audit record so that ambiguity will also be resolved.	Since there is no impact, user namespace auditing is unchanged. Furthermore, most container engines do not use the user namespace in their default container initialization because it breaks access permission to critical libraries on the host and storage features like bind mount may be automatically disabled if the user namespace is enabled in the container.
Time, UTS and Cgroup	These namespaces do not affect dataflow in practice and thus do not directly impact provenance.	N/A

example. The `bash` process 2976 expects that it created child process 10 which is actually process 3030.

3.1.2 Mount Namespace

Figure 4 shows a problematic audit record. A system call `openat` (inferred by syscall=257) is invoked by a process trying to read `/etc/passwd` in the container. As we can see, the CWD is `/` and the PATH is `/etc/passwd`. In fact, all files inside the container are stored under some directory specific to this container. This specific directory may vary due to different container engine choices. Taking Docker as an example, the specific directory is usually `/var/lib/docker/overlay2/container_hash/merged/` where `container_hash` is a hash string related to this container. So to get the global paths of the CWD `/` and the PATH `/etc/passwd`, the path of the specific directory needs to be added to them as the prefix.

Figure 5 illustrates the subgraph exposing the mount namespace virtualization problem described in the motivation example. Two `cat` processes (with PIDs 3030 and 4146), are attempting to read the `/etc/passwd` file, and the two files are confused with each other without mount-namespace awareness. CLARION's host-container mapping enables us to easily distinguish between them.

3.1.3 Network Namespace

Figure 6 illustrates the subgraph exposing the network namespace virtualization problem in the motivation example. Two `nc` processes (PID 3043 and 4149) are listening on socket (0.0.0.0/4000) within their respective containers, and one of them accepts a connection from (10.0.2.15/3884). Since the local IP addresses/ports are virtualized and remote IP addresses/ports are the same, the two sockets can be confused with each other without network-namespace awareness. Without establishing the host-container mapping of sockets inside the container, we are unable to attribute the connection to a socket inside the container, as illustrated in Figure 6.

3.1.4 Soundness Challenge on Other Audit Subsystems

We further investigated the impact of container virtualization on two alternative Linux audit subsystems, specifically Sysdig [21] and LTTng [16], to assess whether soundness challenge impacts other systems besides Linux Audit. We summarize our findings in Table 3. We find that Sysdig suffers from the same soundness challenges confronted by Linux Audit. LTTng provides host-container ID mappings using more low-level events[3] but the soundness challenge in mount namespace still persists. Our investigation shows that soundness challenge is not specific to Linux Audit.

3.1.5 Soundness Challenge on Rootless Containers

Rootless containers refer to the containers that can be created, run, and managed by unprivileged users. They differ from traditional containers in which they have a new unprivi-

[3]For example, the `sched_process_fork` event.

Table 3: Provenance Soundness on Sysdig and LTTng

Namespace	Sysdig	LTTng
PID	Soundness challenge persists because the return values and the arguments providing PID semantics will be virtualized in the audit records corresponding to process manipulation system calls, e.g., `clone`, `fork`.	Soundness challenge persists if only system call events are used in provenance tracking system because the return values and the arguments providing PID semantics will be virtualized. However, LTTng can provide the host-container PID mapping which eliminates the PID namespace soundness challenge.
Mount	Soundness challenge persists because the data fields providing file path semantics, e.g., `name` and `filename`, will be virtualized in the audit records corresponding to file operation system calls, e.g., `open`, `close`, and `read`.	Soundness challenge persists because the data fields providing file path semantics, e.g., `filename`, will be virtualized.
Network	Soundness challenge persists. The data fields having local IP addresses/ports will be virtualized. Examples include the argument (`addr`) of a `bind` system call and the translation of the argument (`fd`) being the socket file descriptor of a `listen` system call.	Local IP addresses and ports are still affected. However, since LTTng does not explicitly transform the `addr` argument in the `bind` system call to a socket address, the soundness challenge in network namespace is less severe.
IPC	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue will be virtuailzed.	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue virtualized by IPC namespace will be virtuailzed.
User	The return values and arguments of UID-manipulation system calls will be virtualized but soundness is not affected.	Soundness is not affected. Furthermore, clarity can be achieved since the UID/GID host-container mapping is provided.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

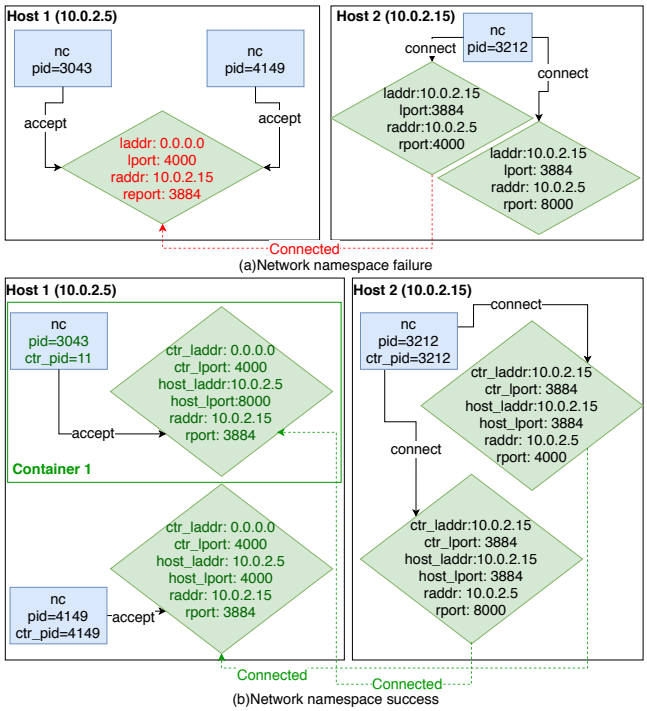


Figure 6: Network Namespace: Failure and Success

leged user namespace. In this user namespace, all UIDs and GIDs are mapped from the global user namespace, including a pseudo-root user. This core difference causes further effects in filesystem and networking in the rootless container. For filesystem, many Linux distributions do not allow mounting overlay filesystems in user namespaces. This limitation makes rootless containers inefficient. For networking, virtual Ethernet (veth) devices[4] cannot be created as only real root users have the privileges to do so.

As summarized in Table 2, the user namespace does not affect the soundness of provenance analysis. Further, although rootless containers have slightly different implementations for filesystem and networking (mentioned above), to support unprivileged root users, they do not affect provenance. Thus, we claim that rootless containers share the same soundness challenges faced by traditional containers.

3.1.6 Soundness Challenge on Other OS Platforms

We also investigated two alternative resource isolation techniques, specifically FreeBSD Jails and Solaris Zones, to see whether soundness challenge can also occur in other platforms. We summarize our findings in Table 4. Our key finding is that if semantics inconsistency exists in the low-level audit events related to virtualized system resources (e.g., PIDs, file paths, network addresses/ports), the resource-isolation technology will suffer from the soundness challenge. We assume this finding also extends to other OS platforms like Windows and MacOS. Semantic inconsistencies are at the core of the soundness challenge so the key to make CLARION feasible on those platforms is to systematically address such inconsistencies.

3.2 Clarity: Essential Container Semantics

We describe the challenges involved in automating the comprehension of essential container semantics. This is a feature that is unique to our provenance tracking system, and we believe it can greatly simplify the understanding and analysis of dataflow provenance in container-based microservice environments.

An important aspect of forensic analysis is accurately knowing what subgraphs correspond to which container so that we

[4]Veth devices are a special type of Linux interface used in virtual networking. They are always created in pairs and usually serve as local Ethernet tunnel between namespaces in container networks.

Table 4: Provenance Soundness in BSD Jails and Solaris Zones

Resource	BSD Jail	Solaris Zone
Process	BSD Jails use JID (Jail ID) to mark the processes inside a jail. Thus no virtualized PID is used and no soundness challenge will be introduced.	Zone ID is used to isolate the processes. Thus no virtualized PID is used and no soundness challenge will be introduced.
Filesystem	Ambiguity exists because filesystem isolation is also achieved by chroot-like operation and file path will be virtualized while the root directory path is specified by `jail` system call.	Ambiguity exists because a Zone needs a new root directory to be specified.
Network	This depends on what network isolation method is applied. If bind-filtering is applied, sockets are actually created under host network stack so that no soundness challenge would occur. Otherwise, if *epair* of VNET is used for network isolation, each jail would have a completely separate network stack just like what happens in Linux network namespace. Then both fragmentation and ambiguity can exist.	Both fragmentation and ambiguity can exist. When the default exclusive-IP setting is applied, Data-link acts just like veth pairs in Ubuntu and epair in BSD to provide the isolated network stack where sockets are virtualized.
IPC	Ambiguity exists. POSIX IPC objects are naturally isolated and System V IPC objects can be isolated with specific parameters so two IPC objects can have the same ID/name.	Ambiguity exists. System V IPC objects are naturally isolated and two System V IPC objects can have the same ID/name.
User	The same provenance effect as that in Table 2 will occur for jails.	The same provenance effect as that in Table 2 will occur for zones.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

can effectively track the origins of a container microservice attack as well as assess the forensic impact of such attacks. For example, was the effect of the attack limited to the specific container or was it used as a stepping stone to other container targets? To effectively answer such questions, we need to demystify the boundary of containers in the provenance graph.

Initialization of containers is another frequent activity that explodes provenance graphs and may be abstracted to simplify analysis. Thus, if we can accurately identify subgraphs corresponding to initialization of each containers, we can produce simplified provenance graphs, effectively reducing the effort for forensic analysts by automatically annotating abnormal cross-container activity. Specifically, we investigate the container initialization regulation of several representative container engines, including Docker, rkt and LXC, and summarize the patterns observed in each of them.

4 System Design and Implementation

In this section, we provide a detailed description of the CLARION prototype design and the implementation that extends the SPADE provenance tracking system with additional container-specific extensions. Our design goal is to propose a solution for addressing *soundness* and *clarity* challenges by only using trusted information from the kernel while limiting extra instrumentation.

We claim that our solution is complete in handling all aliasing introduced by namespaces. First, we cover all system calls that can be used to manipulate namespaces generally, i.e., `clone`, `unshare` and `setns`. We investigate their semantics and provide associated provenance data models with consideration to different argument combinations as shown in Section 4.2. Second, we analyze all existing namespaces and understand what information will be aliased in the low-level audit and cause problems to provenance tracking as shown in Section 3.1. Our solution is designed to address all introduced problems below in Section 4.1.

Table 5: Namespace Provenance Mapping Strategies

Namespace	Affected Provenance Data	Strategy
PID	Process IDs	Host-container mapping
Mount	File paths	Host-container mapping
Network	Local IP addresses and ports	Host-container mapping
IPC	IPC Obejct IDs and names	Namespace labeling

4.1 Mapping Virtualized Namespaces

We summarize our virtualized namespace-mapping strategies in Table 5. For the soundness challenge, we establish a *host-container mapping* view on provenance graph artifacts that are impacted by most Linux namespaces because we believe this will provide the most clear view for users to understand the provenance. However, for the IPC namespace, the host view of an IPC object does not actually exist. Hence, we adopt a *namespace-labeling* approach.

4.1.1 PID Namespace

We considered multiple options to tackle the PID host-container mapping problem including: (*i*) directly using PPID (parent PID) to connect processes; (*ii*) using timestamps to map cloned child processes to its parent; (*iii*) using `/proc/PID/status` for mapping information; and (*iv*) using kernel module injection to get the PID mapping from kernel data structures.

We ultimately eliminated other options and chose to implement a kernel module for several reasons. We found that directly using PPID was infeasible because it sometimes points to the parent of the the process creating it. For the timestamp option, the granularity provided by audit record cannot guarantee that the order of process creation matches the order corresponding system call events. We also decided against using `/proc/PID/status` information as the `/proc` filesystem does not support asynchronous callbacks and the overhead of polling is prohibitive.

We implement our PID namespace host-container mapping solution as a kernel module that intercepts process-manipulation-related system calls, e.g., `clone`, `fork`, and

Table 6: Operator Annotation

Annotation	Explanation
put((key,value), X)	put a pair (key,value) in a mapping X
get(key, X)	get the value from a mapping Y given the key

vfork. Once those system calls are invoked by a process, we do not directly use the return value to determine the PID of its child process because it can be virtualized. Instead, we input this return value to a kernel helper function pid_nr() in /include/linux/pid.h to generate the global PID. Ultimately, we use the global PID to generate the sound provenance graph. However, we still capture both the global PID and virtualized PID for every process vertex such that a complete view can be provided.

4.1.2 Mount Namespace

To obtain the host-container mapping for file paths virtualized by containers, we leverage an empirically derived design principle about the mount namespace, that is consistent across state-of-the-art container engines, to develop an instrumentation-free solution.

This empirical design principle is that the newly created mount namespace needs the init process, i.e., the process with virtual PID 1, to provide a new filesystem view different from that in the parent mount namespace. It is achieved by using root directory change system calls, i.e., pivot_root and chroot, where new root directories are provided in their arguments. Specifically, state-of-the-art container engines make the init process move CWD to the root directory of a new container by using chdir(container_root_path) and then invoke a pivot_root('.','.') or a chroot('.') to wrap up the root directory change.

Therefore, if we monitor those root directory change system calls, we can use the CWD record associated with the chdir to find the host path of the container root directory, and then we attach this host path to every virtualized path as a prefix to establish the host-container mapping on file paths. Given the annotation in Table 6, the algorithm is described as four steps.

Step 1. Handle chdir. (input: PID 'p1', CWD 'cwd1'; operation: put((p1,cwd1), LastCWD)). We do this to record the last working directory for every process. With this information we can know what is the last CWD of the first process inside a new container, which will be the prefix for every virtualized path.

Step 2. Handle pivot_root or chroot. (input: PID 'p1'; operation: put((p1, get(p1, LastCWD)), Prefix)). When a root directory changing system call occurs, we label the corresponding process with the last CWD as the prefix.

Step 3. Handle virtualized PATH records, CWD records and arguments related to file operation system calls with path prefix. (input: PID 'p1', syscall 's1', operation: if 's1' is 'open','read','write' etc. Use get(p1, Prefix) to add a new annotation 'nsroot' representing the host prefix in the corresponding artifacts). This helps propagate the prefix from processes to file artifacts.

Step 4. Handle (clone, fork, vfork). (input: Parent PID 'p1', Child PID 'p2'; operation: put((p2, get(p1, Prefix)), Prefix)). The prefix (root directory) information will be propagated through process creation as kernel does.

We consider our mount namespace mapping solution to be robust because it relies on a standardized implementation technique for filesystem isolation and empirically validate its adoption across representative container engines including Docker, rkt and LXC.

For other cases where directories are shared between host and container than chroot-like cases, we claim that our solution still works well. Taking bind mount as an example, the key components in the bind mount provenance graph will be one process vertex which executes a mount system call along with two file artifacts representing the bound directories and two file artifacts are connected by an edge representing that mount system call. In this case, only the file path of the file artifact inside the container will be affected and our solution can still provide the host view of this file.

4.1.3 Network Namespace

For accurate provenance tracking of container network activity, CLARION needs to establish the host-container mapping for virtualized local IP addresses and ports. To this end, we design a Netfilter-based solution for tracking the host-container IP/port mapping and use the network namespace ID as a distinguisher. Netfilter is a Linux-kernel framework that provides hooks to monitor every ingress and egress packet, including packets from or to containers, on the host network stack [19]. The host network stack will do a source NAT for container egress packets and a destination NAT for container ingress packets before correctly forwarding those packets. Therefore, by monitoring the IP/port NAT about container ingress/egress packets on the host network stack, we can build the host-container mapping of local IP addresses and ports for sockets inside containers. We annotate each network socket artifact with the corresponding network namespace identifier, so sockets from different containers can be reliably distinguished.

The CLARION prototype implementation for the network namespace consists of two parts: network namespace identification and netfilter-based address mapping. For network namespace identification, we modify SPADE's kernel module to intercept network-related system calls and put the network namespace identifier of the calling process on the generated network socket. For netfilter-based mapping, we register kernel modules at the beginning and the end of netfilter hooks corresponding to NAT. Specifically, POST_ROUTING and LOCAL_INPUT are used for source NAT, while PRE_ROUTING and LOCAL_OUTPUT are used for destination NAT. The former two hooks provide the mapping for egress connections from container and the latter two

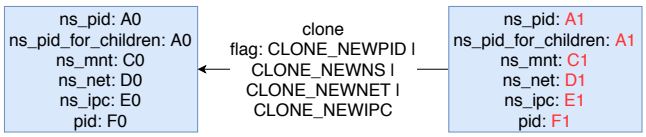


Figure 7: Handling the `clone` system call: a process vertex representing the child will be created with the new namespace label.

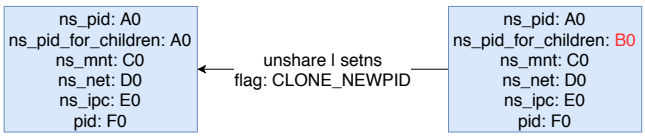


Figure 8: Handling the `unshare` and `setns` system calls on NEWPID: a process vertex representing the calling process itself will be created with the new assigned pid_for_children label.

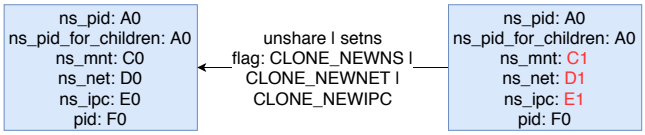


Figure 9: Handling `unshare` and `setns` system calls with other flags: a process vertex representing the calling process itself will be created with the new assigned namespace label.

provide the mapping for ingress connections.

Whenever a new mapping is added, we will search for the network device having the virtualized local IP address in the new mapping, by iterating through network namespaces using the function `ip_dev_find(struct net *net, __be32 addr)`. Through this, we find the container related to this virtualized local IP address and put the mapped global local IP address/port on the socket artifact that has the virtualized local IP address/port in the new mapping. As a special case, a socket may listen on `0.0.0.0` (IN_ADDR_ANY), i.e., it can accept connection on any local IP address. Hence, when we match socket artifacts with the virtualized local IP address/port in the container, we always treat `0.0.0.0` as a matched local IP and only check the local port.

4.1.4 IPC Namespace

The issue in the IPC namespace is that two different IPC objects from different IPC namespaces may have the same ID/name. Unlike other namespaces, the host-container mapping strategy for disambiguation does not extend to IPC object artifacts, because there is no corresponding host IPC object for virtualized IPC objects. Our design involves adding an IPC namespace ID to every IPC object artifact so that IPC objects from different containers can be uniquely distinguished.

The implementation of the IPC namespace solution was effected by adding IPC namespace IDs to IPC objects affected by namespace virtualization. Those objects consist of the POSIX message queue and all System V IPC objects, i.e., message queue, semaphore, and shared memory. We assign and propagate IPC namespace ids by carefully interpreting process management system calls, e.g., `clone`, and IPC object management system calls, e.g., `msgget` and `msgsnd`.

4.2 Essential Container Semantic Patterns

To address the clarity challenge, we propose two essential container semantics which can significantly improve the quality of provenance graph. In addition, we design the semantic patterns for summarizing them during provenance tracking.

4.2.1 Boundary of Containers

We begin by first providing a practical definition for a container at runtime. A container at runtime is a set of processes that share the same PID namespace. Usually processes inside a container can share multiple namespaces but, most critically, they at least have to share the PID namespace. In fact, while container runtimes often provide support for sharing other namespaces, e.g., mount, IPC, and network, between containers, none of them allow for sharing the PID namespace.

Next, we define the relationship between an artifact, e.g., file and network, and a container. An artifact relates to a container if and only if it can be accessed by a process inside that container. Here, "accessed" may refer to any type of read-write operation. An artifact may relate to several containers and thus may be used to infer the relationship between specific containers. An important challenge is labeling each process with the correct namespace identifier. We address this by carefully designing a new provenance data model for system calls related to namespace operations. There are three essential system calls for tracking the boundary of containers, i.e., `clone`, `unshare` and `setns`. `Clone` and `unshare` system calls are used for creating new namespaces; thus, they signal the process of creating a container boundary. `Setns` is used for aggregating two namespace together or making another process join a namespace.

We designed five different namespace labels (corresponding to PID, mount, network, IPC, and pid_for_children) and handle them when three essential namespace-related system calls (i.e., `clone`, `unshare`, and `setns`) occur, as shown in Figure 7, 8 and 9. All figures are illustrated in the OPM provenance data model format. The red areas highlight the changes between before and after. The implementation follows the Linux Kernel semantics for each system call and each namespace. The special case here is that if CLONE_NEWPID flag is specified for `unshare` or `setns` process, this only affects the child process generated by the calling process but does not affect the calling process itself. By adding namespace labels and handling namespace-related system calls, CLARION is able to capture the namespace information for every single process and leverage the PID namespace label to certify the boundary of each container.

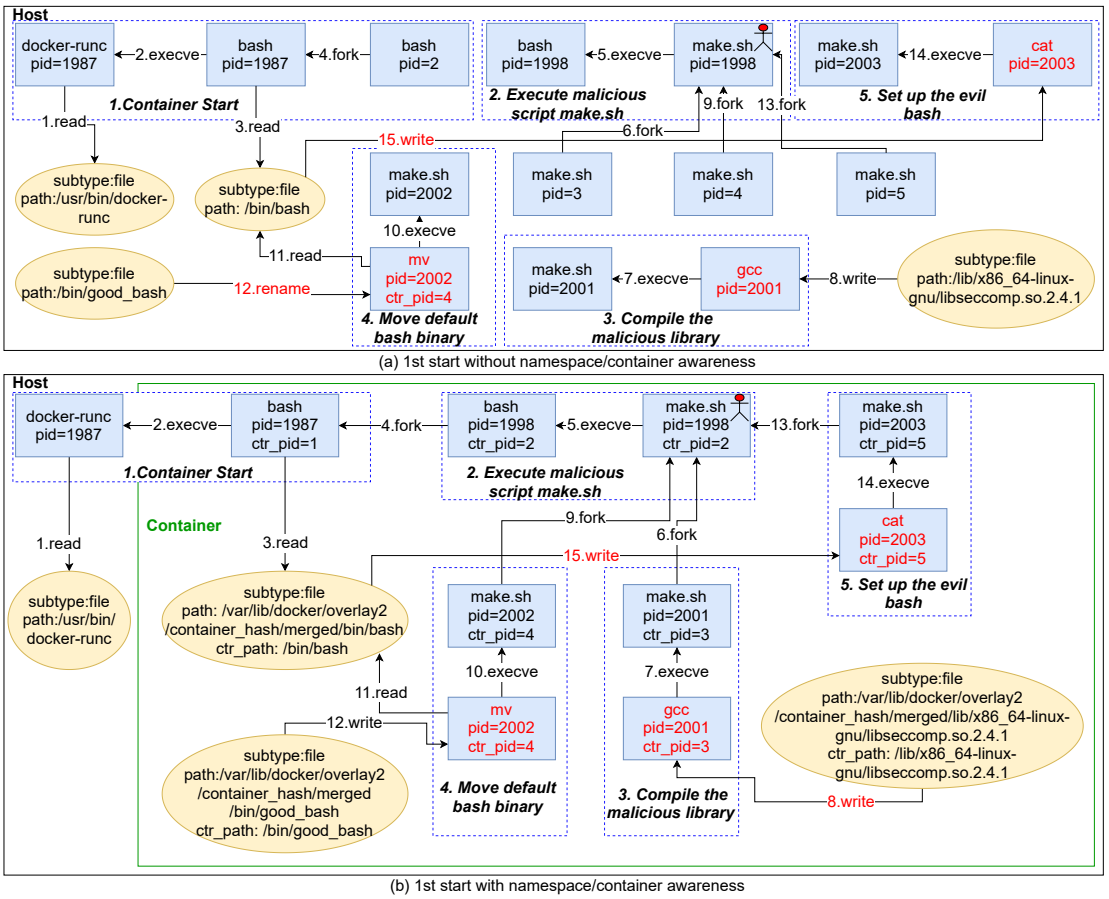


Figure 10: CVE 2019-5736: Provenance graph for 1st start without (top) and with (bottom) namespace/container awareness

4.2.2 Initialization of Containers

By analyzing several state-of-the-art container engines, we find that specific common pattern exist across containers that may be leveraged to identify the initialization of containers. In a nutshell, this pattern can be summarized as follows: start with an `unshare/clone` with new namespace flag specified, and end with an `execve` so that a new application can be launched inside the container. Slight differences exist across different container engines as described in Section 4. Identifying these patterns facilitates abstracting subgraphs in the provenance graph that corresponds to container spawning and initialization activity.

Here, we explain the container initialization patterns for Docker and rkt. For Docker, the initialization pattern is as follows:

- After receiving gRPC connection from `dockerd`, `containerd` will start a `containerd-shim`, which is responsible for starting a new container.
- This `containerd-shim` process will invoke several `runC` processes for initialization.
- One of those `runC` processes will invoke the `unshare` system call and this marks the beginning of the actual container initialization.
- The `runC` process calling `unshare` will clone several child processes to finish several initialization tasks including setting up `/proc`, `/rootfs`, and the network stack.
- Finally, it will clone a child process and make it execute the default container application, e.g., bash and apache.

Unlike centralized container engines like Docker, rkt does not have a daemon process that is responsible for starting a container. It has a more complex three-stage initialization pattern that begins once `rkt` is started with specified parameters to create a rkt container.

- **Stage 0:** It will use several instances of the `systemd` process to set up different namespaces including PID, Mount, Network, IPC and UTS.
- **Stage 1:** It will generate process inside the container with namespace restriction set up and call `chroot` to create a filesystem jail for this container.
- **Stage 2:** Finally, it will run the default application on this process.

We implement those patterns as a SPADE filter, and it automatically finds the starting point of those initialization patterns and attempts to do a backward traversal so the subgraph corresponding to initialization will be marked.

5 System Evaluation

In this section, we evaluate CLARION by answering the following questions.

- **Q1. Usefulness:** How *effective* is CLARION in dealing with real-world container microservice attacks?
- **Q2. Generality:** Are namespace disambiguation strategies implemented by CLARION for performing provenance tracking *generally* applicable across different container engines?
- **Q3. Performance:** Does CLARION provide an *efficient* provenance monitoring solution for real-world microservice deployments?

5.1 Effectiveness Evaluation

To illustrate the effectiveness of CLARION for container-based provenance and forensic analysis, we evaluate against exploits of three recent CVEs affecting Docker. Specifically, we generate the provenance graphs with and without namespace and container awareness to show the capability of our solution. Then, we compare between the original provenance graphs generated by SPADE and CLARION.

The CVEs that we selected include CVE 2019-5736 (`runC`), CVE 2019-14271 (`docker-tar`) and CVE 2018-15664 (`docker-cp`). The first two exploits are particularly detrimental because they can lead to privilege escalation in the host machine. The third is a race-condition (time-dependent) which requires multiple tries and some serendipity for successful exploitation.

5.1.1 CVE 2019-5736: runC Exploit

`runC` through 1.0-rc6 (as used in Docker before 18.09.2 and other platforms like LXC), allows attackers to overwrite the host `runC` binary (and consequently obtain host root access) by leveraging the ability to execute a command as root within one of these containers: (1) a new container with an attacker-controlled image, or (2) an existing container, to which the attacker previously had write access, that can be attached with docker exec. This occurs because of file-descriptor mishandling, related to `/proc/self/exe` [6]. Overwriting `runC` can lead to a privilege escalation attack by replacing `runC` binary with a backdoor program. The following four steps are used to demonstrate a successful exploitation using this vulnerability:

1. Create a container where we have gcc installed.
2. Create three files in this container with the `docker cp` command. Those files are (1) a script (`bad_init.sh`) that overwrites the executable (`/proc/self/exe`) of the process running this script; (2) a C program file (`bad_libseccomp.c`) that invokes `bad_init.sh`; and (3) a shell script (`make.sh`) for compiling `bad_libseccomp.c` and setting up the bait for `runC`.
3. Start this container and execute `make.sh` that accomplishes two goals: (1) replaces the `seccomp` module with `bad_libseccomp.c`. Here `seccomp` module is a regular library which will be automatically loaded when an Ubuntu container starts; (2) replaces the default start-up process, i.e., the bash shell in Ubuntu containers, with `/proc/self/exe`.
4. If and when this container is restarted, `runC` on the host loads the malicious `seccomp` module leading to execution of the malicious script (`bad_init.sh`), which overwrites the parent process, i.e., `runC` will be overwritten with the contents of `bad_init.sh`.

We illustrate the provenance graphs associated with this exploit in Figures 10, 11. This exploit consists of two starts of a malicious container. Figure 10 corresponds to the first start and Figure 11 corresponds to the second start.

We see that in the graphs without namespace awareness, the provenance graph fractures completely. Specifically, subgraphs corresponding to essential exploit steps are fractured, making it challenging for analysts to do backward and forward tracking given the attack points on `make.sh` and `bash`. Furthermore, ambiguity exists everywhere in those namespace-unaware graphs. Among many points exposing ambiguity, the ambiguity between two `/lib/x86_64-linux-gnu/libseccmp.so.2.4.1` file artifacts in the second start is significant. If we cannot distinguish between them, we will not be able to understand that a malicious GNU library inside the container is linked with the `runC` instance (`/proc/self/exe`).

CLARION can successfully restore the connection between essential exploit steps in the namespace-aware provenance graphs and also resolve the associated ambiguity issues, making it very clear that a malicious container library is linked by the `runC` instance (which is anomalous).

5.1.2 CVE 2019-14271: docker-tar Exploit

Docker 19.03.x (prior to 19.03.1) linked against the GNU C Library (`glibc`) is vulnerable to code injection attacks, that may occur when the nsswitch facility dynamically loads a library inside the container using `chroot` [5] [5]. This code injection can affect the library files on the host and may be used to trigger privilege-escalation attacks.

We exploit this privilege-escalation vulnerability using `docker-tar`, a helper process spawned by the Docker engine that obviates the need to manually resolve symlinks in the container filesystem. First, `docker-tar` uses the `chroot` command to change its root to the container's root. This is done so that all symlinks are resolved relative to the container's

[5] The assumption here is that libraries within the containers are untrusted from the perspective of the host.

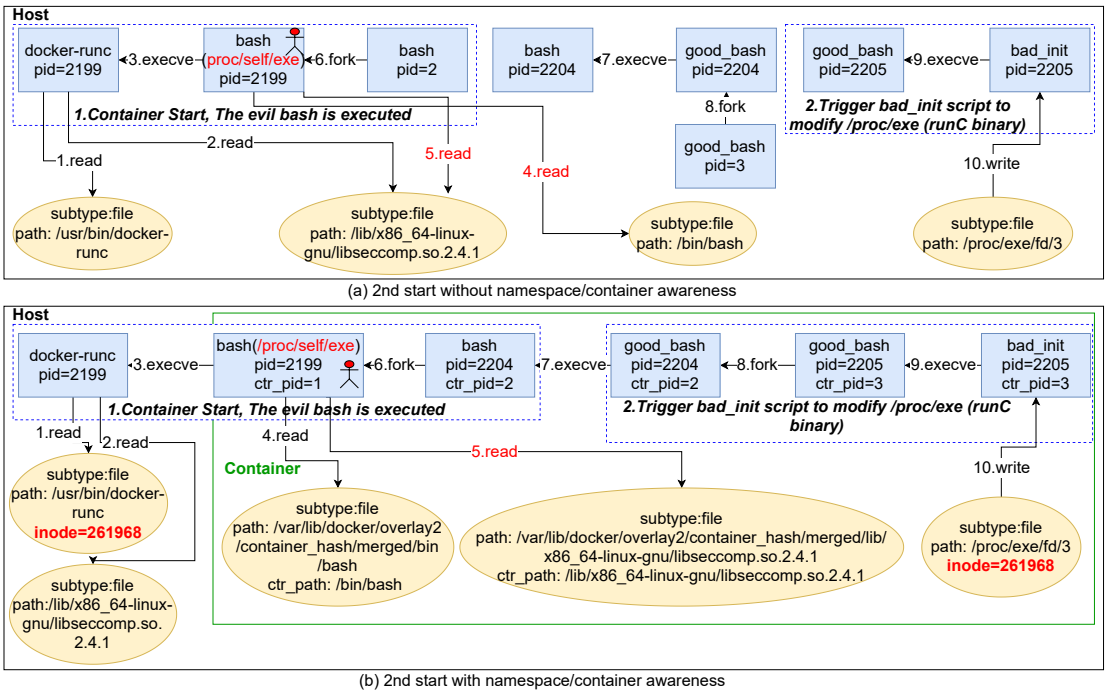


Figure 11: CVE 2019-5736: Provenance graph for 2nd start without (top) and with (bottom) namespace/container awareness

root, preventing any ambiguities. After running `chroot`, `docker-tar` attempts to link with several standard `glibc` libraries, which induces the vulnerability. When `docker-tar` attempts to link with these libraries, it will instead link the library files inside the container. However, a malicious image could inject code inside those library files, such that the malicious code executes as part of the `docker-tar` process since they are linked by `docker-tar`. Specifically, two steps are required to demonstrate exploitation of this vulnerability:

1. We modify `libnss_files.so` in the container image by a malicious script `modify.sh`, an example library linked with `docker-tar`, using a code injection attack such that it includes additional code to execute a malicious script, called `breakout.sh`, that sets up a backdoor on the host using `netcat`.
2. When we then run the `docker-tar` command from a container using this image, the `docker cp` command is executed within the container that copies the library file to the host, leading to an open backdoor on the host.

Provenance graphs for comparison are shown in Figure 12. Similar to the first exploit, the namespace-unaware provenance graph is fractured. We see that bash process 2098 forked a child process but does nothing due to PID namespace fracturing. In addition, this graph implies that the `libnss` library on the host was compromised, which is incorrect. In contrast, CLARION eliminates graph fracturing and provides a sound and clear understanding of how this attack is initiated from inside the container. Specifically, (1) CLARION marks the boundary of containers so we know the starting malicious script `modify.sh` is run inside the container; and (2) CLARION provides the mapped path for the library file so we know the compromised `libnss_files.so` is inside the container.

5.1.3 CVE 2018-15664: Symlink TOCTOU Exploit

In Docker (versions <= 18.06.1-ce-rc2), API endpoints behind the `docker cp` command are vulnerable to a symlink-exchange attack with Directory Traversal. This gives attackers arbitrary read-write access to the host filesystem with root privileges, because `daemon/archive.go` does not do archive operations on a frozen filesystem (or from within a `chroot`) [4].

When `docker cp` attempts to use a symlink from the container directory, it must find the absolute pathname file or directory in the context of the container filesystem. Failing to do so causes the symlink to be resolved in the host's filesystem, which allows symlinks created inside the container to affect files outside the container. When a user executes `docker cp` on a container filesystem, the Docker daemon process first executes a `FollowSymlinkInScope()` function, which returns the non-symlink path to the file/directory. Only after finding the actual path for both source and destination filenames does `docker cp` start copying files. One problem that arises from this process is that there is no guarantee that the filesystem won't change between running `FollowSymlinkInScope()` and copying the files. Here, a possible attack utilizing a Time-of-Check to Time-of-Use (TOCTOU) race condition is to have a process inside the container apply a symlink right after Docker verifies the symlink path, and right before docker begins writing the file. When docker uses the filename it resolved earlier, it will traverse the symlink to a file on the host

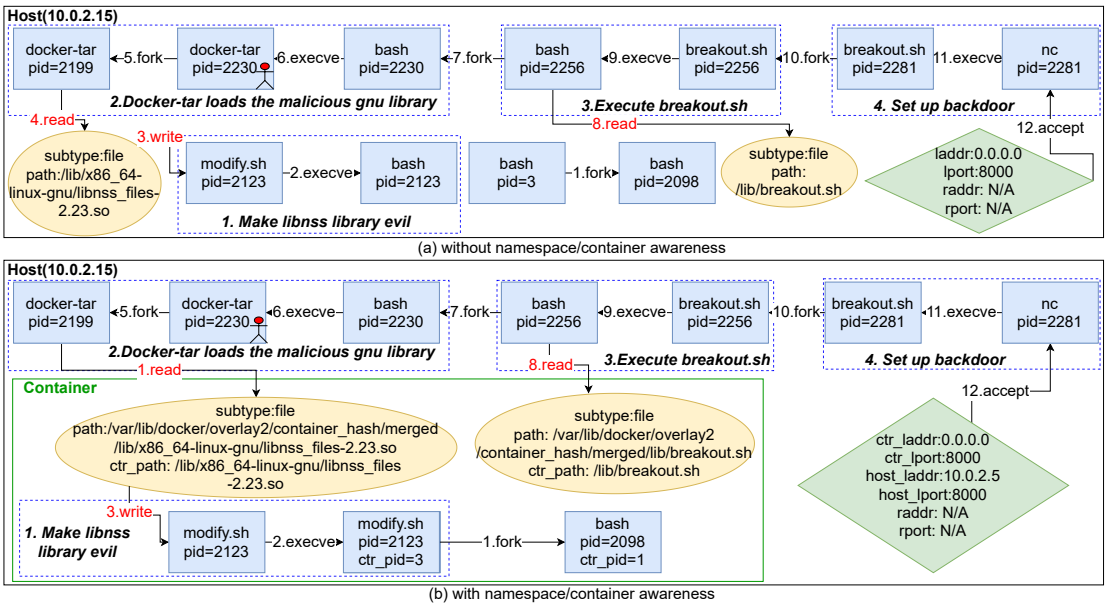


Figure 12: CVE 2019-14271: Provenance graph for the `docker-tar` exploit without (top) and with (bottom) namespace/container awareness

machine.

Through this exploit, a container process could potentially overwrite any file in the container when the host tries to copy a file into that container. This includes crucial system files such as password and user records. In our example, we use the four steps shown below in the order when attackers win the race condition to reproduce the exploit:

1. Host tries to copy a file `w00t_w00t_im_a_flag` from the container's filesystem to the host system by running `docker cp`.
2. Docker engine resolves both source and destination filenames, traversing any necessary symlinks.
3. Malicious process inside the container creates another directory (`stash_path`), applies a symlink to `/`, and performs a rename exchange of the original directory containing `w00t_w00t_im_a_flag`, i.e., `totally_safe_path`, with `stash_path`.
4. Docker engine, uses the filename resolved at Step 2, and performs a read of the container filename, and writes the data to the host filesystem.

Once the malicious process wins the race condition (Step 3), the symlink will be resolved in the host's filesystem and `docker cp` ends up copying the `w00t_w00t_im_a_flag` in the host, rather than the one inside the container. For this exploit, the provenance generated by CLARION graph shown in Figure 13 does not show significant difference from the namespace-unaware graph, because there is only one malicious process which will be run from at container start. Yet, without namespace awareness, the analyst will not be able to know that the key malicious process, i.e., `symlink_swap`, is running inside a container.

5.2 Cross-container Evaluation

To demonstrate that our solution is generic to several popular container engines together with deeper insights about provenance graph statistics, we select LXC (a classical container engine), rkt (a container engine with the second highest market share), Mesos and Docker for evaluation.

5.2.1 Initialization Graphs

We show the provenance graphs for the initialization of a `hello-world` container within each container engine in Figures 14, 15, and 16.

We find the initialization provenance graphs for the three different container engines to be clear and intuitive. They show that even when varying initialization routines are employed by different container engines, (e.g., rkt doesn't start the container before it finishes changing root path, while the other two use the first process inside the container), our initialization patterns always detect them accurately. Moreover, CLARION successfully summarizes the container boundary for all three container engines.

5.2.2 Quantitative Provenance Graph Results

We measured the impact of CLARION on provenance graph statistics to quantitatively assess the implications of namespace awareness with various container engines. We selected five popular Docker images that cover typical use cases in microservices including the base OS (ubuntu), a popular database (redis), a continuous integration server (jenkins) and a web server (nginx). We ran those images on three popu-

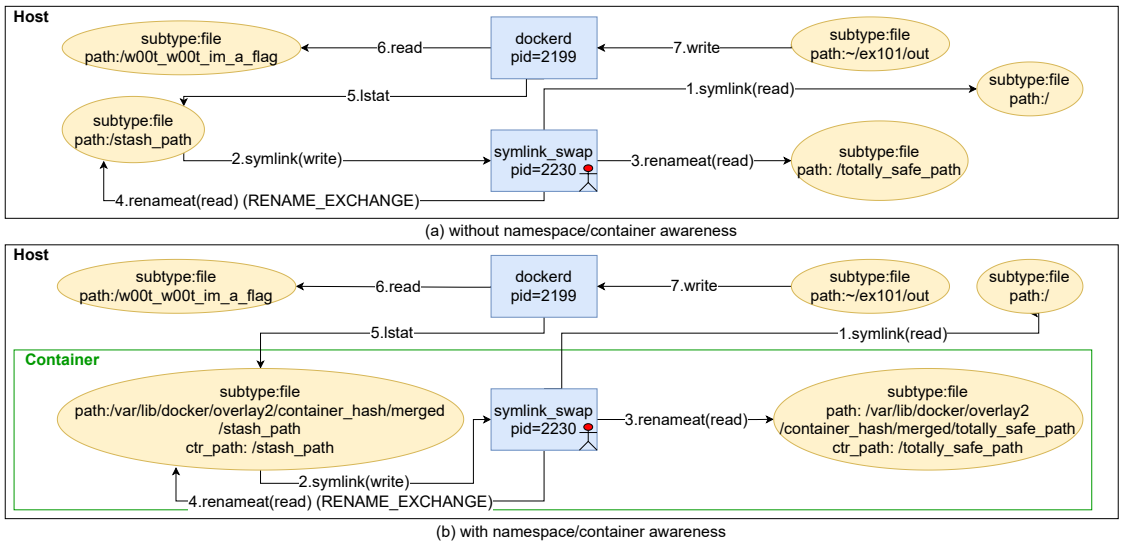


Figure 13: CVE2018-15664: `docker cp` race condition exploit without and with namespace/container awareness. Steps 1 and 2 are attempts to establish the symlink between the stash path inside the container and the root path on the host. Steps 3 and 4 represent the renaming exchange between the symlinked stach path and the path of the file to be copied. Steps 5-7 show that `dockerd` didn't resolve the correct path and ultimately copies the incorrect file.

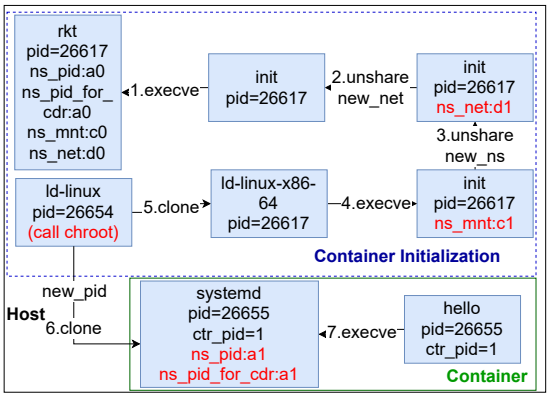


Figure 14: Initialization of a hello-world rkt container

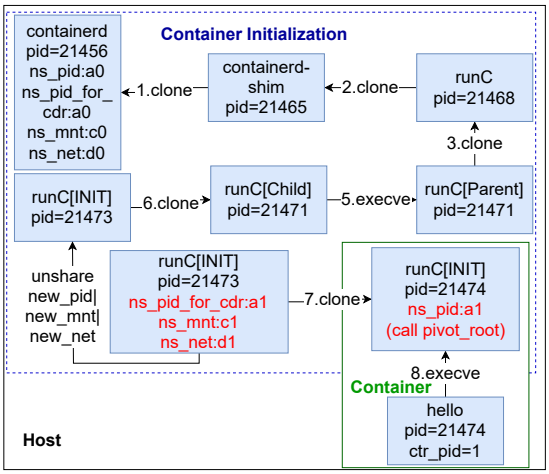


Figure 15: Initialization of a hello-world Docker container

lar container engines: Docker, rkt and Mesos. The results are reported in Tables 7, 8, and 9, respectively. For each image, we collected the behavior from container initialization to stable operation. In addition, we used two advanced configurations for nginx to highlight the effect of namespace awareness. *MT-4* indicates that we ran the nginx server with worker_process=4, while *MC-4* means we ran four nginx containers concurrently.

We see that in most cases, the total count of vertices and edges are not significantly impacted by the addition of namespace awareness. This is because it is possible for namespace unawareness to add or reduce vertices/edges, depending on the workload. For example, process cloning leads to more spurious vertices while false dependencies due to shared filenames in the mount namespace results in fewer vertices. Generally speaking, fewer vertices and more edges will be a better result because the provenance graph suffers from less fracturing. Hence, we count the (lost/extra) error vertices/edges in two common cases, i.e., process creation and file access, causing fragmentation and ambiguities. Though this may not cover all cases causing error vertices/edges, we believe it provides a useful lower bound to illustrate the severity of the soundness issue. Finally, in the case of components, we can observe significant differences when namespace awareness is turned on. Specifically, in nginx(*MC-4*) for rkt, we can see the components of SPADE are doubled in comparison to CLARION, meaning the corresponding provenance graph fractures significantly. This is because the four-container setting has more workload inside the containers and so the subgraph inside the container is much larger. Since the namespace-unaware system will fail to infer correct provenance inside containers,

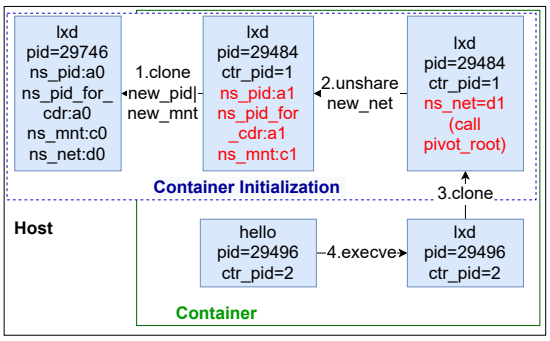


Figure 16: Initialization of a hello-world LXC container

the whole graph becomes more fractured as well. These results underscore how, especially in microservice scenarios, namespace-unawareness can lead to significant errors due to both fragmentation and ambiguities.

5.3 Efficiency Evaluation

Our efficiency evaluation consists of two parts: runtime overhead evaluation and storage overhead evaluation. We deployed a microservice benchmark and conducted a performance comparison between SPADE and CLARION.

5.3.1 Experiment Setup

The server machine we used has a configuration of Xeon(R) E5-4669 CPU and 256 GB memory. The microservice benchmark we selected a very popular microservice demo, **Online Boutique** [10], provided by Google. It contains 10 representative microservices and a web-based e-commerce app in which users can browse items, add them to the cart, and purchase them (i.e., a typical use-case for modern microservices).

5.3.2 Runtime Overhead

To compute the runtime overhead, we started every microservice independently 100 times and recorded the cumulative time for those 100 microservice containers to be initialized. First, we performed this process for each microservice without any audit subsystem enabled to get a baseline. Next, we repeated this evaluation with Linux Audit, SPADE, CLARION, CamFlow, and Linux Audit with SE-Linux labeling [6].

We summarize the detailed results in Table 10. The incremental overhead is calculated by comparing CLARION's overhead with that of SPADE. The "overall overheads" are based on comparison against the performance of the Base system. We find that the additional runtime overhead on SPADE imposed by CLARION is under 5% which we consider to be acceptable.

[6]Our objective is to obtain an estimate for Winnower's computational overhead. Unfortunately, because we do not have access to the Winnower system, we use Linux Audit with namespace-aware audit rules and SE-Linux-enabled Docker to obtain the results shown under SEL-Audit. We believe SEL-Audit results can serve as a lower-bound estimate of Winnower's computational overhead as Winnower uses Linux Audit and relies on SE-Linux labels. This does not measure the cost associated with Winnower's graph reduction or anomaly detection functionality.

The overall overhead of CLARION consists of SPADE overhead and CLARION's (PID namespace, Netfilter) kernel module overhead. By comparing values in the Base column with CLARION's overhead columns, we see that the major overhead originates from Linux Audit as opposed to extra modules introduced by CLARION.

5.3.3 Storage Overhead

We compare the size of raw logs collected by SPADE and CLARION in the aforementioned microservice environment with all 10 microservices. We collected logs for 24 hours and the results are shown in Table 11. We see that the additional storage overhead for CLARION is modest (under 5%) and much lower than CamFlow.

6 Related Work

Container Security. With the growing popularity of container-based virtualization, numerous security issues have been identified in container orchestration systems [8,9,14,18]. The reasons for these security issues may be attributed to a diverse set of flaws in design assumptions. For instance, to simplify support for file-system features like "bind mount", container engines, such as Docker do not enable the user namespace by default because this leads to file access privilege problems. But disabling the user namespace also implies that the root user inside the container also becomes the root user outside the container. In several aforementioned security issues, attackers simply leverage this general vulnerability to achieve privilege escalation on the host OS. Given the prevalence of such security issues, developing defensive technology that supports security analysis in container environments is crucial. This paper describes a first step toward a robust forensics analysis framework for containerized application deployments.

Container Vulnerability Analysis. Many existing efforts [37] have focused on the problem of container system vulnerability analysis. One line of work leverages traditional static analysis techniques to perform compliance checking on container images, such as those built with Docker. However, they do not protect the integrity of container instances at runtime [33,40]. Thus, contemporary container vulnerability analysis tools are limited in their ability to conduct long-term forensic analysis. Our study complements current container vulnerability analytics by providing a dynamic analysis view that leverages semantics-aware comprehension of attacks targeting running containers.

Provenance Tracking and Causality Analysis. Provenance tracking and causality analysis have played a vital role in system forensics [31, 32, 34, 36]. These tools build provenance/causal graphs by connecting system objects like processes, files, and sockets by using low-level events, such as system calls. When an attack entry point is identified, forward and backward tracking along graphs can then be performed to find the attack-related subgraphs. These allow analysts to get

Table 7: Provenance Graph Statistics Comparison (Docker)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	58 (8/50)	900	4236 / 4152	19056 / 19066	22 / 22
redis	78 (18/60)	1612	4759 / 4677	22856 / 22871	23 / 22
jenkins	55 (2/53)	133	4673 / 4581	21024 / 21026	28 / 25
node	72 (9/63)	919	4473 / 4387	19371 / 19376	24 / 21
nginx	72 (18/54)	1558	4737 / 4637	20780 / 20841	26 / 21
nginx MT-4	73 (19/54)	1662	7467 / 7345	40711 / 40781	32 / 26
nginx MC-4	376 (135/241)	7492	23875 / 23233	119128 / 119372	49 / 31

Table 8: Provenance Graph Statistics Comparison (rkt)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	80 (59/21)	10076	19047 / 19031	88022 / 88114	28 / 27
redis	171 (145/26)	12540	19348 / 19330	90471 / 90573	26 / 26
jenkins	99 (84/15)	10749	19441 / 19420	90798 / 90893	28 / 28
node	138 (103/35)	13334	19600 / 19575	90029 / 90125	27 / 25
nginx	85 (69/16)	10671	19666 / 19617	90885 / 91063	34 / 28
nginx MT-4	101 (70/31)	15272	23761 / 23721	106599 / 106754	40 / 33
nginx MC-4	828 (726/102)	65022	92962 / 93158	425550 / 426194	66 / 36

a clear understanding of the attack origin and its impact on the system. Several prior efforts have proposed mechanisms that seek to improve the quality of generated provenance/causal graphs [31, 32, 36] in different ways. While some of these attempt to mitigate the dependency explosion problem and eliminate unrelated data [41], others focus on real-time and scalable graph generation [34]. As described in Section 2, systems such as Winnower [26] and CamFlow [39] also have limitations. CamFlow has namespace awareness but not container awareness (i.e., it only extracts namespace identifiers, but does nothing to deal with container semantics or container boundaries.) In contrast, Winnower is container-aware but not namespace-aware. Although it uses SELinux label information to assign docker container IDs for process, file, and socket objects, those labels are not sufficient to fully disambiguate the effect of important syscalls like `clone`, `fork`. However, since both Winnower and CLARION run on SPADE, the two systems are complementary and could potentially be integrated. Our work is also more general and agnostic to specific container-management frameworks.

Alternative OS-level Virtualization Techniques. Multiple OS-level virtualization techniques exist on other operating system platforms. Among all those techniques, Solaris zones [11] and FreeBSD jails [12] show considerable similarity to Linux namespaces because both of them seek to provide isolation of system resources virtualized by Linux namespaces, e.g., process identifiers, filesystem and network stack, while sharing the same underlying kernel. Although conceptually similar, provenance effects from these techniques depend on multiple factors including virtualized resources, OS platforms, audit frameworks, etc. We provide a summary of our investigation into BSD Jail and Solaris Zones in Section 3.1.

7 Conclusion

In this paper, we present a comprehensive analysis of the soundness and clarity challenges introduced in data provenance analysis by Linux namespaces and containerization. Our analysis informed the development of CLARION, a namespace-aware provenance tracking solution targeting Linux container-based microservice deployments. Specifically, we resolved the soundness challenges introduced in each of the Linux namespaces affected by containerization and developed abstraction patterns to clarify container-specific semantics. We demonstrated the wide applicability of our solution by illustrating the generation of namespace-aware provenance graphs across multiple container engines. Evaluation results on real-world microservice benchmarks show that our solution is more effective than state-of-the-art provenance-tracking techniques and introduces acceptable additional overhead.

Acknowledgements

We thank our shepherd, Kevin Butler, and the anonymous reviewers for their insightful comments and suggestions. This work was sponsored in part by the U.S. Department of Homeland Security (DHS) Science and Technology Directorate under Contract HSHQDC-16-C-00034 and the National Science Foundation under Grants 1514503 and 1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DHS or NSF and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DHS, NSF or the U.S. government.

Table 9: Provenance Graph Statistics Comparison (Mesos)

Service	Error Vertices (lost/extra)	Error Edges	Vertices SPADE/CLARION	Edges SPADE/CLARION	Components SPADE/CLARION
ubuntu	10 (5/5)	241	28019 / 27932	76555 / 76561	18 / 17
redis	30 (18/12)	3149	19667 / 19574	59504 / 59507	17 / 17
jenkins	267 (210/57)	25453	34664 / 34560	141381 / 141387	26 / 24
node	21 (9/12)	1106	4960 / 4864	15492 / 15495	16 / 15
nginx	23 (20/3)	2389	5159 / 5067	17580 / 17582	20 / 17
nginx MT-4	23 (20/3)	2418	5185 / 5093	22545 / 22547	17 / 16
nginx MC-4	1402 (1383/19)	30304	19606 / 18817	66972 / 66982	30 / 22

Table 10: Runtime Overhead Comparison of Container Provenance Systems

Service	Base (secs)	Linux Audit (secs)	SPADE (secs)	CLARION (secs)	Incremental Overhead (CLARION)	Overall Overhead (Audit + SPADE + CLARION)	Overall Overhead (CamFlow)	Overall Overhead (SEL-Audit)
frontend	1503 s	1550 s	1558 s	1578 s	1.3%	3.7%	4.8%	32.4%
productcatalog service	668 s	679 s	681 s	691 s	1.5%	3.4%	9.1%	25.0%
currencyservice	1104 s	1139 s	1153 s	1169 s	1.4%	5.9%	12.9%	8.5%
paymentservice	1082 s	1123 s	1126 s	1143 s	1.5%	5.6%	11.5%	9.7%
shippingservice	434 s	446 s	449 s	451 s	0.4%	3.9%	22.5%	25.8%
emailservice	929 s	960 s	1028 s	1068 s	3.9%	15.0%	1.2%	17.6%
checkoutservice	682 s	719 s	714 s	734 s	2.8%	7.6%	3.2%	13.9%
recommendation service	8726 s	9418 s	9337 s	9729 s	4.2%	11.5%	9.5%	19.5%
adservice	4438 s	4454 s	4518 s	4571 s	1.2%	3.0%	5.3%	8.5%
loadgenerator	200 s	208 s	212 s	215 s	1.4%	7.5%	20.4%	29.4%

Table 11: Storage Overhead Comparison

SEL-Audit	CamFlow	SPADE	CLARION	Incremental Overhead
168.79 GB	312.56 GB	174.68 GB	181.75 GB	4.05%

References

[1] Apache Mesos. http://mesos.apache.org/.

[2] AWS Serverless Computing Services. https://aws.amazon.com/serverless/?nc1=h_ls.

[3] CoreOS rkt. https://coreos.com/rkt/.

[4] CVE-2018-15664 (Symlink TOCTOU). https://nvd.nist.gov/vuln/detail/CVE-2018-15664.

[5] CVE-2019-14271 (Docker-tar). https://nvd.nist.gov/vuln/detail/CVE-2019-14271.

[6] CVE-2019-5736 (RunC). https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[7] Docker. https://www.docker.com/.

[8] Docker Engine Large Integer Denial of Service Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2018-20699.

[9] Escape of Play-with-Docker Containers. https://threatpost.com/hack-allows-escape-of-play-with-docker-containers/140831/.

[10] Google Microservice Demo: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[11] Introduction to Solaris Zones. https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html.

[12] Jails in FreeBSD Handbook. https://docs.freebsd.org/en/books/handbook/jails/.

[13] Kernel Self-Protection Docs. https://www.kernel.org/doc/html/latest/security/self-protection.html.

[14] Kinsing Malware on Containers. https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability.

[15] Linux Namespace. https://www.man7.org/linux/man-pages/man7/namespaces.7.html.

[16] LTTng. https://lttng.org/.

[17] LXC: Linux Container Docs. https://linuxcontainers.org/lxd/docs/master/.

[18] Misconfigured Containers Again Targeted by Cryptominer Malware. https://t.co/J2WXp51xIK.

[19] Netfilter Architecture. https://www.netfilter.org/.

[20] Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html.

[21] Sysdig. https://github.com/draios.

[22] Adam Bates, Dave Jing Tian, Kevin Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *24th USENIX Security Symposium*, 2015.

[23] Ashish Gehani and Dawood Tariq. SPADE: Support for provenance auditing in distributed environments. In *13th ACM/IFIP/USENIX International Conference on Middleware*, 2012.

[24] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The Taser intrusion recovery system. In *20th ACM Symposium on Operating Systems Principles*, pages 163–176, 2005.

[25] Steve Grubb. Redhat Linux Audit. https://people.redhat.com/sgrubb/audit/.

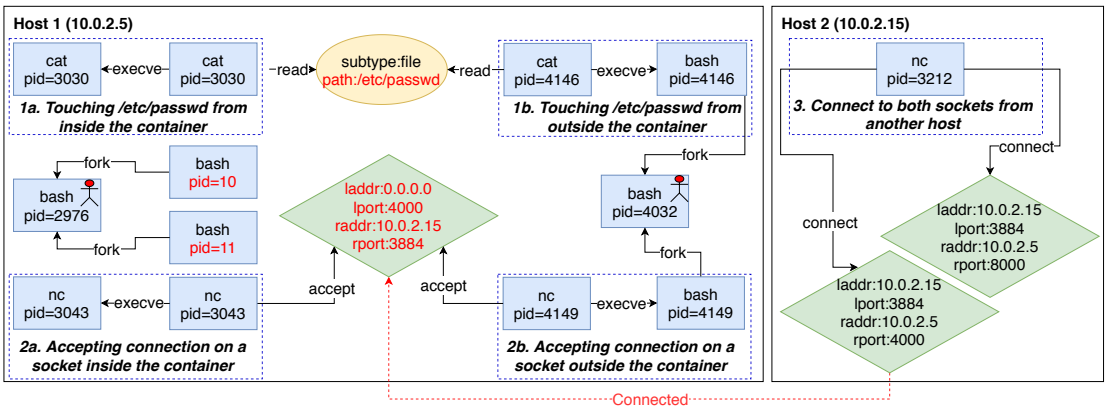


Figure 17: Motivating Example: Trivial insider attack.

[26] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *25th Network and Distributed System Security Symposium*, 2018.

[27] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and Frans Kaashoek. Intrusion recovery using selective re-execution. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[28] Samuel King and Peter Chen. Backtracking intrusions. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[29] Samuel King, Zhuoqing Morley Mao, Dominic Lucchetti, and Peter Chen. Enriching intrusion alerts through multi-host causality. In *12th Network and Distributed System Security Symposium*, 2005.

[30] Srinivas Krishnan, Kevin Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *17th ACM Conference on Computer and Communications Security*, 2010.

[31] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *25th Network and Distributed System Security Symposium*, 2018.

[32] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Network and Distributed System Security Symposium*, 2013.

[33] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *34th Annual Computer Security Applications Conference*, 2018.

[34] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *25th Network and Distributed System Security Symposium*, 2018.

[35] John Lyle and Andrew Martin. Trusted computing and provenance: Better together. In *2nd USENIX Workshop on the Theory and Practice of Provenance*, 2010.

[36] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Network and Distributed System Security Symposium*, 2016.

[37] Antony Martin, Simone Raponi, Theo Combe, and Roberto Di Pietro. Docker ecosystem – Vulnerability analysis. *Computer Communications*, 122, 2018.

[38] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *11th USENIX Security Symposium*, 2002.

[39] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *8th ACM Symposium on Cloud Computing*, 2017.

[40] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding security implications of using containers in the cloud. In *28th USENIX Annual Technical Conference*, 2017.

[41] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. NodeMerge: Template based efficient data reduction for big-data causality analysis. In *25th ACM Conference on Computer and Communications Security*, 2018.

A Insider Attack: Detailed Steps

The attack partially shown in the motivating example is illustrated in Figures 20. The attack involves 3 steps.

Step 1: The `cat` command is used in a bash shell to read the `/etc/passwd` file.

Step 2: The same bash shell is then used to set up a backdoor using the netcat (`nc`) tool on socket with IP address / local port (`0.0.0.0/4000`).

Step 3: Connection is established to this backdoor port from a remote host.

We perform those steps inside a container and on the host. The container is initialized with a port mapping from port `TCP/4000` inside the container to port `TCP/8000` on the host. It looks like the `netcat` process was listening on port `TCP/4000` of this container, but in fact it was listening on port `TCP/8000` on the host. The bash shell processes which start the attacks are process 2976 and process 4032.

As shown in Figure 20, we cannot identify which attack is performed in the container. Furthermore, the process creation provenance, clone between PID 2976 and PID 3030 (VPID 10), inside the container is broken, resulting in *fragmentation*.

In addition, we only see one file and one socket being touched in the graph because two touched files have the same virtualized paths and two connected sockets have the same local addresses, leading to *ambiguity*.