Querying Container Provenance

Aniket Modi[*]
DePaul University
Chicago, USA

Moaz Reyad
DePaul University
Chicago, USA

Tanu Malik
DePaul University
Chicago, USA

Ashish Gehani
SRI International
Menlo Park, USA

ABSTRACT

Containers are lightweight mechanisms for the isolation of operating system resources. They are realized by activating a set of namespaces. Given the use of containers in scientific computing, tracking and managing provenance within and across containers is becoming essential for debugging and reproducibility. In this work, we examine the properties of container provenance graphs that result from auditing containerized applications. We observe that the generated container provenance graphs are hypergraphs because one resource may belong to one or more namespaces. We examine the hierarchical behavior of *PID*, *mount*, and *user* namespaces, that are more commonly activated and show that even when represented as hypergraphs, the resulting container provenance graphs are acyclic. We experiment with recently published container logs and identify hypergraph properties.

1 INTRODUCTION

The conduct of reproducible science improves when computations are both portable and evaluable. A container provides an isolated environment for running computations and thus is useful for porting applications on new machines. Managing an array of virtualized containers is becoming increasingly typical for data and code sharing platforms such as Binder [2], Figshare [4] and Hydroshare [5] that enable users to port applications and execute them repeatedly on the platform.

Despite isolation, applications may fail to reproduce, especially as containerized applications are run repeatedly with different input datasets and parameters [17]. Since application evaluation for reproducibility may happen at different points in time, it is essential to track provenance of applications within containers to provide insights and comprehend the causes of failure [11]. Tracking the provenance of containerized applications, however, raises some unique research challenges. Containers are ephemeral with a limited lifetime [13]. Once an execution completes, the container runtime frees up resources. This necessitates that provenance records are archived on persistent storage so we can reuse them during assessment and subsequent evaluations.

[*]Also with IIT Delhi.

WWW '23 Companion, April 30-May 4, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9419-2/23/04…$15.00
https://doi.org/10.1145/3543873.3587568

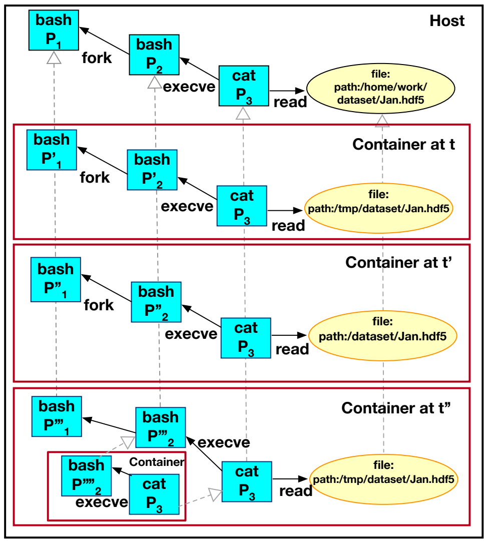

Figure 1: Container provenance graphs at different points in time. We can match the container graph at t, t', and t'' with the host view (top) if all (grey) dashed edges are known. Current provenance systems do not explicitly model dashed edges in grey.

One possible design policy is to securely share these records with the shared-host substrate, which provides a centralized platform and is aware of the array of containers running on it. Consider a shared substrate that stores the system level provenance graph of an application run at time t and then subsequently at time t' (Figure 1). Resolving cross-container provenance records is challenging, as the same physical resource may appear differently within isolated contexts and at different points in time. As shown in Figure 1 the same file at path $/home/work/dataset/Jan.hdf5$ is visible as $/tmp/dataset/Jan.hdf5$ first time but gets mounted as $/dataset/Jan.hdf5$ next time. An alternative approach is for the shared substrate to be container-aware and collect records so that only the host's view (top view in Figure) is persisted. However, users of containerized applications are not aware of resource specification from the host's view, which in the case of Figure 1 is the path $/home/work/dataset/Jan.hdf5$. Consequently, tracking records from both the host substrate and the container-specific execution becomes necessary. This also necessitates that the host substrate effectively maintains the mapping (grey lines) between the host view and the isolated contexts.

In this paper, we consider issues in representing cross-container records at the shared substrate for container provenance analysis. With container-awareness processes map to different isolated contexts, such as P_1 maps to P_1' and P_1'' in Figure 1. Further, P_2 and P_3

may `unshare` at a later point in time t'' and form a new isolated context but continue to read the file from the same path. In this case, the the resulting container provenance graph has not only pair-wise edges between files and processes in the same isolated context, but must also maintain non-pairwise relationships between files and the process identifiers in different isolated contexts. We show that such higher-order relationships are easily modeled as *hyper graphs* at the shared substrate. Hypergraphs are multigraphs that allow edges between multiple nodes of a traditional graph. We consider different lineage queries on container hyper graphs. We show that despite being namespace-aware the resulting container hypergraph is acyclic. We experiment with Docker benchmarks and Kubernetes Common Vulnerabilities and Exposures (CVEs) database and identify hypergraph structure in container provenance graphs. We show that while hyperprocesses and hyperfiles exist, the current logs are not high-fidelity and do not represent full container provenance.

The rest of this paper is structured as follows. We cover background on namespaces, containers, and provenance tracking in containerized hosts in Section 2. Section 3 shows how resources across namespaces map to nodes in a hypergraph, and describe acyclicity of forward lineage queries in Section 3.1. We then describe preliminary experiments, related work, and conclusions in Sections 4, 5, and 6, respectively.

2 BACKGROUND

We provide basic background information on namespaces, Linux containers and auditing container provenance.

2.1 Namespaces

An operating system namespace provides an illusion to a set of processes that they have complete control of a resource. The kernel ensures that a namespace is isolated, allowing a global resource to be shared without any change to the application's interfaces to the system. The Linux kernel wraps identifiers of various global system resources such as PIDs, hostnames, mount points, user identifiers, time, network devices, ports, interprocess-communication, and resource accounting information with namespaces. Each of the namespaces provides an isolated view of the particular global resource to the set of processes that are members of that namespace. Figure 2 shows an example of the mount namespace.

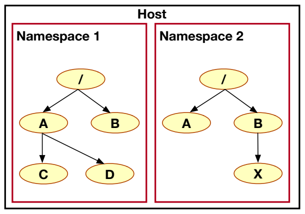

Figure 2: The behavior of mount namespaces. The root, A and B mount points are shared but then the namespaces can continue to grow independently.

One of the significant uses of namespaces is to support the implementation of containers, a tool for lightweight virtualization. Within containers, our examples focus on PIDs and mount point resources, since data flow tracking heavily relies on these resources, but our approach of modeling provenance graphs over namespaces applies to all kinds of system resources.

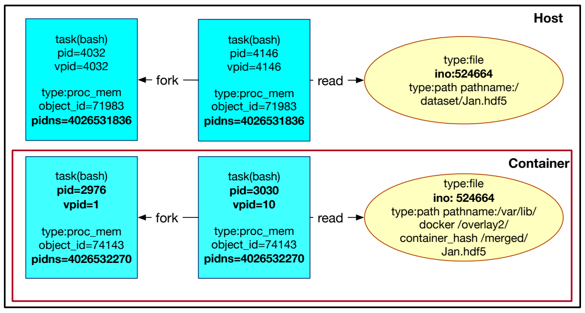

Figure 3: Sound container provenance graphs from OS-level provenance tracking systems [11]

2.2 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 engines using their runtime. There are several runtimes such as LXC[7], rkt[8], Mesos[1], Docker[3], and Singularity[15]. Each of these runtimes differ in their application programming interface (API) and how they manage creation, destruction and persistence of namespaces. Our treatment of provenance tracking is at the system level and thus while we respect the same container boundary that all engines recognize, our formalism is independent of the specific APIs used by the specific runtime.

2.3 Namespace and Container-awareness in Provenance Systems

Figure 3 shows a provenance graph of a containerized application running on a host system that is also executing the same application. The graph is obtained from provenance systems that track data flows at the operating system level [12, 18]. We particularly note that the Linux auditing mechanisms such as Linux Audit, SysDig, and Lttng do not automatically generate such sound provenance graphs. Current provenance tracking systems rely on a combination of host-container mapping view and namespace-labeling approaches that disambiguate and map virtual nodes with host nodes on the provenance graph to generate sound provenance graphs [9, 11]. This soundness property is demonstrated in the Figure 2, as it shows a process' real identifier (*pid* value) and a virtual identifier (*vpid* value) in the containerized namespace. If the process' *vpid* value is different from the process' *pid* value then it only lists the process in the containerized namespace. An example is *pid*=3030 and *vpid*=10, where *pid* is in host namespace and *vpid* in containerized namespace. Similarly, the virtualized file path is different from the real file path even though the underlying *inode* is the same.

3 QUERYING CONTAINER PROVENANCE

Sound provenance records collected by provenance tracking systems are typically maintained at the host substrate. These records

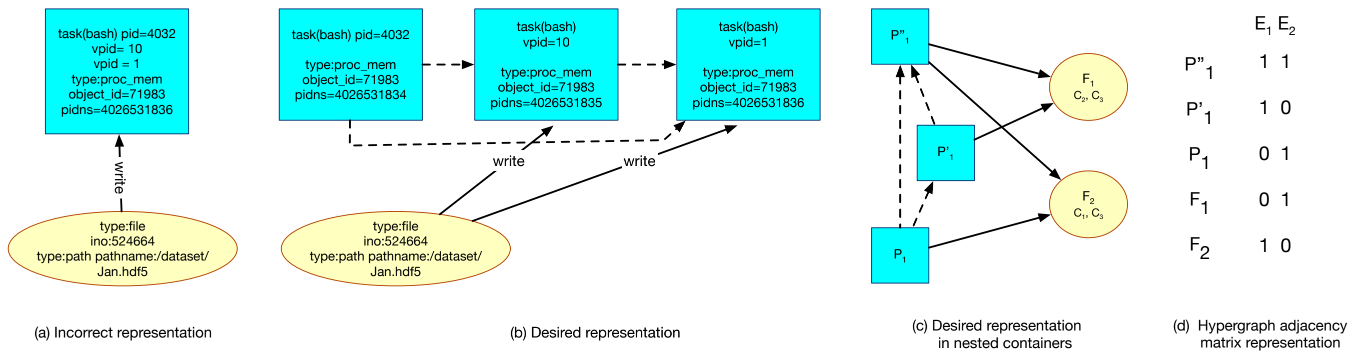

Figure 4: Hypergraph representation of container graphs

include edges between process and file nodes, but maintain namespace relationships as properties of the node and not as a graph relationship. From a querying perspective, the representation of namespace information within the audited provenance graph is sub-optimal. Consider the following queries on container graphs: (i) list the processes running in namespace 4026531836, and (ii) find which processes identifiers wrote to file '/dataset/Jan.hdf5'. The first query will return only *pids* 4032 and 4146 even though process 3030 is also in the same namespace. Similarly for the second query, as shown in Figure 4(a) we will return all the *pids* even though as shown in Figure 4(b) the file was only visible within *pid* namespaces 4026531835 and 4026531836.

3.1 Modeling Container Provenance as Hypergraphs

We observe that to answer the above queries correctly, process nodes in different namespaces must be represented as separate nodes in the graph. Also, read or write action between the group of processes and file nodes must be represented such that they respect container boundaries. Consider Figure 4(c), which represents the combined scenario occurring in the two queries: The physical process P_1'' in C_3 exists in two parent container namespaces C_2 and C_1 as P_1' and P_1, respectively; File F_1 is visible in container namespaces C_2 and C_3 and F_2 is visible in C_1 and C_3, respectively. This graph captures process nodes across namespaces using additional nodes: P_1 and P_1', and dashed edges to connect them. It still does not capture the higher-order write relations which actually connects multiple file and process nodes within container namespaces. A better way is to represent is in Figure 4(d) which groups the write between P_1', P_1'' and F_1 as one event, and the write between P_1, P_1'' and F_2 as another event.

Representing edges in Figure 4(c) as a grouped relation in that an edge can connect any number of vertices, leads to a hypergraph representation of Figure 4(d). In general, a hypergraph is a couple $H = (V, E)$ consisting of a finite set V and a set E of non-empty subsets of V. The elements of V are called vertices and those of E are called hyperedges. While a regular graph edge is a pair of nodes, a hyperedge $e \in E$ connects a set of vertices $\{v\} \subseteq V$.

A primary concern with lineage querying is ensuring that underlying graphs are acyclic, so conjunctive join queries do not take exponential time. In non-container system graphs, this is obtained via versioning of process and file nodes: every write to a file after close is versioned, and every read by a process leads to versioning of the process nodes. With process and file nodes arising due to namespaces as explicit nodes in a graph, we must ensure that the resulting graph is acyclic. In the following subsection, we define a path in a directed container hypergraph, and show that such a path will never be cyclic based on namespace system calls.

Definition 3.1. A directed hypergraph $H = (V, E)$, where V is a finite set of nodes and $\overrightarrow{E} \subset \{(T(e), H(e)) : T(e), H(e) \in P(V)\backslash\phi \,\&\, T(e) \cap H(e) = \phi\}$ is the set of directed edges.

In H, $P(V)$ is the power set of V, $T(e)$ and $H(e)$ are said to be the tail and the head of e respectively. The head and tail represent the set of nodes where the hyperedge ends and starts respectively. It is clear that $|T(e)| < 0$ and $|H(e)| > 0$.

Definition 3.2. A forward edge is a hyperedge $e = |T(e), H(e)|$ with $|T(e) = 1|$.

Definition 3.3. A simple directed hypergraph path from s and t in $\overrightarrow{H}$ is a sequence $(v_1, e_1, v_2, \ldots, v_{n-1}, e_{n-1}, v_n)$ consisting of (i) nodes v_i where $1 \leq i \leq n$, $v_i \in T(e_i)$, and (ii) distinct hyperedges e_j where $1 \leq j \leq n$ such that $s = v_1$ and $t = v_n$ and for every $1 \leq i \leq n$, $v_i \in T(e_i)$ and $v_i \in H(e_i)$.

Definition 3.4. A simple directed hypergraph path in hypergraph H from $s = v_1$ to $t = v_n$, $\overrightarrow{P} = (v_1, e_1, e_n, v_n)$ is called a cycle if $|T(e_1)| \geq 1$ and $t \in T(e_1)$.

We show for the namespaces that such path cycles do not exist.

- PID namespace. Cycles do not occur in PID namespaces because, while processes may freely descend into child PID namespaces (e.g., using setns(2) with a PID namespace file descriptor), they may not move in the other direction. That is to say, processes may not enter any ancestor namespaces (parent, grandparent, etc.). Changing PID namespaces is a one-way operation. This remains true irrespective of the type of namespace call such as clone, unshare, setns. Thus a process's PID namespace membership is determined when the process is created and cannot be changed thereafter. This means that the parental relationship between processes mirrors the parental relationship between PID namespaces: the parent of a process is either in the same namespace or resides in the immediate parent PID namespace.
- Mount namespace. Mount namespaces are not nested and yet cycles do not occur because use of system calls such as chroot

and pivot_root lead to unmounting of the host filesystem, making it impossible to access any file within it in a child namespace. This acyclicity is true irrespective of the mount flags used during propagation of mount points.

- User, network and UTS namespaces. These namespaces do not create cycles as these namespaces create one-one mapping between resources in the parent and child namespaces. For example, cycles do not occur in user namespace since `uid` and `gid` mappings are only set in the parent namespace for the child namespace. While the same user can be mapped to different identifiers in child namespaces, the mapping only leads to a hierarchical structure and thus avoids cycles.

4 HYPERGRAPH IMPLEMENTATION AND EXPERIMENTS

Our basic objective was to identify hypergraph structure in available container provenance graphs. We store the incidence matrix of the hypergraph, which stores the vertices that each hyperedge contains (rows correspond to vertices, columns correspond to hyperedges, and nonzeros i,j designate that hyperedge j contains vertex i). The incidence matrix allows for quickly determine if two processes are in the same namespace. We used three container provenance graphs that were generated in [9] which were on Docker benchmarks and Kubernetes CVEs. Table 1 shows basic details about container provenance graphs. In #processes and #files, the number outside bracket is the total number, including all versions of all files/processes and the number in bracket is the number ignoring versions. Table 2 shows identified hypergraph properties. The analysis ignores file versioning and if a file was introduced in multiple namespaces in a later version, and pathnames don't exist for that version, then that is not counted. A significant limitation of these graphs is that provenance is recorded only across atmost two namespaces in these graphs, and namespace annotations for files are missing, which limits the analysis. In future, we plan to collect provenance graphs both using DocLite [20] and simulated benchmarks.

Table 1: Log details.

Log	#vertices	#edges	#processes	#files
hotel_docker	472889	1058298	280562 (2958)	28074 (6140)
cve-2019-1002101	1023370	2176519	647336 (2223)	131024 (19842)
cve-2021-30465	1089634	3233319	655660 (3770)	77865 (12584)

Table 2: Hypergraph results

Log	#hyperprocesses	#hyperfiles	#paths
hotel_docker	209	1499	960
cve-2019-1002101	659	5753	982
cve-2021-30465	593	1965	805

5 RELATED WORK

Containers are implemented with Linux namespaces[14]. Both containers and namespaces create challenges for provenance collection. Clarion[11] solves the provenance clarity and soundness challenges that exist in Linux Audit framework[6]. Tracing the execution provenance of containers became an interesting problem in the security domain. There are systems that uses provenance to solve security challenges such as Container Escape Detection[9].

PROV[16] defines a provenance model and its serializations. The PROV data model (PROV-DM) does not define the concept of *container* or even a more generic concept like *context* that can be used to model containers. The most close concept is *collection* If we model computer resources (e.g. files, processes, users) as entities, we can add them to named *collections* through the *HadMember* relation. This is a very simple representation of a namespace as a collection that had members which are resources that belong to it. A more advanced form of context should be added to PROV if we need to model containers. PROV can be serialized with RDF or OWL. Both of them lack the support for contexts. PaCE[19] adds context to provenance as a special entity. C-OWL[10] defines a context in OWL by its local contents which are not shared. This is similar to namespaces local IDs of resources.

6 CONCLUSIONS

The increasing interest in containers and their wide usage in numerous applications inspired a careful study of their provenance. We presented the problem of querying provenance hyper graphs in containerized applications. We formalized the definition of hypergraphs and identified hypernodes and hyperedges in real world datasets. In the future, we plan to efficiently query large provenance hypergraphs.

ACKNOWLEDGEMENTS

This work is supported under NSF grant CNS-1846418 and NASA grant AIST-21-0095-80NSSC22K1485.

REFERENCES

[1] Apache mesos. https://mesos.apache.org/. Accessed: 2023-02-05.
[2] Binder. https://mybinder.org/. Accessed: 2023-02-05.
[3] Docker. https://www.docker.com/. Accessed: 2023-02-05.
[4] Figshare. https://figshare.com/. Accessed: 2023-02-05.
[5] Hydroshare. https://www.hydroshare.org/. Accessed: 2023-02-05.
[6] Linux audit. https://github.com/linux-audit. Accessed: 2023-02-05.
[7] Linux containers. https://linuxcontainers.org/. Accessed: 2023-02-05.
[8] rkt. https://github.com/rkt. Accessed: 2023-02-05.
[9] Mashal Abbas, Shahpar Khan, and et. al. Paced: Provenance-based automated container escape detection. In *IC2E*, pages 261–272. IEEE, 2022.
[10] Paolo Bouquet, Fausto Giunchiglia, and et. al. C-owl: Contextualizing ontologies. In *The Semantic Web-ISWC 2003*, pages 164–179. Springer, 2003.
[11] Xutong Chen, Hassaan Irshad, and et. al. {CLARION}: Sound and clear provenance tracking for microservice deployments. In *USENIX Security 21*, 2021.
[12] Ashish Gehani and Dawood Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. *Middleware*, 2012.
[13] Jack S. Hale, Lizao Li, and et. al. Containers for portable, productive, and performant scientific computing. *CiSE*, 19(6):40–50, 2017.
[14] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook.* No Starch Press, 2010.
[15] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.
[16] Luc Moreau and Paul Groth. Provenance: an introduction to prov. *Synthesis lectures on the semantic web: theory and technology*, 3(4):1–129, 2013.
[17] Yuta Nakamura, Tanu Malik, and Ashish Gehani. Efficient provenance alignment in reproduced executions. In *TaPP*, pages 6–12, 2020.
[18] Thomas Pasquier, Xueyuan Han, and et. al. Practical whole-system provenance capture. In *SoCC*. ACM, 2017.
[19] Satya S Sahoo, Olivier Bodenreider, and et. al. Provenance context entity (pace): Scalable provenance tracking for scientific rdf data. In *SSDBM*, 2010.
[20] Blesson Varghese, Lawan Thamsuhang Subba, Long Thai, and Adam Barker. Doclite: A docker-based lightweight cloud benchmarking tool. In *CCGrid*, 2016.