Explaining and Replaying Containers Using Provenance[*]

Raza Ahmad[#], Madeline Deeds, Yuta Nakamura, Naga Nithin Manne[*], Tanu Malik[#]
School of Computing, DePaul University, Chicago IL, USA
raza.ahmad, mdeeds, ynakamu1, nithin.manne, tanu.malik@depaul.edu
[#]: Contact authors, [*]: Work performed during an internship

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) [4], Docker [1] and Singularity [5], 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 recently proposed auditing provenance as part of container-like, sandbox systems [8]. Sciunit creates a container-like, sandbox by auditing an application's execution with *ptrace* and copying application contents. Sciunit creates a self-contained sandbox that includes the provenance audited during application execution. The sandbox, similar to a container, is portable across different environments. Unlike namespace containers [1,4,5], Sciunit uses provenance to verify if repeated results match (or do not match) within the sandbox. This match is performed only on encapsulated application files, and not by isolating the application's execution from other processes. Sciunit works in user space and does not require any compiler-level application instrumentation to determine dependencies [6]. Encapsulated files within a Sciunit are stored in a lightweight de-duplication storage system [7]. Runs of an application that incrementally differ from each other, are stored in a deduplicated storage thus significantly reducing the container size.

In this demonstration we showcase Sciunit's declarative interface comprising of commands that create portable containers, verifiably repeat them, and explain their contents using provenance logs. We will demonstrate how the interface controls all changes to the content and thus guarantees execution repeatability. Using the interface commands, users can compare two executions based on its provenance and content, and incrementally repeat them. In case a provenance-based repeatability analysis fails, Sciunit informs the user of the files from where the experiment started diverging.

2 Use cases and Setup

We describe the scenario and setup of the demonstration.

Use cases. We exemplify Sciunit on two types of use cases: (i) an instructional use case that demonstrates the interface at a feature and an intuitive 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 [3]). Both use cases will demonstrate breadth and generality of Sciunit on multiple types of computational studies and their experiments. The second use case will further illustrate explaining containers 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 installed. Consider a user application, such as Artifact #1 in [3], on a cloud machine. Each parametric execution of this use case corresponds to an experiment.

Declarative Interface. Sciunit offers both a command-line client and a Python-based API. A user starts a *PWDemo sciunit* as in:

» **sciunit open** *PWDemo*

Within this open sciunit the user encaspulates an experiment and its provenance with:

» **sciunit exec** *main.sh <params>*

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

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 sciunit. This command also checks in all the binary, data, configuration, and environment files corresponding to the experiment along with its provenance in a de-duplicated storage [7]. The user can perform a provenance-based replay [8] by:

» **sciunit repeat** e_i

in which e_i represents the execution identifier of the experiment to be repeated. In the current version of Sciunit, a repeat does not fail if a provenance-based verification fails; the user is only informed that the repeat does not correspond to original execution.

The Sciunit interface considers 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 same repeat command to modify the parameter in a given position, allows the system to maintain related experiments (owing to parameter changes) together. 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. It also executes the experiment to store the change and its associated provenance.

Our demonstration will show how the declarative Sciunit command-line interface provides strict version control on experiment executions and its provenance. The interface prevents uncontrolled changes and enables comparisons and incremental replay.

Explaining sciunits. The declarative interface includes commands for explaining an experiment in a container. Sciunit lists all audited experiments as:

» **sciunit list**

A given experiment details are obtained with:

» **sciunit show** e_2

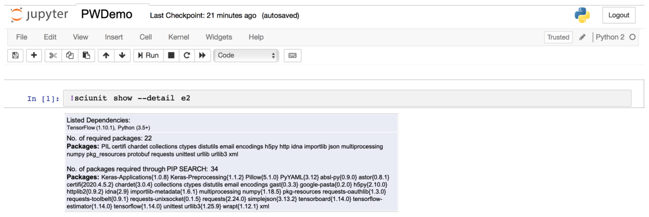
which shows size and other metadata of an experiment. Sciunit uses provenance to further explain container contents by:

» **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 [3].

We will demonstrate that Sciunit 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 *libcrypto.so.1.1* then the system dependency *libssl* is mentioned. This can be useful for generating a `README` or *requirements.txt* of required system information. Version information of files is also generated in the classified view. This is particularly useful for scripts since their source code is audited. A Sciunit package, however, does not record source code versions, which we assume are typically maintained by the author in a version control repository, such as Git.

Repeating Experiments Incrementally. When a sciunit is shared across compute nodes, a typical operation is to repeat the experiments in the container. The repeat operation in the Sciunit 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 Sciunit distinguishes notebook cells that have not changed from those that have. By using provenance, Sciunit 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 Sciunit's command-line interface but is part of the Python API. The user must also install the Sciunit kernel within JupyterHub.



```
In [1]: !sciunit show --detail e2
```

```
Listed Dependencies:
TensorFlow (1.10.1), Python (3.5+)
No. of required packages: 22
Packages: PIL certifi chardet collections ctypes distutils email encodings h5py http idna importlib json multiprocessing
numpy pkg_resources protobuf requests unittest urllib urllib3 xml

No. of packages required through PIP SEARCH: 34
Packages: Keras-Applications{1.0.8} Keras-Preprocessing{1.1.2} Pillow{5.1.0} PyYAML{3.12} absl-py{0.9.0} astor{0.8.1}
certifi{2020.4.5.2} chardet{3.0.4} collections ctypes distutils email encodings gast{0.3.3} google-pasta{0.2.0} h5py{2.10.0}
httplib2{0.9.2} idna{2.9} importlib-metadata{1.6.1} multiprocessing numpy{1.18.5} pkg-resources requests-oauthlib{1.3.0}
requests-toolbelt{0.9.1} requests-unixsocket{0.1.5} requests{2.24.0} simplejson{3.13.2} tensorboard{1.14.0} tensorflow-
estimator{1.14.0} tensorflow{1.14.0} unittest urllib3{1.25.9} wrapt{1.12.1} xml
```

Figure 1: Provenance-based explanations of sciunit

Artifacts of this demonstration, *i.e.*, the generated sciunits for the two use cases, and the demonstration recording are available via [2].

References

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

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

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

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

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

[6] Gregory Malecha and *et.al.* Automated software winnowing. In *ACM Symposium on Applied Computing*, 2015.

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

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