Querying Container Provenance

Aniket Modi, Moaz Reyad, Tanu Malik
School of Computing, DePaul University

Ashish Gehani
Computer Science Laboratory, SRI

ABSTRACT

Containers are lightweight mechanisms for the isolation of operating system resources. They are realized by activating a set of namespaces. Since use of containers is rising in scientific computing, tracking and managing provenance within and across containers is required for debugging and reproducibility. In this work, we examine the properties of container provenance graphs that result from auditing containerized scientific computing experiments. We observe that the generated container provenance graphs are hypergraphs because one resource may belong to one or more namespaces. We examine the behavior of three namespaces, namely the *PID*, *mount*, and *user* namespaces, that are prominently used in scientific computing and show that operations over namespaces do not result in cycles in the resulting container provenance graphs. Thus we can identify container boundaries, distinguish container processes from host processes, and answer conjunctive lineage queries in polynomial time. We experiment with complex lineage queries on container provenance graphs and show the hypergraph formulation helps us answer those queries more efficiently than a non-hypergraph formulation.

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], Hydroshare [4], WholeTale [8] 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 [19]. 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 [13, 23]. Tracking the provenance of containerized applications, however, raises some unique research challenges. Containers are ephemeral with a limited lifetime [15]. 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.

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 $/home/work/dataset/Jan.hdf5$

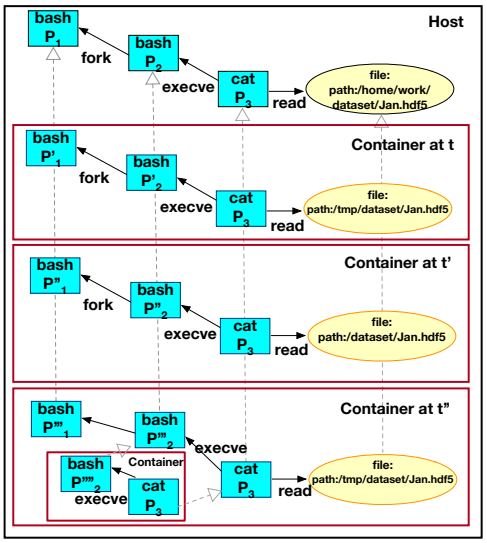

Figure 1: Container provenance graphs at different points in time. We can match the container graph at t, t', and t'' with the host view (top) if all (grey) dashed edges are known. Current provenance systems do not explicitly model dashed edges in grey.

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 maintaining this mapping of cross-container records at the shared substrate for container provenance analysis. Container-awareness results in processes mapping to different isolated contexts, such as P_1 mapping to P_1' and P_1''. In addition, processes such as P_2 and P_3 in Figure 1 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. Thus 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 and therefore conjunctive lineage querying on namespace-aware container records terminates in polynomial time. Our experiments show that our hyper graph formulation applies to records collected from Docker containers and related benchmarks, and that our queries terminate in reasonable time.

The rest of this paper is structured as follows. We provide a basic overview of namespaces, containers, and provenance tracking in containerized hosts in Section 2. We then show how resources across namespaces map to nodes in a hypergraph 3. We formulate a directed hypergraph and describe forward lineage queries are acyclic in Section 3.1. Section 4 describes an efficient implementation of hypergraphs with experiments. We discuss related work in Section 5, and we conclude in Section 6.

2 BACKGROUND

We provide basic background information on Linux containers and namespaces, and their provenance tracking.

2.1 Namespaces

An operating system namespace provides a set of processes the illusion that they have complete control of a resource. The kernel ensures that different instances of the same namespace are isolated, allowing a global resource to be shared without any changes to the application's interfaces to the system. The Linux kernel wraps various global system resources such as PIDs, hostnames, mount points, user identifiers, time, network devices and ports, interprocess-communication, and resource accounting information into 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. 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.

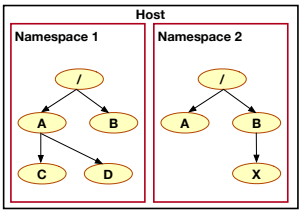

Figure 2: The behavior of mount namespaces. The root and A and B mount points are shared but then the namespaces can continue to grow independently.

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.

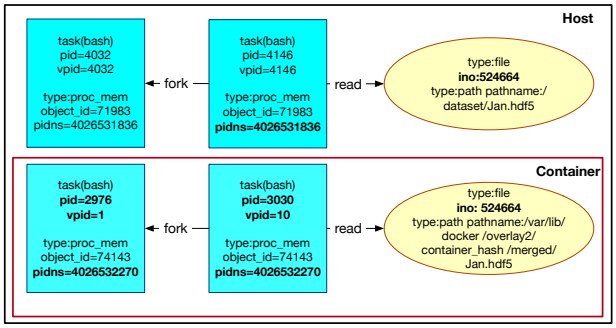

Figure 3: Sound container provenance graphs from OS-level provenance tracking systems [12]

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 engine using its container runtime. The container runtime will specify the namespaces to be shared among processes running inside the container. In general there are several runtimes such as LXC[6], rkt[7], Mesos[1], Docker[3], Singularity[17], and Charlie Cloud[21]. 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 [14, 20]. 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. This soundness property is demonstrated in the Figure as it shows for a process its real identifier in the host namespace and virtual identifiers in the containerized namespace. Similarly the virtualized file path is different from the real file path even though the underlying inode is the same.

From a querying perspective, however, the representation of namespace information within the audited provenance graph is sub-optimal. Consider the process id 3030 which is mentioned in namespace 4026532270 but is truly in namespace 4026531836. Thus queries such as "what are the processes running in namespace 4026531836?" will not return accurately. While one may argue that all processes are in the host namespace and such a query is easily modeled by querying all process nodes, the argument is not true in the case of nested containers or shared mount trees, which

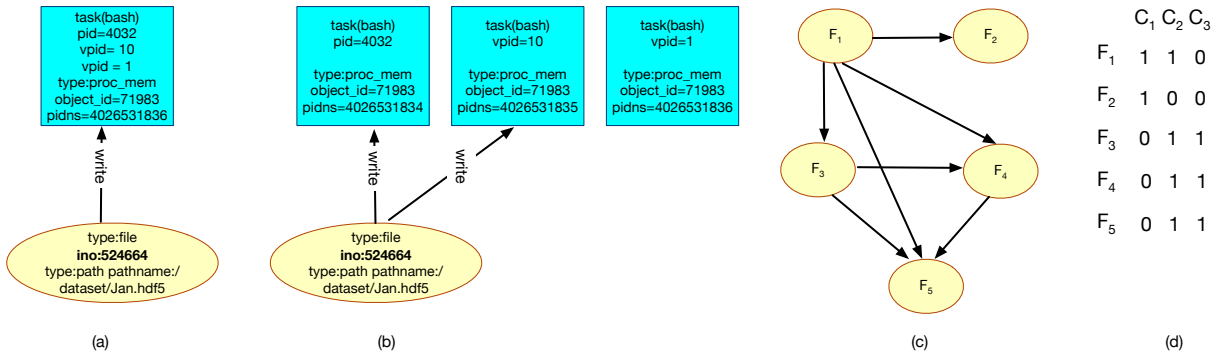

Figure 4: Container queries

are often created for performance purposes. Thus determining all the processes in a namespace will be incorrect and so will be any query that determines the forward or backward lineage path on which such nodes lie. Similarly the provenance semantics that file paths of the file used by processes appear distinct in the graph, but correspond to the same physical file are not captured.

3 QUERYING CONTAINER PROVENANCE

Sound provenance records collected by provenance tracking systems are typically maintained at the host substrate. These records include pairwise edges between process and files, but maintain namespace relationships as properties of the node and not as a graph relationship. Consider a provenance query on container graphs such as find which processes identifiers wrote to a file visible across namespaces 1-3, will return all processes identifiers across all namespaces since identifiers in different namespaces are not separated. Figure 4(a) distinguishes the returned query result from the expected one in Figure 4(b). Similarly, consider another provenance query on container graphs such as find all resources that were derived from each other in namespaces 1, 2, and 3. Figure 4(c) shows the directed subgraph that is obtained as a result of this query. However, the directed graph represents derivation across all namespaces. It does not reflect the grouped derivation between individual namespaces as shown in Figure 4(d).

We observe that to answer the above queries correctly, simple graph edges do not capture the higher-order relation which connects these multiple objects, but is more appropriately captured by a hypergraph which is a generalized graph data structure, where an edge can connect any number of vertices. 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 graph-based querying is ensuring that underlying graphs are acyclic, so conjunctive 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 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.

3.1 Acyclicity in container provenance

Definition 3.1. A directed hypergraph $H = (V, E)$, where V is a finite set of nodes and $\overrightarrow{E} \subset \{(T(e), H(e)) : T(e), H(e) \in P(V)$ $\phi \& T(e) \cap H(e) = \phi\}$ is the set of directed edges, where $P(V)$ 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 $|T(e)| < 0$, $|H(e)| > 0$.

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

We show for different namespaces 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

We have mimiced hyperedge structure using n-ary relationships in a Postgres database. 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 our result. The analysis ignores file versioning and if a file was introduced in multiple namespaces in a later version, and pathnames dont exist for that version, then that is not counted.

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[16]. Both containers and namespaces create challenges for provenance collection. Clarion[12] solves the provenance clarity and soundness challenges that exist in Linux Audit framework[5]. 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[18] 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. For RDF, PaCE[22] aims to add context to provenance as a special entity. For OWL, C-owl[11] defines a context by its local contents which is not shared. This is similar to namespaces local IDs of resources. Hyper graphs [10] model various types of objects and the relations between them. RDF data model can be represented as hyper graph natively in System Π[24].

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.

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] Hydroshare. https://www.hydroshare.org/. Accessed: 2023-02-05.
[5] Linux audit. https://github.com/linux-audit. Accessed: 2023-02-05.
[6] Linux containers. https://linuxcontainers.org/. Accessed: 2023-02-05.
[7] rkt. https://github.com/rkt. Accessed: 2023-02-05.
[8] Wholetale. https://wholetale.org/. 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] Claude Berge. Graphs and hypergraphs. 1973.
[11] Paolo Bouquet, Fausto Giunchiglia, and et. al. C-owl: Contextualizing ontologies. In *The Semantic Web-ISWC 2003*, pages 164–179. Springer, 2003.
[12] Xutong Chen, Hassaan Irshad, and et. al. {CLARION}: Sound and clear provenance tracking for microservice deployments. In *USENIX Security 21*, 2021.
[13] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *SIGMOD*, pages 593–596, 2012.
[14] Ashish Gehani and Dawood Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. *Middleware*, 2012.
[15] Jack S. Hale, Lizao Li, and et. al. Containers for portable, productive, and performant scientific computing. *CiSE*, 19(6):40–50, 2017.
[16] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.
[17] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.
[18] Luc Moreau and Paul Groth. Provenance: an introduction to prov. *Synthesis lectures on the semantic web: theory and technology*, 3(4):1–129, 2013.
[19] Yuta Nakamura, Tanu Malik, and Ashish Gehani. Efficient provenance alignment in reproduced executions. In *TaPP*, pages 6–12, 2020.
[20] Thomas Pasquier, Xueyuan Han, and et. al. Practical whole-system provenance capture. In *SoCC*. ACM, 2017.
[21] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Supercomputing*, pages 1–10, 2017.
[22] Satya S Sahoo, Olivier Bodenreider, and et. al. Provenance context entity (pace): Scalable provenance tracking for scientific rdf data. In *SSDBM*, 2010.
[23] Victoria Stodden, Marcia McNutt, and et. al. Enhancing reproducibility for computational methods. *Science*, 354(6317):1240–1241, 2016.
[24] Gang Wu, Juan-Zi Li, and et. al. System π: A native rdf repository based on the hypergraph representation for rdf data model. *Journal of Computer Science and Technology*, 24(4):652–664, 2009.