Auditing and Maintaining Provenance in Software Packages

Quan Pham[1](✉), Tanu Malik[2], and Ian Foster[1,2]

[1] Department of Computer Science, The University of Chicago, Chicago, IL 60637, USA
quanpt@cs.uchicago.edu

[2] Computation Institute, The University of Chicago, Chicago, IL 60637, USA
tanum@ci.uchicago.edu

Abstract. Science projects are increasingly investing in computational reproducibility. Constructing software pipelines to demonstrate reproducibility is also becoming increasingly common. To aid the process of constructing pipelines, science project members often adopt reproducible methods and tools. One such tool is CDE, which is a software packaging tool that encapsulates source code, datasets and environments. However, CDE does not include information about origins of dependencies. Consequently when multiple CDE packages are combined and merged to create a software pipeline, several issues arise requiring an author to manually verify compatibility of distributions, environment variables, software dependencies and compiler options. In this work, we propose software provenance to be included as part of CDE so that resulting provenance-included CDE packages can be easily used for creating software pipelines. We describe provenance attributes that must be included and how they can be efficiently stored in a light-weight CDE package. Furthermore, we show how a provenance in a package can be used for creating software pipelines and maintained as new packages are created. We experimentally evaluate the overhead of auditing and maintaining provenance and compare with heavy weight approaches for reproducibility such as virtualization. Our experiments indicate minimal overheads.

Keywords: Reproducibility · Software packaging tools · Software provenance · Tools and methods

1 Introduction

Computational reproducibility is a challenge, yet crucial for science. To meet the challenge, large-scale science projects are increasingly adhering to reproducibility guidelines. For instance, software associated with a publication is made available for download (see Figshare [20], RunMyCode [21], and Research Compendia [19]); but increasingly many science projects are making end-to-end software pipelines available. These pipelines are often for the larger scientific community, as in the case of Bio-Linux 5.0 [15], which is a bioinformatics virtual machine

B. Ludäscher and B. Plale (Eds.): IPAW 2014, LNCS 8628, pp. 97–109, 2015.
DOI: 10.1007/978-3-319-16462-5_8

that provides access to several pipelines for conducting next-generation sequence analysis, or sometimes to demonstrate project impacts as in the case of Swift Appliance [3], a virtual machine, which demonstrates crop simulation models using workflow systems.

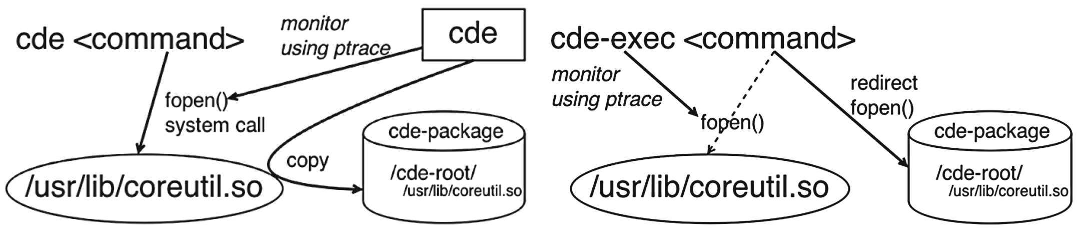
To help projects adhere to these reproducibility guidelines, project members often adopt best practices and tools for developing and maintaining software so that their contributed software quickly becomes part of a pipeline. In this paper, we focus on software packaging tools. We describe how auditing and maintaining *software provenance* as part of a packaging tool can significantly help in building and deploying software pipelines. In particular, provenance can be helpful in cutting down manual effort involved in ensuring software compatibility, thus leading to improved administration of software pipelines.

A software pipeline consists of many individual software modules. Given the collaborative nature of science, it is not uncommon for modules to develop independently. Furthermore, a module itself may depend upon externally-developed libraries, which evolve independently. To ensure library compatibility, and avoid what is often called "dependency hell", a software module is often packaged together with specific versions of libraries that are known to work with it. In this way, contributing project members can ensure that their module will run on any target system regardless of the particular versions of library components that the target system might already have installed.

However, packaging software modules with associated dependencies, but without clearly identifying the origin of the dependencies, gives rise to a number of provenance-related questions, especially when constructing software pipelines. For instance, determining the environment under which a dependency was built or other dependencies which must be present for using a module, are questions that must be answered when combining packages for creating software pipelines. Similarly, if a new software package is released, then through dependency analysis it will be useful to know which packages of a pipeline can use it. If a new version of a library is released that contains security fixes, then it will be useful to know which pipelines or packages are vulnerable.

To answer such questions, we must be able to capture and determine the provenance of a software entity, i.e., capture and determine where it came from. Current package management systems do not provide a means to audit or maintain software provenance within it. We use CDE, a software packaging tool that creates a source code and data package while identifying all static and dynamic software dependencies. CDE has also been successfully shown to create software packages out of many development environments. Though CDE packages static and dynamic dependencies for an application, it does not store associated provenance.

The first contribution of this paper is to enhance CDE to include software provenance, i.e., provenance of shared libraries and binaries on which a program depends. We call this enhanced CDE as CDE-SP. We describe tools and methods to audit, store, and query this provenance in CDE-SP. We then describe a science project use case in which software reproducibility is a concern. Our second contribution is to show how provenance, audited and stored as part of



(a) Audit mode (b) Execution mode

Fig. 1. CDE audit and execution modes

a CDE-SP package, can help in creating software pipelines for this use case. Finally, we show how provenance can be maintained as new packages are built during construction of software pipelines.

The remainder of the paper is structured as follows: We describe CDE, a software packaging tool that can identify and package program dependencies, in Sect. 2. Currently, CDE does not audit provenance of the program dependencies that it determines. In Sect. 3, we describe provenance that can be audited, stored, and queried in CDE-SP, resulting in a provenance-included package. In Sect. 4 we describe a science use case where provenance, included as part of software packages, can help in creating pipelines. In Sect. 5 we further enhance CDE-SP to enable it to maintain correct provenance as new packages are created. In Sect. 6, we conduct a thorough experimental evaluation to measure the overheads associated with auditing and maintaining provenance. Section 7 provides an overview of the related work in this area. We conclude in Sect. 8.

2 CDE: A Software Packaging Tool

The CDE tool [12,13] aims to easily create a package on a source resource and execute a program in that package on a target resource without any installation, configuration, or privilege permissions. It runs in two main modes: audit mode to create a CDE package, and execution mode to execute a program in a CDE package.

In audit mode (Fig. 1a), CDE uses the UNIX *ptrace* system call interposition to identify the code used by a running application (e.g., program binaries, libraries, scripts, data files, and environment variables), which it then records and combines to create a package. For example, when a process accesses a file or a library using the system call *fopen()*, CDE intercepts that syscall, extracts the file path parameter from the call, and makes a copy of the accessed file into a package directory, rooted at *cde-root* and consisting of all sub-directories and symbolic links of the original file's location.

The resulting package can be redistributed and run on another target machine, provided that the other machine has the same architecture (e.g. x86). The original CDE as available through [12,13] was limited to major Linux kernel versions (e.g. 2.6.x), but we have removed that restriction by adapting it for the newly released Linux kernel 3.0 as well as for Mac OS X by using the specification described here [2].

In execution mode (Fig. 1b), while executing a process from a package, CDE also monitors that process via *ptrace*. Each file system call is interrupted and its path argument is redirected to refer to the corresponding path of that file within the root directory of the CDE package on the target resource. In essence, CDE provides a lightweight virtualization environment to its running processes by providing the *cde-root* directory as a sandbox in a *chroot* operation. Redirecting all library dependency requests into this sandbox, CDE fools the target program into believing that it is executing on the original source machine [12]. It is to be noted that CDE binary only captures a single execution path, which is the execution path taken during run-time. If different execution paths need different types of dependencies, some dependencies may be left out. However, CDE does provide external scripts in its source code to find additional dependencies from strings inside binaries and libraries of captured packages.

3 CDE-SP: Software Provenance in CDE

The objective of auditing provenance is to capture additional details of the creation and origins of a library or a binary, such as the version of the compiler, the compilation options used, the exact set of libraries used for linking. This information must be gathered on a per environment basis so that it becomes easy to compile and create software pipelines.

Audit. CDE's audit feature identifies static and dynamic program dependencies. We instrument this feature to first determine a dependency tree, and then use UNIX utilities to store additional provenance information about each dependency. To create a dependency tree, process system calls are monitored that audit process name, owner, group, parent, host, creation time, command line, environment variables and the process binary's path. Whenever a process executes a file system call, a dependency of that process is recorded. In general, this dependency can be a data file or a shared library. We identify shared libraries using standard extensions, such as .so for system libraries and .jar for Java libraries, and create a dependency tree based on these libraries. Information about binaries and required shared libraries, such as version number, released version of shared libraries, and associated kernel distribution, is audited using UNIX commands *file, ldd, strings*, and *objdump*. By including these commands, we can obtain other static and dynamic dependencies, some of which are not audited by CDE during run-time. This set of commands is a more comprehensive way of obtaining dependencies comparing to CDE's external scripts. Current operating system distribution and user information is recorded from command *uname -a* and function *getpwuid(getuid())*.

Storage. Each package can store captured provenance to a relational database. Since this provenance will be useful for whatever target resource package is being used, we believe it is best to store this provenance within the package itself. We use LevelDB, a very fast and light-weight key-value storage library for storing provenance. To store provenance graphs that contain process-file and process-process edges, in a key-value store, we encode in the key the UNIX process identifier along with spawn time. The value is the file path or the process time. Table 1 describes the LevelDB schema for storing provenance graphs:

Table 1. LevelDB key-value pairs that store file and process provenance. Capital letter words are arguments.

Key	Value	Explanation
pid.PID1.exec.TIME	PID2	PID1 wasTriggeredBy PID2
pid.PID.[path, pwd, args]	VALUES	Other properties of PID
io.PID.action.IO.TIME	FILE(PATH)	PID wasGeneratedBy/wasUsedBy FILE(PATH)
meta.agent	USERNAME	User information
meta.machine	OSNAME	Operating system distribution

Query. LevelDB has a minimal API for querying. Instead of providing a rich provenance query interface, currently we implement a simple, light-weight query interface. The interface takes as input the program whose dependencies need to be retrieved. Using depth first search algorithm, a dependency tree in which the input program is the root is determined. The result is saved as a GraphViz file. Since the result may include multiple appearances of common files like those in */lib/, /usr/lib/, /usr/share/*, and */etc/* directories, the query interface also provides an exclusion option to remove uninteresting dependencies.

4 Using CDE-SP Packages to Create Software Pipelines

We describe a software pipeline through a use case. We then describe how CDE-SP packages can help to create the described software pipeline. The use case will also be used for experimental evaluation in Sect. 6.

4.1 Software Pipelines

Scientists with varying expertise at the Center for Robust Decision Making on Climate and Energy Policy (RDCEP) engage in open-source software development at their individual institutions, and rely primarily on Linux/Mac OS X environments. The Center often needs to merge its individual software modules to create software pipelines. We describe software modules being developed by three scientists, henceforth denoted as **A**lice, **B**ob, and **C**harlie, and the associated software pipeline that needs to be constructed.

– **A** measures and characterizes land usage and changes within it. She develops **data integration methods** to produce higher-resolution datasets depicting

inferred land use over time. To develop the needed methods, her software environment consists of R, geo-based R libraries (raster, ggplot2, xtable, etc.), and specific versions of Linux packages (r-base v2.15, libgdal v1.10, libproj v4.8).
- **B** develops **computational models** for climate change impact analysis. He conducts model-based comparative analysis, and his software environment consists of **A**'s software modules to produce high-resolution datasets, and other Linux packages, including C++, Java, AMPL [11] modeling toolkits and libraries.
- **C** uses **A** and **B**'s software modules within **data-intensive computing methods** to run them in parallel. **C**'s scientific focus is the efficiency of distributed computing methods and his software environment is primarily Java and Python and its libraries on Linux.
- For the **Center**, the goal of their combined collaboration is to predict future yields of staple agricultural commodities given changes in the climate; changes that are expected to drive, and be influenced by, changes in land usage [9]. The Center curator's environment is Mac OS X and a basic Unix shell.

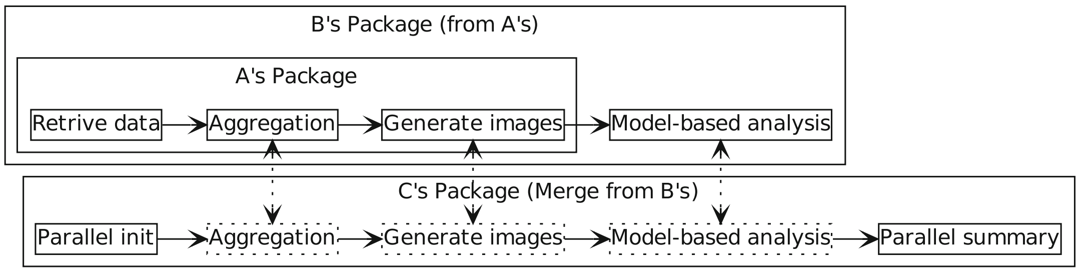


Fig. 2. Software packages of **A**, **B**, and **C**

Given the linear workflow of the science problem, it is often the case that **B** needs to rerun **A**'s software in his own environment. Instead of installing, this can simply be achieved if **A** shares a CDE package with **B**. However, if **B** attempts to create a software pipeline that includes **A**'s package and her software modules, then he needs to verify the provenance of each dependency included in **A** and her software. This is because a dependency with the same file path, but built on different Linux distributions (therefore different content), will conflict. In fact, if B creates a CDE package corresponding to this pipeline, one of the dependencies will be overwritten in the newly created package. By using the provenance-enabled CDE packages, which store md5 checksums of dependencies, such origins can be immediately verified, without manually tracking kernel distributions on which the dependency was built or communicating with the author of the software. Similarly, by checking versions of all dependencies within the package, **B** can document the compatibility of the newly created software pipeline.

As the use case demonstrates, **C** needs to use **A**'s and **B**'s packages, and the problem of dependency tracking, i.e., determining distributions and versions, given several dependencies and software environments, can increase significantly. In the Appendix we describe the magnitude of the dependency tracking problem if software development is undertaken in cloud-based environments.

5 Merging Provenance in CDE-SP

While provenance-included packages can eliminate much of the manual and tedious efforts of ensuring software compatibility, the downside is that provenance stores within a package need to be effectively maintained as software pipelines are themselves cast into new packages. Consider the Center's need for creating a software pipeline that satisfies reproducibility guidelines. To help the Center build this software pipeline, assume **A**, **B**, and **C** share their individual provenance-included packages. By exploring A, B and C's package provenance, the Center can examine all data and control dependencies among the contributing packages. The Center can then define a new experiment with steps using data and control dependencies from the three contributed packages, and create a new software package of this experiment. In particular, correct pathnames, attribution, etc., will need to be verified. We next describe how CDE-SP, with a $-m$ option, can be used to merge provenance from contributing packages.

In the typical CDE audit phase, file system binaries and libraries found in the path of program execution are copied to the *cde-root* directory. However, provenance may indicate two dependencies with the same path but emerging from different distributions or versions. In CDE-SP, these two files are stored in separate directories identified by a UUID, which is unique to the machine on which CDE-SP is executed. The UUID is the hash of the Mac address and the operating system. By creating this separate directory based on a UUID, files with the same paths but different origins can be maintained separately. Note that only files with differing content but the same path are maintained in separate UUID directories. Files with different paths can all still be in the same generic *cde-root* folder. We also include versioning of UUID directories so that they are copied and maintained correctly in new packages.

Because provenance informs that separate UUID based directories be created within a CDE-SP package, correspondingly, the modifications are needed in the LevelDB provenance store and the CDE-SP redirection mechanism. The LevelDB path in the value field needs to reflect the UUID directory where the dependency exists. The CDE redirection, which redirects all system calls to the *cde-root* directory, in CDE-SP needs to redirect to the appropriate UUID directory. This redirection can be tricky since it needs to know where the process is running. To enable correct redirection, CDE-SP with merge maintains a *current_root_id* pointer for each tracing process. This bookkeeping pointer helps in redirecting to the package root directory of the pointer in case the process forks other processes. Alternatively, if the process performs an *execve()* system call, or accesses a file, or changes directories, absolute paths are read and checked to determine if redirection is necessary.

Another issue when merging two packages is maintaining licensing information. While general licensing issues are outside the scope of this paper, the current CDE-SP maintains authorship of software modules during the merge process. When two packages are merged in their entirety, the authorship of a new package is the combined authorship of the contributing packages. However, when part of a contributing package is used to create a new package, then authorship must be validated from the provenance stored in the original package. To validate, CDE-SP generates the subgraph associated with the part of the package, and, using subgraph isomorphism, validates that it is indeed part of the original provenance graph.

The subgraph isomorphism (or matching) problem is NP-complete [22] leading to an exponential time algorithm. In our case, we compare file paths and names to determine if two provenance graphs are subgraph-isomorphic. In our implementation of VF2 subgraph-isomorphism algorithm [6], we reduce computation time by only matching provenance nodes of processes with the same path to their binary and working directory, and only matching provenance nodes of files with the same path. We believe that this implementation is sufficient for validating provenance subgraph isomorphism among lightweight packaging tools.

6 Experiment and Evaluation

The benefits of reproducibility can be hard to measure. In this Section, we describe the three experiments we conducted to determine the overall performance of CDE-SP.

1. We determined the performance of CDE-SP in: auditing performance overhead, disk storage increase, and provenance query runtime;
2. We determined the redirection overhead if multiple UUID-based directories are created in CDE-SP; and
3. We compared the lightweight virtualization approach of CDE-SP with Kameleon [10], a heavyweight virtualization approach used for reproducibility.

All experiments in this section are tested on an Ubuntu 12.04.3 LTS workstation with an 8 GBs RAM and 8-core Intel(R) processor clocking at 1600 MHz.

6.1 Audit Performance and Size Overhead in CDE-SP

In Table 2, we record execution times and disk usage of CDE and CDE-SP in auditing a software pipeline mentioned in Sect. 4.1. Both CDE-SP and CDE are set up for a pipeline with two applications: *Aggregation* and *Generate Image*. Each is repeated 10 times. The result shows approximately a 2.1 % slowdown of CDE-SP in comparison with CDE due to provenance capture. The result fits with our observation that the overhead is from *ptrace* which both CDE and CDE-SP rely on heavily to implement their capture capabilities. Additional functions that store provenance record to LevelDB database introduce negligible provenance capture overhead compared to 0–30% CDE virtualization overhead [12]. In this setup, CDE package uses 732 MB; while CDE-SP, in addition to the

Table 2. Increase in CDE-SP performance is negligible in comparison with CDE

	Create package	Execution	Disk usage	Provenance query
CDE	852.6 ± 2.4 (s)	568.8 ± 2.4 (s)	732 MB	
CDE-SP	870.5 ± 2.5 (s)	569.5 ± 1.8 (s)	732 MB + 236 kB	0.4 ± 0.03 (s)

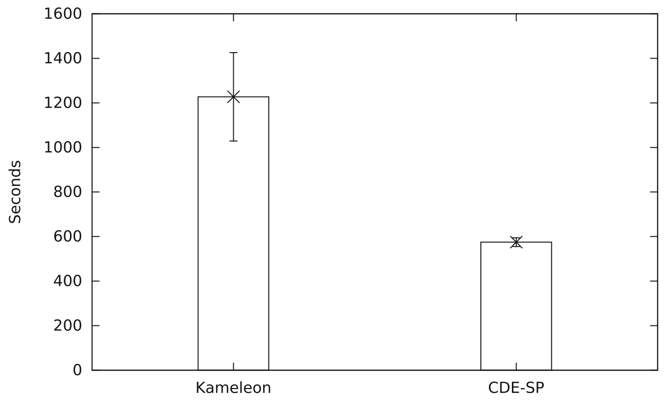


Fig. 3. Overhead when using CDE with Kameleon VM appliance

software package, creates a LevelDB database of size 236 kB (0.03 % increase) that contains approximately 12,000 key-value pairs.

To measure provenance query performance, we created a Python script to query the audited LevelDB provenance database and create a provenance graph of the experiment with common shared libraries filtered out. The Python script reads through approximately the 12,000 key-value pairs in 0.39 s to create a GraphViz script that can be converted to image or visualized later.

6.2 Redirection Overhead in CDE-SP

We also compared an execution of CDE package and CDE-SP package to measure the redirection overhead of CDE-SP. Using the packages created by the above experiment with two applications, *Aggregation* and *Generate Image*, we pipelined output of *Aggregation* to input of *Generate Image*, which requires CDE-SP to apply redirection among multiple CDE roots. The experiment showed 3 data files, as outputs of *Aggregation* package, were moved to *Generate Image* package. After the data was moved to the next package, the experiment was executed the same as in CDE. The result shows less than a 1 % slowdown of CDE-SP, which maybe due to initial loading of library dependencies in *Generate Image* package.

6.3 CDE-SP Vs Kameleon

In this experiment, we used the Kameleon engine to make a bare bone VM appliance that contains the content of a CDE-SP package corresponding to the software pipeline described in the use case (Sect. 4.1). The package content was copied directly to the root file system of the VM appliance. In terms of user software, the new VM appliance is close to a replica of the package, without any redundant installed software. We compared the two approaches qualitatively and quantitatively.

Qualitatively, the overhead of instantiating a VM is significant as compared to creating a CDE-SP package. In particular, for CDE-SP the user needs to specify input packages, and using one command, the author can create a new software package. Kameleon is user friendly and can create virtual machine appliances in different formats for different Linux distributions. But, users must provide self-written YAML-formatted recipes or self-written macrosteps and microsteps to generate customized virtual images. Based on the recipe input, it generates bash scripts to create an initial virtual image of a Linux distribution, and populates the initial image with more Linux packages to produce needed appliances.

Quantitatively, we compared the time for executing the software pipeline within a CDE-SP package with time for execution within a VM. Note that we do not compare time for initializing, since time for writing YAML scripts cannot be measured in the case of Kameleon. During the execution, CDE-SP redirected 2717 file-read system calls, 10 file-write system calls, 17 file-read-write system calls. Figure 3 shows that the Kameleon VM appliance slowed down the experiment significantly: approximately 200 % or more. This heavyweight VM overhead is substantial in comparison with the CDE-SP lightweight approach.

7 Related Work

Details about software have been included in provenance collected within workflow systems. For instance, Research Objects [4], packages scientific workflows with auxiliary information about workflows, including provenance information and metadata, such as the authors, the version. Our focus here is not limited to any specific workflow system.

Software packaging tools such as CDE [12,13] and Sumatra [8] can capture an execution environment in a lightweight fashion. Sumatra captures the environment at the programming level (Python), while CDE operates at the operating system level, and is thus more generic. Even at the system level, different tracing mechanisms can be used. At the user-space level, *ptrace* [1] is a common mechanism, whereas at the kernel-level, use of SystemTap [18] is more common. SystemTap, being kernel-based, has better performance compared to *ptrace* since it avoids context switching between the tracee (which is in the kernel) and the tracer (which is user space) [14]. However, from a reproducibility standpoint, SystemTap needs to run at a higher privilege level, i.e., it requires root access, creating a more restricted environment.

Virtual machine images (VMIs) provide a means of capturing the environment in a form that permits later replay of a computation. Kameleon [10] uses a bash script generator to create virtual images from scratch for any Linux distributions. Using recipes, users can generate customized virtual images with predefined software packages to run on different cloud computing service providers. We have compared our approach with creating VMIs for reproducibility.

Tools such as Provenance-to-Use (PTU) [17] and ReproZip [5] have demonstrated the advantages of including provenance in self-contained software packages. Currently, these tools include execution provenance and not software provenance. Finally, software provenance is an emerging area that uses Bertillonage metrics for finding software entities in large code repositories [7]. In this paper, we have described how software provenance can help in building packages that can satisfy reproducibility guidelines.

8 Conclusion

CDE is a software packaging tool that helps to encapsulate static and dynamic dependencies and environments associated with an application. However, CDE does not encapsulate provenance of the associated dependencies such as their build, version, compiler, and distribution. The lack of information about the origins of dependencies in a software package creates issues when constructing software pipelines from packages. In this paper, we have introduced CDE-SP, which can include software provenance as part of a software package. We have demonstrated how this provenance information can be used to build software pipelines. Finally, we have described how the CDE-SP can maintain provenance when used to construct software pipelines.

Acknowledgments. The authors would like to thank the following participants in the RDCEP Center, in particular, Neil Best, Joshua Elliott and Justin Wozniak at The University of Chicago, Columbia University, and Argonne National Laboratory for motivating our use case, and Allison Brizius for describing the Center's activities. This work is supported by NSF grant SES-0951576 and subcontract award under grant GEO-1343816.

Appendix

In our use case, **A**, **B**, and **C** develop open-source code and use publicly-available datasets. Their specified software environments, which may appear different, can be still overlapping. To demonstrate the magnitude of overlap, we assume that each developer uses the cloud for their research, which is not uncommon in today's projects, and chooses a different Linux distribution. Differences in the choice of linux distributions is also not surprising as the Linux Counter Distributions Report [16] indicates that there is no clean winner in terms of usage of Linux distributions, with no one distribution accounting for more than 30 %. Further, we limit software environments to refer to application binaries and libraries that are often overlapping and create conflicts.

If the two assumptions are sound, then the overlap in the environment, i.e., files which have the same path, but differing content, can be as high as 18 %. We calculate this by taking five Linux distributions with similar setup available on Amazon EC2. For each pair of machines, we calculate the number of files with the same path on two machines, and the number of files with the same path on two machines but having different md5 checksum. Table 3 shows that between any two machines, on average, 6.8 % of files have the same path but differ in content. In other words, these files are not interchangeable but depend on the underlying operating system.

Table 3. Ratio of different files having the same path in 5 popular AMIs. The denominator is number of files having the same path in two distributions, and the numerator is the number of files with the same path but different md5 checksum. Ommited are manual pages in */usr/share/* directory.

	RH	SUSE	U12	U13
Amz	5498/23 k	3184/11 k	1203/5.4 k	1819/5.5 k
RH		3861/12 k	1654/6.6 k	2223/6.3 k
SUSE			1245/3.9 k	2085/6.4 k
U12				8226/24 k

References

1. *ptrace*(2) - Linux man page. http://linux.die.net/man/2/ptrace
2. Replacing *ptrace()*. http://uninformed.org/index.cgi?v=4&a=3&p=14
3. Swift appliance at science clouds. http://scienceclouds.org/appliances/swift-appliance/
4. Belhajjame, K., Corcho, O., et al.: Workflow-centric research objects: First class citizens in scholarly discourse. In: Proceedings of Workshop on the Semantic Publishing (SePublica), Crete, Greece (2012)
5. Chirigati, F., Shasha, D., Freire, J.: ReproZip: using provenance to support computational reproducibility. In: USENIX Workshop on the Theory and Practice of Provenance, TaPP 2013 (2013)
6. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
7. Davies, J., German, D.M., Godfrey, M.W., Hindle, A.: Software bertillonage. Empirical Softw. Engg. **18**(6), 1125–1155 (2013)
8. Davison, A.P.: Automated capture of experiment context for easier reproducibility in computational research. Comput. Sci. Eng. **14**, 48–56 (2012)
9. Elliott, J., et al.: Constraints and potentials of future irrigation water availability on agricultural production under climate change. In: Proceedings of the National Academy of Sciences (2013)
10. Emeras, J., Richard, O., Bzeznik, B.: Reconstructing the software environment of an experiment with kameleon (2011)
11. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Mathematical Programming Language. ATT Bell Laboratories, Murray Hill (1987)
12. Guo, P.: CDE: Run any linux application on-demand without installation. Technical. report, USENIX Association, Boston, Massachusetts (2011)
13. Guo, P.J., Engler, D.: CDE: Using System Call Interposition to Automatically Create Portable Software Packages. USENIX Association, Portland (2011)

14. Keniston, J., Mavinakayanahalli, A., Panchamukhi, P., Prasad, V.: Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In: Linux Symposium (2007)
15. Krampis, K., et al.: Cloud BioLinux: pre-configured and on-demand bioinformatics computing for the genomics community. BMC Bioinf. **13**(1), 42 (2012)
16. Löhner, A.: Lico-Project information (2012)
17. Pham, Q., Malik, T., Foster, I.: Using provenance for repeatability. In: USENIX Workshop on the Theory and Practice of Provenance (2013)
18. Prasad, V., Cohen, W., Eigler, F., Hunt, M., Keniston, J., Chen, B.: Locating system problems using dynamic instrumentation (2005)
19. Seiler, J.: Research compendia: Connecting computation to publication (2013)
20. Singh, J.: FigShare. J. Pharmacol. Pharmacotherapeutics **2**(2), 138–139 (2011)
21. Stodden, V., Hurlin, C., Perignon, C.: RunMyCode.Org: a novel dissemination and collaboration platform for executing published computational results (2012)
22. Wegener, I.: Complexity Theory Exploring the Limits of Efficient Algorithms. Springer, Berlin (2005)