PROV-CRT: Provenance Support for Container Runtimes

Raza Ahmad, Yuta Nakamura, Naga Nithin Manne[#], Tanu Malik[*]
School of Computing,
DePaul University
Chicago, IL, 60604, USA
{raza.ahmad, ynakamu1, nmanne, tanu.malik}@depaul.edu

Abstract

A container runtime isolates computations and its associated data dependencies and is thus useful for porting applications on new machines. Current container runtimes, such as LXC and Docker, however, do not automatically track provenance, which is essential for verifying computations. We demonstrate PROV-CRT, a provenance module in a container runtime that tracks the provenance of computations during container creation and uses audited provenance to compare computations during container replay. We show how this module simplifies and improves the efficiency of complex container management tasks, such as classifying container contents and incrementally replaying containerized applications.

1 Introduction

Containers are light-weight alternatives to virtual machines; They are increasingly used for sharing and deploying applications, and thus facilitate the conduct of reproducible science. Container engines, such as Linux Containers (LXC) [5], Docker [2] and Singularity [6], use namespace primitives within the Linux kernel to execute an application in isolation on a host machine, and encapsulate the application's data and all its system dependencies. When a container is shared and deployed on a target machine it can be re-executed in isolation using encapsulated data and dependencies without the target environment interfering with its execution.

Using a container to isolate and port a computation is a necessary, albeit, only the first step toward the conduct of reproducible science. Current containers do not present any further guidance required for conducting reproducible science, such as verify if the results of repeated computations match (or do not match) with results obtained from the original execution of the computation on the host machine. To facilitate such guidance, we have previously proposed auditing and managing provenance as part of application virtualization [9] [8] [7].

In this demonstration, we showcase PROV-CRT, a provenance module within a container runtime that audits provenance during container creation time and uses the audited provenance to (i) compare and validate container replay, and (ii) classify container contents. To support these features efficiently, PROV-CRT associates a reference execution with every application that is to be containerized. This reference execution forms the basis for and classification of container contents and comparison with future container re-executions and container modifications.

PROV-CRT audits provenance at the granularity of system calls. The audited provenance is expressed in W3C format [10], where in each entity is a file that the system call operates on, each activity is a process that issues the system call, and the agent is the user containerizing the application. The audited provenance is application-independent and medium-fidelity that enables reasoning about process namespaces and ensures that the container application continues to remain isolated. Internally, PROV-CRT maintains a database of application executions and distinguishes between a reference execution and a re-execution of an application that is either a repetition or a modification. A repetition is validated against the original application execution by comparing provenance audited during the two executions. More details about the provenance auditing mechanisms are present here [11].

Our demonstration will unfold as follows: We plug PROV-CRT into Sciunit, our own container-runtime that is used for creating, isolating, and replaying containers. We leverage Sciunit's declarative interface for container management tasks. We will demonstrate how PROV-CRT provides re-execution guarantee and re-execution efficiency. Using the interface commands, it is possible to compare two executions based on its provenance, classify content, and incrementally repeat them.

2 Use cases and Setup

We describe the scenario and setup of the demonstration.

Use cases. We use two types of applications: (i) an instructional use case that demonstrates the module at an intuitive

[*]Contact author, [#]: Work performed during an internship

feature level, and (ii) a real use case comprising of artifacts submitted for evaluation to a Systems and Machine Learning (SysML) conference and stamped reusable from the ACM reproducibility initiative (Artifact #1 in [4]). Both use cases will demonstrate breadth and generality of PROV-CRT on multiple types of computational studies and their experiments. We use the second use case to demonstrate container classification and incremental replay in a real setting.

Setup. We use the Ubuntu and CentOS distribution of the Linux kernel. For this demonstration, we use machines hosted on the cloud with Sciunit container runtime installed. The runtime offers both a command-line client and a Python-based API, downloadable from [1]. Consider a user application, such as the Artifact #1 in [4], on a cloud machine. Each parametric execution of this use case corresponds to an experiment.

A user starts the runtime with an empty project *PWDemo* as in:

» **sciunit open** *PWDemo*

This creates an empty directory within which all the container's content is maintained, including any associated provenance that is audited and maintained by PROV-CRT. The user containerizes an experiment by executing the application within the context of the Sciunit:

» **sciunit exec** *main.sh <params>*

in which *main.sh* refers to an entry or start command for running an experiment. Parameters of the experiment, *<params>*, may be specified on command-line. This command assigns an execution identifier, e_i, to the experiment within the PWDemo container. During the execution of this command, the Sciunit runtime checks in all the binary, data, configuration, and environment files corresponding to the experiment and PROV-CRT audits provenance of the execution. Both the contents and the provenance is stored and indexed in a de-duplicated storage [9].

In the current version of PROV-CRT, the audited provenance is at the level of files and processes, which is application-independent, high-fidelity provenance. The audited provenance is expressed in W3C format. More details about audited provenance are described in [11]. PROV-CRT associates audited provenance with the hash values of content stored in de-duplicated storage. This association keeps provenance in-sync with de-duplicated content.

The user performs a container replay by issuing the command:

» **sciunit repeat** e_i

in which e_i represents the execution identifier of the experiment that was previously audited. PROV-CRT informs if the repeat was similar to the reference execution. For this it performs a recursive hash-based comparison as described in [11]. In the current version of PROV-CRT, a repeat does not stop if a provenance-based verification fails; the user is only informed that the entire repeat does not correspond to the original referenced execution.

The Sciunit runtime allows parameter changes (input argument or data file) to an execution as new executions. For a given experiment, parameters are changed as:

» **sciunit given** *<params>* **repeat** e_i *%<pp>*

in which *<params>* are the changed parameters and *%<pp>* is their respective parameter position. By using the repeat command but modifying the parameter in a given position, allows the system to maintain related experiments (owing to parameter changes) as a collection.

Modifications to source code of an experiment result in entirely new experiments. A user can, however, modify files of only one experiment at a time. Modifications must be committed as in:

» **sciunit commit**

to commit the checked out experiment. Both the `given` and `commit` command re-executes the experiment to store the change and its new associated provenance. Our demonstration will show how the Sciunit container runtime supported with PROV-CRT provides strict version control on experiment executions and its provenance. The interface prevents uncontrolled changes and enables comparisons while maintaining container isolation.

Classifying Container Content. The declarative interface includes commands for classifying content in a container. Sciunit lists all audited experiments as:

» **sciunit list**

A given experiment details are obtained with:

» **sciunit show** e_2

which shows size and other metadata of an experiment. PROV-CRT uses provenance to further classify container contents by:

» **sciunit show – detail** e_2

The detailed show command creates a classified view of the file contents of e_2 based on read/write patterns in the provenance log. The view is classified into input, transient,

and output files, configuration files, and system dependencies. Figure 1 shows part of the view as install requirements generated from the provenance log of the real use case [4].

We will demonstrate that PROV-CRT distinguishes between system and user-listed dependencies, and for each system dependency documents the relevant package manager. For instance, if the execution trace specifies a path to the dependency *libcrypto.so.1.1* then the library *libssl* is mentioned. This is useful for generating a `README` or *requirements.txt* of required system information. Version information of dependency files is also generated in the classified view. This is particularly useful for scripts since their source code is audited. The resulting container, however, does not record source code versions, which we assume are typically maintained by the author in a version control repository, such as Git.

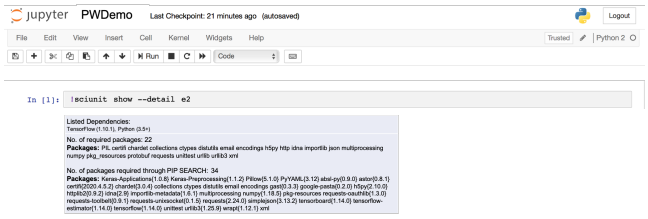

Figure 1: Provenance-based explanations from PROV-CRT

Incremental Replay of Containers. When the container is shared across compute nodes, a typical operation is to repeat the experiments in the container. The repeat operation in the interface re-starts the experiment process. We will demonstrate incremental repeat of an experiment. In particular, for our two use cases we will consider a Jupyter notebook format and show how PROV-CRT distinguishes notebook cells that have not changed from those that have. By using provenance, PROV-CRT bypasses the repeat operation for unchanged cells and replaces it with memoized output from a previous audit, and executes changed cells. Such an incremental repeat allows experiments to be branched. The incremental repetition of experiments is not available via PROV-CRT's command-line interface but is part of the Python API. The user must also install the PROV-CRT kernel within JupyterHub. Artifacts of this demonstration, *i.e.*, the generated containers for the two use cases, and the demonstration recording are available via [3].

Acknowledgements

This work is supported in part by NSF CNS-1846418, ICER-1639759 and a DOE Exascale Computing Project BSSw Fellowship.

References

[1] Sciunit. https://sciunit.run/, 2017. [Online; accessed 10-Sep-2017].

[2] Docker. https://www.docker.com/, 2019.

[3] Provenance week sciunit demonstration. https://bitbucket.org/depauldbgroup/pw20-sciunitdemo/, 2020.

[4] ACM. Systems and machine learning conference. https://ctuning.org/ae/artifacts.html#mlsys2019, 2019.

[5] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[6] Gregory M Kurtzer and *et. al.* Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5), 2017.

[7] Q. Pham, T. Malik, and I. Foster. Using provenance for repeatability. In *TaPP'13*, 2013.

[8] Q. Pham, T. Malik, B. Glavic, and I. Foster. LDV: Light-weight database virtualization. In *ICDE'15*, pages 1179–1190, 2015.

[9] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. Sciunits: Reusable research objects. In *IEEE eScience*, 2017.

[10] W3C. W3C PROV-DM: The PROV data model, 2013.

[11] Zhihao Yuan and et.al. Utilizing provenance in reusable research objects. *Informatics*, 5(1), 2018.