A benchmark suite and performance analysis of user-space provenance collectors

Samuel Grayson
grayson5@illinois.edu
University of Illinois Urbana-Champaign
Department of Computer Science
Urbana, IL, USA

Faustino Aguilar
faustino.aguilar@up.ac.pa
University of Panama
Department of Computer Engineering
Panama City, Panama

Reed Milewicz
rmilewi@sandia.gov
Sandia National Laboratories
Software Engineering and Research Department
Albuquerque, NM, USA

Daniel S. Katz
d.katz@ieee.edu
University of Illinois Urbana-Champaign
NCSA & CS & ECE & iSchool
Urbana, IL, USA

Darko Marinov
marinov@illinois.edu
University of Illinois Urbana-Champaign
Department of Computer Science
Urbana, IL, USA

ABSTRACT

Computational provenance has many important applications, especially to reproducibility. System-level provenance collectors claim to be able to track provenance data without requiring the user to change anything about their application. However, system-level provenance collectors are not commonly used in computational science. This work aims to bring research in provenance collection closer to practice by evaluating prior work on a common benchmark subset and identifying gaps in prior work. This subset can be used as goalposts for future work on system-level provenance collectors.

ACM Reference Format:
Samuel Grayson, Faustino Aguilar, Reed Milewicz, Daniel S. Katz, and Darko Marinov. 2024. A benchmark suite and performance analysis of user-space provenance collectors. In . ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Within the computational science and engineering (CSE) community, there is a consensus that greater reproducibility is a pathway towards increased productivity and more impactful science [67]. In the past decade, this has inspired a diverse range of research and development efforts meant to give us greater control over our software, including containers and virtual machines to capture environments [14, 38, 65, 79], package managers for fine-grained management of dependencies [28, 47], interactive notebooks and workflows [11, 25, 46], and online platforms for archiving and sharing computational experiments [19, 30, 84, 85]. In this work, we focus on **computational provenance** as another complementary strategy for managing reproducibility across the research software lifecycle. Computational provenance is the history of a computational task, describing the artifacts and processes that led to or influenced the result [27]; the term encompasses a spectrum of tools and techniques ranging from simple logging to complex graphs decorated with sufficient detail to replay a computational experiment.

Provenance data can provide crucial information about the hardware and software environments in which a code is executed. The use cases for this data are numerous and many different tools for collecting it have been independently developed. However a rigorous comparison of those available tools and the extent to which they are practically usable in CSE application contexts has been lacking from prior work. To summarize the state of the art and to establish goalposts for future research in this area, our paper makes the following contributions:

- *A rapid review on available system-level provenance collectors.* We identify 45 provenance collectors from prior work, classify their method of operation, and attempt to reproduce the ones that meet specific criteria. We successfully reproduced 9 out of 15 collectors that met our criteria.
- *A benchmark suite for system-level provenance collectors*: Prior work does not use a consistent set of benchmarks; publications often use an overlapping set of benchmarks from their prior work. We find the superset of all benchmarks used in the prior work, identify unrepresented areas, and find a statistically valid subset of the benchmark. Our benchmark subset is able to recover the original benchmark results within 5% of the actual value 95% of the time.
- *We develop a predictive performance model for each provenance collector*: We use linear models for predicting the overhead of an application in a provenance collector based on the application's performance characteristics (e.g., number of file syscalls per second). These models can estimate hidden latencies in the system without directly observing them.

The remainder of the paper is structured as follows. In Section 2, we motivate the value of provenance and the pros/cons of

Conference'17, July 2017, Washington, DC, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

system-level provenance compared to application- and workflow-level provenance.

2 BACKGROUND

As one *Nature* editoralist put it, "behind every great scientific finding of the modern age, there is a computer' '[69]. The production of scientific results now often involve complex and lengthy operations on hardware and software systems; transparency is fundamental to the practice of science, and increasing the transparency of those processes is the end goal of provenance research. Provenance capture represents a wide spectrum of tools and techniques. A taxonomy by Regan et al. identifies five different kinds of provenance information: interaction (of user actions and commands on a system), data (of the transformation and movement of data), visualization (of the history of the representation of those results), insight (of the resulting hypotheses and analytic findings), and rationale (of the underlying reasoning and intentions behind running the software)[75].

In this paper, we focus our attention on system interaction and data provenance. A recent Department of Energy Advanced Scientific Computing Research report by Heroux et al. has called for further research to develop solutions for highly automatic and portable provenance capture and replay[**?**]. The potential applications for such tools are numerous, including the following noted by Pimentel et al. [72] and Sar and Cao [78]:

1. **Reproducibility**. A description of the inputs and processes used to generate a specific output can aid manual and automatic reproduction of that output[1]. Empirical studies [21, 32, 89, 96] show that reproducibility is rarely achieved in practice, probably due to its difficulty under the short time budget that scientists have available to spend on reproducibility. If reproducibility was easier to attain, perhaps because of automatic provenance tracking, it may improve the reproducibility rate of computational research. Provenance data improves **manual reproducibility**, because users have a record of the inputs, outputs, and processes used to create a computational artifact. Provenance data also has the potential to enable **automatic reproducibility**, if the process trace is detailed enough to be "re-executed". This idea is also called "software record/replay". Automatic reproduciblity opens itself up to other applications, like saving space by deleting results and regenerating them on-demand. However, not all provenance collectors make this their goal.
2. **Caching subsequent re-executions**. Computational science inquiries often involve changing some code and re-executing the workflows (e.g., testing different clustering algorithms). In these cases, the user has to keep track of what parts of the code they changed, and which processes have to be re-executed. However, an automated system could read the computational provenance graphs produced by previous executions, look at what parts of the code changed, and safely decide what processes need to be re-executed. The dependency graph would be automatically deduced, leaving less chance for dependency misspecification, unlike Make and CMake, which require the user to manually specify a dependency graph.
3. **Comprehension**. Provenance helps the user understand and document workflows and workflow results. An automated tool that consumes provenance can answer queries like "What version of the data did I use for this figure?" and "Does this workflow include FERPA-protected data?". A user might have run dozens of different versions of their workflow and may want to ask an automated system, "show me the results I previously computed based on that data with this algorithm?".

There are three high-level methods by which one can capture computational provenance: 1) by modifying an application to report provenance data, 2) by leveraging a workflow engine or programming language to report provenance data, and 3) by leveraging an operating system to report provenance data [27]. Application-level provenance is the most semantically rich but the least general since it only applies to particular applications modified to disclose provenance. Workflow- and language-level provenance is a middle ground between semantic richness and generality, applying to all programs using a certain workflow or programming language. System-level provenance is the least semantically rich but most general, applying to all programs on that particular system.

The implementation cost of adopting system-level provenance in a project that currently has no provenance is low because the user need not change *anything* about their application or workflow; they merely need to install some provenance collector onto their system and rerun their application. Although the user may eventually use a more semantically rich provenance, low-initial-cost system-level provenance would get provenance's "foot in the door". Since system-level provenance collection is a possibly valuable tradeoff between implementation cost and enabling provenance applications, system-level provenance will be the subject of this work.

In the context of system-level provenance, artifacts are usually files, processes, or strings of bytes. Operations are usually syscalls involving artifacts, e.g., `fork`, `exec`, `open`, `close`. For example, suppose a bash script runs a Python script that uses matplotlib to create a figure. A provenance collector may record the events in Figure 1, including all file dependencies of the process, without knowledge of the underlying program or programming language.

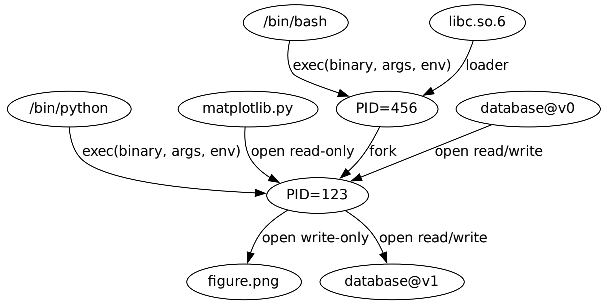
We defer to the cited papers for details on versioning artifacts [8] and cycles [62]. Some collectors may also record calls to network resources, the current time, process IPC, and other interactions.

While there is little additional programmer-time in using system-level provenance (no user code change), there is a non-trivial implicit overhead in monitoring and recording each computational process. Even a minor overhead per I/O operation would become significant when amplified over the tens of thousands of I/O operations a program might execute per second. Prior publications in system-level provenance usually contain benchmark programs to evaluate the overhead imposed by the system-level provenance tool. However, the set of chosen benchmark programs is inconsistent from one publication to another, and overhead can be sensitive to the exact choice of benchmark, so these results are incomparable between publications. Most publications only benchmark their new system against native/no-provenance, so prior work cannot easily establish which system-level provenance tool is the fastest.

[1]"Reproduction", in the ACM sense, where a **different team** uses the **same input artifacts** to generate the output artifact [6].

(1) The user created a process, call it PID=123.
(2) The process PID=123 executed bash.
(3) The loader of process PID=123 loaded libc.so.6.
(4) The process PID=123 forked a process, call it PID=456.
(5) The process PID=456 executed python.
(6) The process PID=456 read matplotlib.py (script library).
(7) The process PID=456 opened database for reading and writing, which creates a new version of the node in th provenance graph.
(8) The process PID=456 wrote figure.png.

(a) Abridged list of events.



(b) Abridged graph of events. The arrows point in the direction of dataflow. Other authors use other conventions for what they render as nodes, edges, and arrow direction.

Figure 1: An abridged list and graph of events a hypothetical system-level provenance collector would collect from a Bash script that invokes Python to plot some data. This collector could infer the required files (including executables, dynamic libraries, scripts, script libraries (e.g., matplotlib), data) *without* knowing anything about the program or programming language.

2.1 Prior work

Each result of our rapid review (Table 2) is an obvious prior work on provenance collection. However, those priors studies look at only one or two competing provenance tools at a time. To the best of our knowledge, there has been no global comparison of provenance tools. ProvBench [53] uses 3 provenance collectors (CamFlow, SPADE, and OPUS), but they are solely concerned with the differences between representations of provenance, not performance.

On the other hand, benchmark subsetting is a well-studied area. This work mostly follows Yi et al.'s publication [94], which evaluates subsetting methodologies and determines that dimensionality reduction and clustering ar broadly good strategies. Phansalkar et al. [71] apply dimensionality reduction and clustering to SPEC CPU benchmarks.

3 METHODS

3.1 Rapid Review

We preformed a rapid review to identify the research state-of-the-art tools for automatic system-level provenance.

Rapid Reviews are a lighter-weight alternative to systematic literature reviews with a focus on timely feedback for decision-making. Schünemann and Moja [80] show that Rapid Reviews can yield substantially similar results to a systematic literature review, albeit with less detail. Although developed in medicine, Cartaxo et al. show that Rapid Reviews are useful for informing software engineering design decisions [17, 18].

We conducted a rapid review with the following parameters:

- **Search terms**: "system-level AND provenance", "computational provenance"
- **Search engine**: Google Scholar
- **Number of results**: 50 of both searches. This threshold is the point of diminishing returns, as no new collectors came up in the 40th – 50th results.
- **Criteria**: A relevant publication would center on one or more operating system-level provenance collectors that capture file provenance. A tool requiring that the user use a specific application or platform would be irrelevant.

3.2 Benchmark Selection

Using the tools selected above, we identified all benchmarks that have been used in prior work. We excluded benchmarks for which we could not even find the original program (e.g., TextTransfer), benchmarks that were not available for Linux (e.g., Internet Explorer), benchmarks with a graphical component (e.g., Notepad++), and benchmarks with an interactive component (e.g., GNU Midnight Commander).

We implemented the benchmarks as packages for the Nix package manager[2], so they are runnable on many different platforms. Nix has official installers for Linux, Mac OS X, and Windows Subsystem for Linux on i686, x86_64, and aarch64 architectures, but FreeBSD and OpenBSD both package Nix themselves, and it can likely be built from source on even more platforms.

We also added new benchmarks:

- **Data science**: None of the benchmarks resembled a typical data science program, so we added the most popular Notebooks from Kaggle.com, a data science competition website. Data science is a good use-case for provenance collection because a user might have a complex data science workflow and want to know from what data a specific result derives and if a specific result used the latest version of that data and code.
- **Compilations**: Prior work uses compilation of ApacheHttpd of Linux. We added compilation of several other packages used in computational science to our benchmark. Compiling packages is a good use-case for a provenance collection because a user might trial-and-error multiple compile commands and not

[2]See https://nixos.org/guides/how-nix-works

Table 1: Our experimental machine description.

Name	Value
CPU	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
RAM	16 GiB of SODIMM DDR4 Synchronous 2400 MHz
Kernel	Linux 6.1.64
Disk	Sandisk Corp WD Black SN770 250GB NVMe SSD

remember the exact sequence of "correct" commands; the provenance tracker would be able to recall the commands that were not clobbered over, so the user can know what commands actually worked [16].

3.3 Performance Experiment

To get consistent measurements, we run a complete matrix (every collector on every benchmark) 3 times in a random order. Table 1 describes our experimental machine. We use BenchExec [12] to precisely measure the CPU time, wall time, memory utilization, and other attributes of the process (including child processes) in a Linux CGroup without networking, isolated from other processes. We disable ASLR, which does introduce non-determinism into the execution time, but it randomizes a variable that may otherwise have a confounding effect [63]. We restrict the program to a single core to eliminate unpredictable scheduling and prevent other daemons from perturbing the experiment (they can run on the other N-1 cores). We wrap the programs that exit quickly in loops so they take about 10 seconds without any provenance system, isolating the cold-start costs. While cold-start costs can be significant, if the total program execution time is small, the user may not notice even the highest overhead of provenance collectors.

3.4 Benchmark Subsetting

We implemented and ran many different benchmarks, which may be costly for future researchers seeking to evaluate new provenance collectors. A smaller, less costly set of benchmarks may sufficiently represent the larger set.

Following Yi et al. [94], we evaluate the benchmark subset in two different ways:

- **Accuracy**. How closely do features of the subset resemble features of the original set? We will evaluate this by computing the root mean squared error (RMSE) of a non-negative linear regression from the standardized features of selected benchmarks to the mean of features of the total set.
- **Representativeness.** How close are benchmarks in the original set to the closest benchmarks in the subset? We will evaluate this by computing RMSE on the euclidean distance of standardized features from each benchmark in the original set to the closest benchmark in the selected subset.

We use a non-negative linear regression to account for the possibility that the total set has unequal proportions of benchmark clusters. We require the weights to be non-negative, so doing better on each benchmark in the subset implies a better performance on the total. Finally, we normalize these weights by adding several copies of the following equation to the linear regression: $\text{weight}_A + \text{weight}_B + \cdots = 1$. Yi et al. [94] were attempting to subset SPEC CPU 2006, which one can assume would already be balanced in these terms, so their analysis uses an unweighted average.

We standardize the features by mapping x to $z_x = (x - \bar{x})/\sigma_x$. While x is meaningful in absolute units, z_x is meaningful in relative terms (i.e., a value of 1 means "1 standard deviation greater than the mean"). Yi et al., by contrast, only normalize their features $x_{\text{norm}} = x/x_{\text{max}}$, which does not take into account the mean value. We want our features to be measured relative to the spread of those features in prior work.

We score by RMSE over mean absolute error (MAE), used by Yi et al. [94], because RMSE punishes outliers more. MAE permits some distances to be large, so long it is made up for by shrinking other distances. RMSE would prefer a more equitable distribution, which might be worse on average but better on the outliers than MAE. We think this aligns more with the intent of "representativeness."

We will use features that are invariant between running a program ten times and running it once as features. These features give long benchmarks and short benchmarks which exercise the same functionality similar vectorization. In particular, we use:

1. The log overhead ratio of running the benchmark in each provenance collector. We use the logarithm of the ratio rather than the ratio directly because the ratio cannot be distributed symmetrically, but the logarithm may be. Suppose some provenance collector makes programs take roughly twice as long but with a large amount of variance, so the expected value of the ratio is 2. A symmetric distribution would require the probability of observing a ratio of -1 for a particular program is equal to the probability of observing a ratio of 5, but a ratio of -1 is impossible, while 5 is possible due to the large variance. On the other hand, $\log x$ maps positive numbers (like ratios) to real numbers (which may be symmetrically distributed); choosing 2x as our center, $e^{0.3} \approx 2$, $e^{0.7} \approx 5$ and $e^{-0.1} = 0.9$ are equidistant in log-space (negative logs indicate a speedup rather than slowdown, which are theoretically possible). Also note that exp(arithmean(log(x))) is the same as geomean(x), which is preferred over arithmean(x) for performance ratios according to Mashey [59].
2. The ratio of CPU time to wall time. When limited to a single core on an unloaded system, wall time includes I/O, but CPU time does not.
3. The number of syscalls in each category per wall time second, where the categories consist of socket-related, file-metadata-related, directory-related, file-related, exec-related, fork-related, exit-related syscalls, IPC-related syscalls, and chdir syscalls.

In order to choose the subset, we will try clustering (k-means and agglomerative clustering with Ward linkage[3]), preceded by optional dimensionality reduction by principal component analysis (PCA). Once the benchmarks are grouped into clusters, we identify one benchmark from each of the k clusters to consist the benchmark subset. We will determine the best k experimentally.

[3]k-means and agglomerative/Ward both minimize within-cluster variance, which is equivalent to minimizing our metric of "representativeness" defined earlier, although they minimize it in different ways: k-means minimizes by moving clusters laterally; Agglomerative/Ward minimizes by greedily joining clusters.

3.5 Performance Model

A related problem to subsetting is inferring a performance model, which would predict the approximate overhead of a workload under different provenance systems based on characteristics of the workload. Inferring a performance model may improve our understanding of the bottlenecks in provenance collectors.

A priori, provenance collectors put a "tax" on certain syscalls (e.g., file I/O operations, process forks, process execs), because the system has to intercept and record these. Therefore, we expect a low-dimensional linear model (perhaps number of I/O operations per second times a weight plus number of forks per second times another weight) would predict overhead optimally. To estimate this, we use non-negative LASSO regression. Non-negativity allows us to interpret the coefficients of the model as the slowdown imposed by specific syscalls. LASSO tries to find a sparse linear model, ignoring as many features as possible while maintaining good results. LASSO's tradeoff between ignoring features and getting better accuracy is determined by α. We will determine the optimal α through cross-validation. Outside of that cross-validation, we use a larger cross-validation to estimate the out-of-sample standard error of the predictor.

Unlike the previous section, we are not comparing provenance systems as a ratio to native; we are trying to find a dependency on the absolute number of system calls to an absolute amount of time (e.g., "every open costs 100 extra ns"). Therefore, we regress the difference between provenance and no-provenance wallclock time and use absolute (non-standardized) features. However, the scoring metric is still percent error because the runtimes of the programs in the benchmark may be different orders of magnitude.

4 RESULTS

4.1 Selected Provenance Collectors

Table 2 shows the provenance collectors we collected and their qualitative features. Because there are not many open-source provenance collectors in prior work, we also include the following tools, which are not necessarily provenance collectors, but may be adapted as such: strace, ltrace, fsatrace, and RR. See Appendix A for more in-depth description of notable provenance collectors. The second column shows the "collection method" (see Appendix B for their exact definition).

The last column in the table categorizes the "state" of that provenance collector in this work into one of the following:

- **Not for Linux.** Our systems are Linux-based and Linux is used by many computational scientists. Therefore, we did not try to reproduce systems that were not Linux based.
- **VMs too slow.** Some provenance collectors require running the code in a virtual machine. We know a priori that these methods are prohibitively slow, with Panorama reporting 20x average overhead [95], which is too slow for practical use.
- **Requires recompilation.** Some provenance collectors require users to recompile their entire application and library stack. Recompiling is prohibitively onerous and negates the otherwise low cost of switching to system-level provenance we are pursuing.
- **Requires special hardware.** Some methods require specific CPUs, e.g., Intel CPUs for a dynamic instrumentation tool called Intel PIN. Being limited to specific CPUs violates our goal of promulgating reproducibility to as many people as possible.
- **No source.** We searched the original papers, GitHub, BitBucket, Google, and emailed the first author (CCing the others). If we still could not find the source code for a particular provenance collector, we cannot reproduce it. Note that RecProv is implemented using rr, so we can use rr as a lower-bound for RecProv.
- **Requires custom kernel (Hi-Fi, LPM/ProvMon, CamFlow).** Collectors that modify Linux kernel code are out-of-scope for this work due to their increased maintenance overhead, security risk, and difficulty of system administration. Indeed, many of the systems are too old to be usable: LPM/ProvMon is a patch-set for Linux 2.6.32 (reached end-of-life 2016), Hi-Fi is a patch-set for Linux 3.2 (reached end-of-life in 2018). On the other hand, SingularityCE/Apptainer requires Linux $\geq$ 3.8 for user namespaces.
- **Not reproducible (OPUS).** We tried to get this provenance system to run with several weeks of effort: we emailed the original authors and other authors who used this system, and we left a GitHub issue describing the expected and actual results [4]. However, we still could not get the system to run correctly.
- **Needs more time (DTrace, SPADE, eBPF/bpftrace).** We simply needed more time to implement these provenance collectors.
- **Reproduced/rejected (ltrace, CDE, Sciunit, PTU).** We could reproduce These provenance collectors on some workloads but not others (see Appendix D). Missing values would complicate the data analysis, so we had to exclude these from our running-time experiment.
- **Reproduced (strace, fsatrace, RR, ReproZip, CARE).** We reproduced this provenance collector on all of the benchmarks.

4.2 Implemented Benchmarks

Of these, Table 7 shows the benchmarks used to evaluate each tool, of which there are quite a few. We prioritized implementing frequently-used benchmarks, easy-to-implement benchmarks, and benchmarks that have value in representing a computational science use-case.

Table Table 4 shows the aggregated performance of our implemented benchmarks in our implemented provenance collectors. From this, we observe:

- Although SPLASH-3 CPU-oriented benchmarks contain mostly CPU-bound tasks, they often need to load data from a file, which does invoke the I/O subsystem. They are CPU benchmarks when the CPU is changed and the I/O subsystem remains constant, but when the CPU is constant and the I/O subsystem is changed, the total running time is influenced by I/O-related overhead.
- cp is the slowest benchmark. It even induces a 45% overhead on fsatrace.

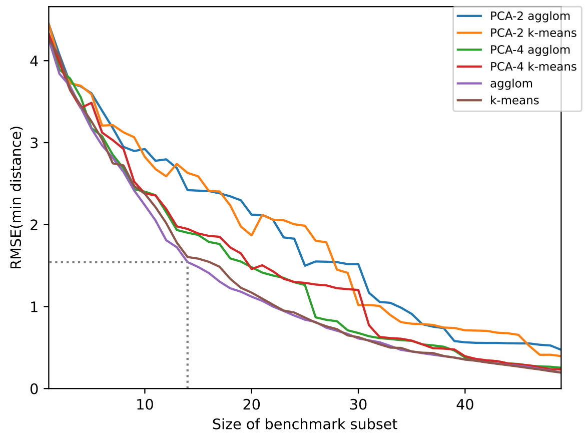
4.3 Subsetted Benchmarks

Figure 2 shows the performance of various algorithms on benchmark subsetting. We observe:

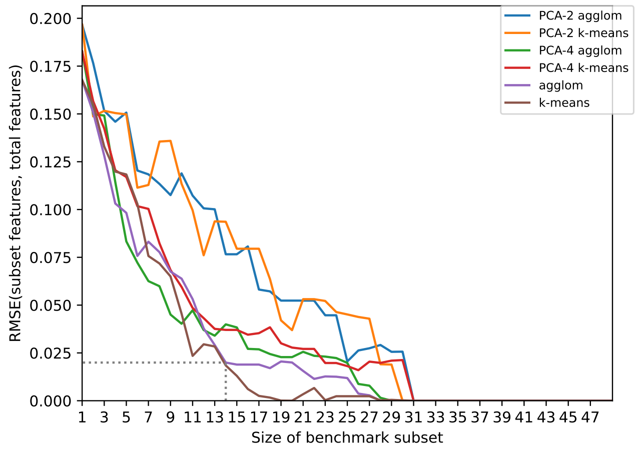
1. The features are already standardized, so PCA has little to offer besides rotation and truncation. However, the truncation is

[4] See https://github.com/dtg-FRESCO/opus/issues/1

[5] URSprung depends on IBM Spectrum Scale to get directory change notifications, so it is not for a *generic* Linux system.



(a) Subsetting algorithms scored by the RMSE of the distance of each benchmark to the nearest selected benchmark. A dotted line shows the x- amd y-value of the point of diminishing return.



(b) Subsetting algorithms scored by the RMSE of the difference between (weighted) features of the subset and features of the original set. A dotted line shows the x- amd y-value of the point of diminishing return.

Figure 2: Competition for best benchmark subsetting algorithm, sweeping over subset size on the x-axis.

throwing away potentially valuable data. Since we have a large number of benchmarks, and the space of benchmarks is open-ended, the additional dimensions that PCA trims off appear to be important for separating clusters of data.

2. K-means and agglomerative clustering yield nearly the same results. They both attempt to minimize within-cluster variance, although by different methods.
3. RMSE of the residual of linear regression will eventually hit zero because the k exceeds the rank of the matrix of features by benchmarks; Linear regression has enough degrees of freedom to perfectly map the inputs to their respective outputs.

It seems that agglomerative clustering with $k = 14$ has performs qquite well, and further increases in k exhibit diminishing returns. At that point, the RMSE of the linear regression is about 0.02. Assuming the error is iid and normally distributed, we can estimate the standard error of the approximation of the total benchmark by linear regression is about 0.02 (log-space) or $e^{0.02} \approx 1.02$ (real-space). Within the sample, 68% of the data falls within one standard error (either multiplied or divided by a factor of 1.02x) and 95% of the data falls within two standard errors ($e^{2 \cdot 0.04}$ or 1.04x). We examine the generated clusters and benchmark subset in Figure 4 and Table 5.

Figure 3a shows the a posteriori clusters with colors. Figure 3b shows a priori benchmark "types", similar but more precise than those in Table 3. From these two, we offer the following observations:

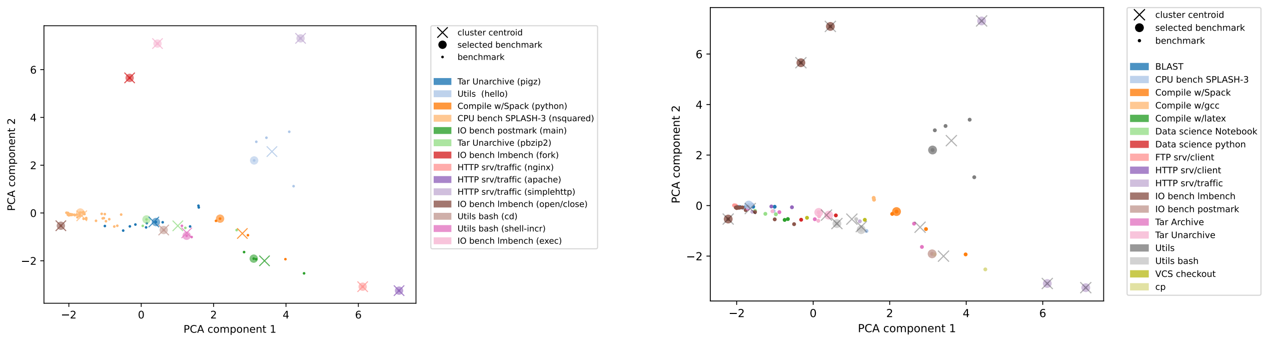
1. It may appear that the algorithm did not select the benchmark closest to the cluster center, but this is because we are viewing a 2D projection of a high-dimensional space, like how three stars may appear next to each other in the sky but in reality, one pair may be much closer than the other, since we cannot perceive the radial distance to each star.
2. Many clusters are singletons, e.g., `simplhttp` near $(4, 6)$; this is surprising, but given there are no points nearby, that decision seems reasonable.
3. We might expect that benchmarks of the same type would occupy nearby points in PCA space, but they often do not. lmbench is particularly scattered with points at $(-2, 0)$ and $(0, 5)$, perhaps because it is a microbenchmark suite where each microbenchmark program tests a different subsystem.
4. Postmark is intended to simulate the file system traffic of a web server (many small file I/O). Indeed the Postmark at $(3.5, -2)$ falls near several of the HTTP servers at $(6, -3)$ and $(7, -3)$. Copy is also similar.

To elucidate the structure of the clusters, we plotted a dendrogram (Figure 4) and listed the members of each cluster (Table 5). We offer the following observations:

1. lmbench fork and lmbenhc exec are close in feature-space, probably because programs usually do both.
2. Utilities (especially GNU hello, which prints hello and exits) terminate very quickly, so they probably measure resources used to load and exit a program. We run these commands in a loop hundreds or thousands of times, so the runtime is more accurately measurable. cd and shell-increment, on the other hand, are shell builtins, so they do not even need to load a program. That cluster probably represents purely CPU-bound workloads.
3. Many of the CPU-heavy workloads are grouped together under lm-protection-fault.
4. Many of the un/archive benchmarks are grouped together with lighttpd, which also accesses many files.

4.3.1 *Our suggested subset.*

- Running a CPU heavy benchmark (**from the 55% cluster** in Table 5) is important, in some sense. It has the heaviest weight



(a) Benchmark subset, where color shows resulting clusters. The same-color small dots are benchmarks in the same cluster, the "x" of that color is their hypothetical benchmark with their average features, and the big dot of that color is the closest actual benchmark to the average of their features. A benchmark subset replaces each cluster of small dots with just the single big dot.

(b) Benchmark subset, where color shows benchmark class (see Table 3). For example, archive-with-gzip and archive-with-bzip2 are two benchmarks of the same type, and therefore color. The "x" still shows a posteriori cluster centers as in Figure 3a.

Figure 3: Benchmarks, clustered agglomeratively into 20 subsets using standardized performance features. These axes show only two dimensions of a high-dimensional space. We apply PCA *after* computing the clusters, in order to project the data into a 2D plane.

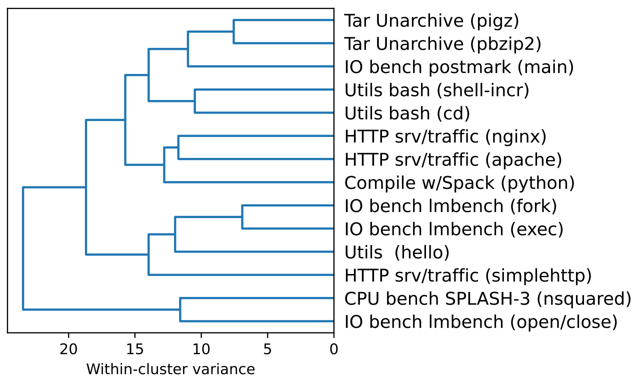


Figure 4: Dendrogram showing the distance between clusters. A fork at $x = x_0$ indicates that below that threshold of within-cluster variance, the two children clsuters are far away enough that they should be split into two; conversely, above that threshold they are close enough to be combined. See Figure 5 for a bigger version that includes all benchmarks.

because more of the selected programs are similar. This weighting will change with the domain, but it holds on our sample of programs.

- The programs in **lmbench** have very different performance characteristics (see Figure 3b). Due to their simplicity, their results are interpretable (e.g., testing latency of `open()` followed by `close()` in a tight loop). We report the total time it takes to run a large number of iterations[6] rather than latency or throughput to be consistent with benchmarks for which latency and throughput are not applicable. If one has to run part of lmbench, it is not too hard to run all of lmbench.
- Figure 3b shows that **HTTP servers are very "unique"**. Three of five were selected as cluster centers, and we can tell from Figure 3a they are quite far from other programs in feature-space. This "uniqueness" means that future work interested in representativeness and consistency with prior work should include HTTP servers, but future work not on security may be able to do without them. In that case, they should run **Postmark** instead, which is intended to mimic the workload of a webserver, and according to **??**, will pull the benchmarks the direction of Nginx and ApacheHttpd.
- Surprisingly, **shell builtins** and **Linux utilities** in a tight loop exercise provenance collectors well according to Table 4, probably due to their fast execution time compared to the fixed cost of loading a program and its libraries into memory. At least they are easy to run.

There is an old adage, *the best benchmark is always the target application.* Benchmarking lmbench reveals certain aspects of performance, but benchmarking the target application reveals the *actual* performance. If we may hazard a corollary, we might say, *the second best benchmark is one from the target domain.* Supposing one does not know the exact application or inputs their audience will use, selecting applications from that domain is the next best option. Future work on system-level provenance for computational science should, of course, use a computational science benchmark, such as BLAST, compiling programs with Spack, or a workflow, whether or not they are selected by this clustering analysis. Likewise, work on security should include HTTP servers.

Finally, researchers presenting new provenance collectors should report *all* benchmark runtimes, not just a geometric mean [59].

[6]Users should set the environment variable `ENOUGH` to a large integer. Otherwise, lmbench will choose a number of iterations based on the observed speed of the machine, which can vary between runs.

Table 2: Provenance collectors from our search results and from experience. See Appendix B for their exact definition.

Tool	Method	Status
strace	tracing	Reproduced
fsatrace	tracing	Reproduced
rr [66]	tracing	Reproduced
ReproZip [20]	tracing	Reproduced
CARE [37]	tracing	Reproduced
Sciunit [70]	tracing	Reproduced/rejected
PTU [70]	tracing	Reproduced/rejected
CDE [33]	tracing	Reproduced/rejected
ltrace	tracing	Reproduced/rejected
SPADE [29]	audit, FS, or compile-time	Needs more time
DTrace [1]	audit	Needs more time
eBPF/bpftrace	audit	Needs more time
SystemTap [74]	audit	Needs more time
PROV-IO [34]	lib. ins.	Needs more time
OPUS [8]	lib. ins.	Not reproducible
CamFlow [68]	kernel ins.	Requires custom kernel
Hi-Fi [73]	kernel ins.	Requires custom kernel
LPM/ProvMon [10]	kernel ins.	Requires custom kernel
Arnold[24]	kern ins.	Requires custom kernel
LPS [23]	kern ins.	Requires custom kernel
RecProv [41]	tracing	No source
FiPS [87]	FS	No source
Namiki et al. [64]	audit	No source
LPROV [91]	kernel mod., lib. ins.	No source
S2Logger [86]	kernel mod.	No source
ProTracer [56]	kernel mod.	No source
PANDDE [26]	kernel ins., FS	No source
PASS/Pasta [62]	kernel ins., FS, lib. ins.	No source
PASSv2/Lasagna [61]	kernel ins.	No source
Lineage FS [78]	kernel ins.	No source
RTAG [40]	bin. ins.	No source
BEEP [51]	bin. ins.	Requires HW
libdft [43]	bin., kernel, lib. ins.	Requires HW
RAIN [39]	bin. ins.	Requires HW
DataTracker [83]	compile-time ins.	Requires HW
MPI[55]	compile-time ins.	Requires recompilation
LDX [48]	VM ins.	Requires recompilation
Panorama [95]	VM ins.	VMs are too slow
PROV-Tracer [82]	audit	VMs are too slow
ETW [5]	audit	Not for Linux
Sysmon [58]	audit	Not for Linux
TREC [90]	tracing	Not for Linux
URSprung [76]	audit	Not for Linux[5]
Ma et al. [54]	audit	Not for Linux
ULTra [15]	tracing	Not for Linux

Readers can be the ones to determine weights for which benchmarks are most relevant to their workload.

4.4 Predictive Model

From the predictive performance model in Table 6, we can see:

- The performance model for ReproZip and RR can be off by almost 30% and 50%, so we should be wary about drawing conclusions from the coefficients of that model.
- RR appears to have a slowdown even on programs with no syscalls. This generalized slowdown could probably be externalized into other independent variables, but the variables we use are not a complete set. The model learned to predict runtimes by the heuristic that all programs get 10% slower, even before counting for syscalls.
- Our LASSO regression is able to ignore features that do not seem to matter. The model for fsatrace is parsimonious. While there certainly is some syscall overhead, it is so small that multiplying

Table 3: Benchmarks implemented by this work. For brevity, we consider categories of benchmarks in Table 7. See Appendix E for a description of each benchmark group and how we implemented them.

Prior works	This work	Instances	Benchmark group and examples from prior work
12	yes	5	HTTP server/traffic
10	yes	2	HTTP server/client
10	yes	8	Compile user packages
9	yes	19 + 1	I/O microbenchmarks (lmbench + Postmark)
9	no		Browsers
6	yes	3	FTP client
5	yes	1	FTP server/traffic
5	yes	5 × 2	Un/archive
5	yes	5	BLAST
5	yes	10	CPU benchmarks (SPLASH-3)
5	yes	8	Coreutils and system utils
3	yes	2	cp
2	yes	2	VCS checkouts
2	no		Sendmail
2	no		Machine learning workflows (CleanML, Spark, ImageML)
1	no		Data processing workflows (VIC, FIE)
1	no		RUBiS
1	no		x264
1	no		mysqld
1	no		gocr
1	no		Memcache
1	no		Redis
1	no		php
1	no		pybench
1	no		ping
1	no		mp3info
1	no		ngircd
1	no		CUPS

Table 4: This table shows percent overhead of the mean walltime when running with a provenance collector versus running without provenance. A value of 1% means the execution in that cell takes 1.01 times the execution without provenance. Negative slowdown can occur sometimes due to random statistical noise. We aggregate values using geometric mean.

	(none)	fsatrace	CARE	strace	RR	ReproZip
BLAST	0	0	2	2	93	8
CPU bench SPLASH-3	0	5	9	16	49	75
Compile w/Spack	0	-1	119	111	562	359
Compile w/gcc	0	4	136	206	321	344
Compile w/latex	0	7	72	40	23	288
Data science Notebook	0	4	15	32	20	174
Data science python	0	5	85	84	150	346
FTP srv/client	0	1	2	4	5	18
HTTP srv/client	0	-23	20	33	165	248
HTTP srv/traffic	0	5	135	414	1261	724
IO bench lmbench	0	-10	1	3	11	36
IO bench postmark	0	2	231	650	259	1733
Tar Archive	0	-0	75	113	179	140
Tar Unarchive	0	4	44	114	195	149
Utils	0	17	118	280	1378	697
Utils bash	0	5	75	20	426	2933
VCS checkout	0	5	71	160	177	428
cp	0	37	641	380	232	5791
Total (gmean)	0	0	45	66	146	193

	Representative	Members
54.7	**CPU bench SPLASH-3 (nsquared)**	BLAST (all), CPU bench SPLASH-3 (ocean, lu, cholesky, radiosity, spatial, volrend, radix, raytrace), Compile w/latex (all), Data science Notebook (all), FTP srv/client (all), HTTP srv/client (all), HTTP srv/traffic (minihttp), IO bench lmbench (write, select-file, mmap, catch-signal, protection-fault, getppid, install-signal, page-fault, fs, bw_unix, select-tcp, bw_file_rd, bw_pipe, read), Tar Archive (gzip, bzip2), Tar Unarchive (bzip2)
12.7	**IO bench postmark (main)**	Tar Archive (archive), cp (all)
7.7	**Tar Unarchive (pbzip2)**	HTTP srv/traffic (lighttpd), Tar Archive (pigz, pbzip2)
5.8	**Compile w/Spack (python)**	Compile w/Spack (rest)
5.4	**Utils (hello)**	Utils (rest)
3.7	**Utils bash (shell-incr)**	CPU bench SPLASH-3 (fft), Utils bash (shell-echo)
1.6	**Utils bash (cd)**	
1.4	**IO bench lmbench (exec)**	
1.4	**HTTP srv/traffic (nginx)**	
1.3	**IO bench lmbench (open/close)**	
1.2	**HTTP srv/traffic (simplehttp)**	
1.0	**IO bench lmbench (fork)**	
0.2	**HTTP srv/traffic (apache)**	
0.0	**Tar Unarchive (pigz)**	Compile w/gcc (all), Data science python (all), IO bench lmbench (stat, fstat), Tar Unarchive (gzip, unarchive), VCS checkout (all)
98.1	**Sum**	

Table 5: A table showing cluster membership and weights as percentages. The weights show one way of approximating the features in the original set, which is by multiplying the features of the cluster representative by the weight and summing over all clusters.

by the original time by $1 + \epsilon$ for a small, positive ϵ is sufficient to approximate the runtime quite well.

- Directory- and socket-related calls are the most expensive. Directory-syscalls includes renames (since they modify the directory entry structure) in which most provenance collectors will copy the entire file into an archive, especially as it occurs in the `cp` benchmark.
- IPC is particularly tricky for RR, which records the exact streams of bytes sent and received. Other provenance collectors may be more relaxed with respect to IPC.

4.5 Discussion

Prior work focuses on security, not computational science. Table 3 shows the top-used benchmarks are server programs, followed by I/O benchmarks. Server programs access many small files with concurrency, which is a different file-access pattern than scientific applications. BLAST (used by 5 / 29 publications with benchmarks; see Table 7) is the only scientific program to be used as a benchmark by more than one publication.

One difference between security and computational science is that security-oriented provenance collectors have to work with adversarial programs: there should be no way for the program to circumvent the provenance tracing, e.g. `PTRACE_DETACH`. Computational science, on the other hand, may satisfied by a solution that *can* be intentionally circumvented by an uncooperative program but would work most of the time, provided it can at least detect when provenance collection is potentially incomplete. Interposing standard libraries, although circumventable, has been used by other tools [93].

Provenance collectors vary in power and speed, but fast-and-powerful could be possible. While all bear the title provenance collector, some are **monitoring**, merely recording a history of operations, while others are **interrupting**, interrupt the process when the program makes an operation. Fsatrace, Strace, and Ltrace are monitoring, while ReproZip, Sciunit, RR, CARE, and CDE are interrupting, using their interruption store a copy of the files that would be read or appended to by the process. None of the interrupting provenance collectors we tested use library interposition or eBPF (although PROV-IO does, we did not have time to implement it). Perhaps a faster underlying method would allow powerful features of interrupting collectors in a reasonable overhead budget.

Provenance collectors are too slow for "always on". One point of friction when using system-level provenance collection is that users have to remember to turn it on, or else the system is useless. An "always on" provenance system could alleviate that problem; for example, a user might change their login shell to start within a provenance collector. Unfortunately, the conventional provenance collectors exhibit an intolerably high overhead to be always used, with the exception of fsatrace. fsatrace is able to so much faster because it uses library interpositioning rather than ptrace (see "fast-and-powerful" discussion above), but fsatrace is one of the weakest collectors; it only collects file reads, writes, moves, deletes, queries, and touches (nothing on process forks and execs).

The space of benchmark performance in provenance systems is highly dimensional. The space of benchmarks is naturally embedded in a space with features as dimensions. If there were many linear relations between the features (e.g., CPU time per second = 1 - (file syscalls per second) * (file syscall latency)), then we would expect clustering to reveal fewer clusters than the number of features. Indeed, there are somewhat more clusters than features ($14 > 12$), it seems that most dimensions are not redundant or if they are, their redundancy is not expressible as linear relationship. Even the relationship between workloads is non-linear; if A is a weighted average of B and C in feature-space, its runtime is not necessarily the same weighted average of B and C's runtime. This complexity is also present when predicting performance as a function of workload features; either the relationship is non-linear, or we are missing a relevant feature.

Computational scientists may already be using workflows. While system-level provenance is the easiest way to get provenance out of many applications, if the application is already written in a workflow engine, such as Pegasus [45], they can get provenance through the engine. Computational scientists may move to workflows for other reasons because they make it easier to parallelize code on big machines and integrate loosely coupled components. That may explain why prior work on system-level provenance focuses more on security applications.

Collector	Rel. err. 95% interval	Intercept	Time multiplier	Estimate syscall latencies (microseconds)						
				metadata	file	IPC	dir	chdir	other	socket
CARE	1.5	0.7	1.0	3	8	444	2	1326	810	
RR	42.6	5.2	1.1	12	17	4725	17	10344	1031	1163
ReproZip	27.4	8.8	1.0	16	16	4287			203	165
fsatrace	0.1	-0.1	1.0							
strace	5.0	1.8	1.0	8	6	941	1	1100		148

Table 6: Coefficients of a predictive performance model. 95% error interval shows that most of the time, the relative error percentage is bounded by the value in that cell. The intercept represents a constant amount of time added on to the programs runtime. fsatrace does not make a program with no syscalls 0.1 seconds faster; this is how the model learned to minimize errors due to missing factors elsewhere. The "original time" column is a multiplier on the original time, probably needed to represent slowdowns not associated with these particular syscalls. Finally, syscall overheads columns show the learned cost of those sycalls in microseconds.

4.6 Threats to Validity

Internal validity: We mitigate measurement noise by: - Isolating the sample machine Section 3.3 - Running the code in cgroups with a fixed allocation of CPU and RAM - Rewriting benchmarks that depend on internet resources to only depend on local resources - Averaging over 3 iterations helps mitigate noise. - Randomizing the order of each pair of collector and benchmark within each iteration We use cross-validation for the performance model

External validity: When measuring the representativeness of our benchmark subset, we use other workload characteristics, not just performance in each collector. Therefore, our set also maintains variety and representativeness in underlying characteristics, not just in the performance we observe. Rather than select the highest cluster value, we select the point of diminishing return, which is more likely to be generalizable.

Regarding the performance model, we use cross-validation to asses out-of-sample generalizability.

5 FUTURE WORK

In the future, we plan to implement compilation for more packages, particularly xSDK [9] packages. Compilation for these packages may differ from ApacheHttpd and Linux because xSDK is organized into many dozens of loosely related packages. We also plan to implement computational workflows. Workflows likely have a different syscall access pattern, unlike HTTP servers because the files may be quite large, unlike `cp` because workflows have CPU work blocked by I/O work, and unlike archiving because there are multiple "stages" to the computation.

We encourage future work that implements an interrupting provenance collector using faster methods like library interposition or eBPF instead of `ptrace`. Between them, there are pros and cons: eBPF requires privileges but could be exposed securely by a setuid/setgid binary; library interposition assumes the tracee only uses libc to make I/O operations. Another optimization postponing work to off the critical path: if a file is read, it can be copied at any time unless/until it gets mutated ("copy-on-write-after-read"). Other reads can be safely copied after the program is done, and new file writes obviously do not need to be copied at all. Perhaps the performance overhead would be low enough to be "always on", however storage and querying cost need to be dispatched with as well.

6 CONCLUSION

We intend this work to bridge research to practical use of provenance collectors and an invitation for future research. In order to bridge research into practice, we identified reproducible and usable provenance collectors from prior work and evaluated their performance on synthetic and real-world workloads. In order to invite future research, we collated and minimized a benchmark suite and identified gaps in prior work. We believe this work and the work it enables will address the practical concerns of a user wanting to use a provenance collector.

A NOTABLE PROVENANCE COLLECTORS

- **CDE** is a record/replay tool proposed by Guo and Engler [33]. During record, CDE uses `ptrace` to intercept its syscalls, and copy relevant files into an archive. During rerun, can use `ptrace` to intercept syscalls and redirect them to files in the archive. PTU uses a modified version of CDE that works on all of our benchmarks, so we can use that as a proxy.
- **ltrace** similar to strace, but it traces dynamic library calls not necessarily syscalls. It still uses `ptrace`.
- **strace** is a well-known system program that uses Linux's `ptrace` functionality to record syscalls, their arguments, and their return code to a file. strace even parses datastructures to write strings and arrays rather than pointers. In this work, we use an strace configuration that captures all file-related syscalls but read/write[7], file-metadata realated syscalls, socket- and IPC- related sycalls but send/recv, and process-related syscalls.
- **fsatrace** reports file I/O using library-interpositioning, a technique where a program mimics the API of a standard library. Programs are written to call into the standard library, but the loader sends those calls to the interpositioning library instead. The interpositioning library can log the call and pass it to another library (possibly the "real" one), so the program's functionality is preserved. This avoids some context-switching overhead of `ptrace`, since the logging happens in the tracee's process.

[7]We do not need to capture individual reads and writes, so long as we capture that the file was opened for reading/writing.

- **CARE** is a record/replay tool inspired by CDE. However, CARE has optimizations enabling it to copy fewer files, and CARE archives can be replayed using `chroot`, `lxc`, or `ptrace` (by emulating `chroot`); CDE only supports `ptrace`, which is slower than the other two.
- **RR** [66] is a record/replay tool. It captures more syscalls than just file I/O, including `getrandom` and `clock_gettime` and it is able to replay its recordings in a debugger. Where other record/replay tools try to identify the relevant files, RR only memorizes the responses to each syscall, so it can only replay that exact code path. CDE, CARE, ReproZip, PTU, and Sciunit allow one to replay a different binary or supply different inputs in the filesystem of an existing recording.
- **ReproZip** is a record/replay inspired by CDE. ReproZip archives can be replayed in Vagrant, Docker, Chroot, or natively. Unlike other record/replay tools, ReproZip explicitly constructs the computational provenance graph.
- **PTU** (Provenance-To-Use) is an adaptation of CDE which explicitly constructs the computational provenance graph.
- **Sciunit** is a wrapper around PTU that also applies block-based deduplication.

B COLLECTION METHODS

- **User-level tracing**: A provenance tool may use "debugging" or "tracing" features provided by the kernel, e.g., `ptrace(2)` [4], to trace another program's I/O operations.
- **Built-in auditing service**: A provenance tool may use auditing service built in to the kernel, e.g., Linux Auditing Framework [57], enhanced Berkeley Packet Filter (eBPF) [2], kprobes [44], and ETW [5] for Windows.
- **Filesystem instrumentation**: A provenance tool may set up a file system, so it can log I/O operations, e.g., using Filesystem in User SpacE (FUSE) interface [3], or Virtual File System (VFS) interface [31].
- **Dynamic library instrumentation**: A provenance tool may replace a library used to execute I/O operations (e.g., glibc) with one that logs the calls before executing them.
- **Binary instrumentation**: A provenance tool may use binary instrumentation (dynamic or static) to identify I/O operations in another program.
- **Compile-time instrumentation**: A provenance tool may be a compiler pass that modifies the program to emit provenance data, especially intra-program control flow.
- **Kernel instrumentation**: A provenance tool may be a modified kernel either by directly modifying and recompiling the kernel's source tree.
- **Kernel module**: Rather than directly modify the kernel's source, the provenance tool may simply require that the user load a custom kernel module.
- **VM instrumentation**: A provenance tool may execute the program in a virtual machine, where it can observe the program's I/O operations.

C TABLE OF BENCHMARKS BY PRIOR PUBLICATION

See Table 7 for a list of prior publications and what benchmarks they use, if, for example, one wishes to see the original contexts in which Firefox was used.

Table 7: Benchmarks used by prior works on provenance collectors (sorted by year of publication).

Publication	Benchmarks	Comparisons
TREC [90]	open/close, compile Apache, LaTeX	Native
ULTra [15]	getpid, LaTeX, Apache, compile package	Native, strace
PASS [62]	BLAST	Native ext2
Panorama [95]	curl, scp, gzip, bzip2	Native
PASSv2 [61]	BLAST, compile Linux, Postmark, Mercurial, Kepler	Native ext3, NFS
SPADEv2 [29]	BLAST, compile Apache, Apache	Native
Hi-Fi [73]	lmbench, compile Linux, Postmark	Native
libdft [43]	scp, {tar, gzip, bzip2} x {extract, compress}	PIN
PTU [70]	Workflows (PEEL0, TextAnalyzer)	Native
LogGC [50]	RUBiS, Firefox, MC, Pidgin, Pine, Proftpd, Sendmail, sshd, vim, w3m, wget, xpdf, yafc, Audacious, bash, Apache, mysqld	None[8]
CARE [37]	Compile perl, xz	Native
Arnold[24]	cp, CVS checkout, make libelf, LaTeX, Apache, gedit, Firefox, spreadsheet, SPLASH-2	Native
LPM/ProvMon [10]	lmbench, compile Linux, Postmark, BLAST	Native
Ma et al. [54]	TextTransfer, Chromium, DrawTool, NetFTP, AdvancedFTP, Apache, IE, Paint, Notepad, Notepad++, simplehttp, Sublime Text	Native
ProTracer [56]	Apache, miniHTTP, ProFTPD, Vim, Firefox, w3m, wget, mplayer, Pine, xpdf, MC, yafc	Auditd, BEEP
LDX [48]	SPEC CPU 2006, Firefox, lynx, nginx, tnftp, sysstat, gif2png, mp3info, prozilla, yopsweb, ngircd, gocr, Apache, pbzip2, pigz, axel, x264	Native
PANDDE [26]	ls, cp, cd, lpr	Native
MPI [55]	Apache, bash, Evince, Firefox, Krusader, wget, most, MC, mplayer, MPV, nano, Pine, ProFTPd, SKOD, TinyHTTPd, Transmission, Vim, w3m, xpdf, Yafc	Audit, LPM-HiFi
CamFlow [68]	lmbench, postmark, unpack kernel, compile Linux, Apache, Memcache, redis, php, pybench	Native
BEEP [51]	Apache, Vim, Firefox, wget, Cherokee, w3m, ProFTPd, yafc, Transmission, Pine, bash, mc, sshd, sendmail	Native
RAIN [39]	SPEC CPU 2006, cp linux, wget, compile libc, Firefox, SPLASH-3	Native
Sciunit [88]	Workflows (VIC, FIE)	Native
LPS [23]	IOR benchmark, read/write, MDTest, HPCG	Native
LPROV [91]	Apache, simplehttp, proftpd, sshd, firefox, filezilla, lynx, links, w3m, wget, ssh, pine, vim, emacs, xpdf	Native
MCI [49]	Firefox, Apache, Lighttpd, nginx, ProFTPd, CUPS, vim, elinks, alpine, zip, transmission, lftp, yafc, wget, ping, procps	BEEP
RTAG [40]	SPEC CPU 2006, scp, wget, compile llvm, Apache	RAIN
URSPRING [76]	open/close, fork/exec/exit, pipe/dup/close, socket/connect, CleanML, Vanderbilt, Spark, ImageML	Native, SPADE
PROV-IO [34]	Workflows (Top Reco, DASSA), I/O microbenchmark (H5bench)	Native
Namiki et al. [64]	I/O microbenchmark (BT-IO)	Native

D NOTE ON FAILED REPRODUCIBILITY

- While we could run **ltrace** on some of our benchmarks, it crashed when processing on the more complex benchmarks, for example FTP server/client. We localized the problem to the following code[9]:

```
/* FIXME: not good -- should use dynamic allocation. 19990703 mortene. */
if (proc->callstack_depth == MAX_CALLDEPTH - 1) {
 fprintf(stderr, "%s: Error: call nesting too deep!\n", __func__);
```

[8]LogGC measures the offline running time and size of garbage collected logs; there is no comparison to native would be applicable.

[9]See https://gitlab.com/cespedes/ltrace/-/blob/8eabf684ba6b11ae7a1a843aca3c0657c6329d73/handle_event.c#L775

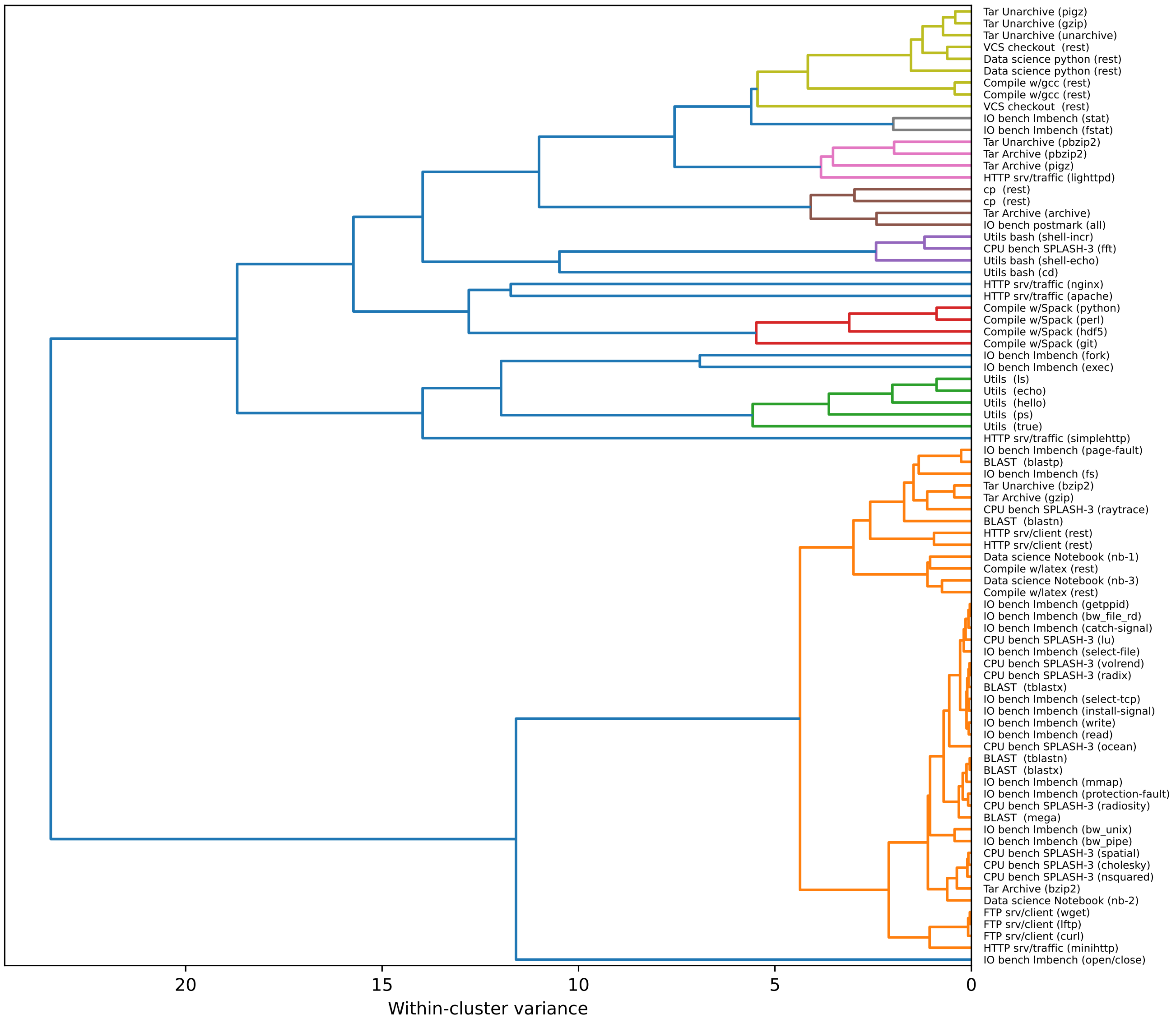


Figure 5: Dendrogram showing the distance between clusters. See Figure 4 for details.

```
  abort();
  return;
}
```

- **CDE** can run some of our benchmarks, but crashes on others, for example BLAST. THe crash occurs when trying to copy from the tracee process to the tracer due to `ret == NULL`[10]:

```
static char* strcpy_from_child(struct tcb* tcp, long addr) {
  char* ret = strcpy_from_child_or_null(tcp, addr);
  EXITIF(ret == NULL);
  return ret;
}
```

  The simplest explanation would be that the destination buffer is not large enough to store the data that `strcpy` wants to write. However, the destination buffer is `PATHMAX`.

[10]See https://github.com/usnistgov/corr-CDE/blob/v0.1/strace-4.6/cde.c#L2650

- **PTU** seems to work on most test cases outside of our BenchExec container. However, there is a bug causing it to crash inside our container.
- **Sciunit** works on most benchmarks, but exhausts the memory of our system when processing FTP server/client and Spack compile package. We believe this is simply due to the benchmarks manipulating a large number of files and Sciunit trying to deduplicate them all.

E BENCHMARK DESCRIPTIONS

- The most common benchmark classes from prior work are, **HTTP servers/traffic**, **HTTP servers/clients**, **FTP servers/traffic**, and **FTP servers/clients** are popular because prior work focuses overwhelmingly on provenance for the sake of security (auditing, intrusion detection, or digital forensics). While these benchmarks may not be specifically relevant for computational science workloads, we wanted to include them in our suite to improve our coverage of benchmarks used frequently in prior works. We implemented 5 HTTP servers (ApacheHttpd, miniHTTP, Python's http.server, lighttpd, Nginx) running against traffic from Hey (successor to ApacheBench) and 2 HTTP clients (curl and Wget). We implemented 1 FTP server (ProFTPD) running against traffic from httpbench[11] and 3 FTP clients (curl, Wget, and lftp).

- **Compiling packages** from source is a common operation in computational science, so we implemented as many of these as we could and also implemented some of our own. However, compiling glibc and LLVM takes much longer than everything else in the benchmark suite, so we excluded LLVM and glibc. We implemented a pattern for compiling packages from Spack that discounts the time taken to download sources, counting only the time taken to unpack, patch, configure, compile, link, and install them. We implemented compiling Python, HDF5, git, and Perl.
- Implementing headless for **browsers** in "batch-mode" without GUI interaction is not impossibly difficult, but non-trivial. Furthermore, we deprioritized this benchmark because few computational science applications resemble the workload of a web browser.
- **Archive** and **unarchiving** is a common task for retrieving data or source code. We benchmark un/archiving several archives with several compression algorithms. Choosing a compression algorithm may turn an otherwise I/O-bound workload to a CPU-bound workload, which would make the impact of provenance tracing smaller. We implemented archive and unarchiving a medium-sized project (7 MiB uncompressed) with no compression, gzip, pigz, bzip, and pbzip2.
- **I/O microbenchmarks** could be informative for explicating which I/O operations are most affected. Prior work uses lmbench [60], which benchmarks individual syscalls, Postmark [42], which focuses on many small I/O operations (typical for web servers), IOR [81], H5bench [52] and BT-IO[12], which are specialized for parallel I/O on high-performance machines, and custom benchmarks, for example running open/close in a tight loop. Since we did not have access to a high-performance machine, we used lmbench and Postmark. We further restrict lmbench to the test-cases relevant to I/O and used by prior work.
- **BLAST** [7] is a search for a fuzzy string in a protein database. However, unlike prior work, we split the benchmark into query groups described by Coulouris [22], since the queries have different performance characteristics: blastn (nucleotide-nucleotide BLAST), megablast (large numbers of query sequences) blastp (protein-protein BLAST), blastx (nucleotide query sequence against a protein sequence database), tblastn (protein query against the six-frame translations of a nucleotide sequence database), tblastx (nucleotide query against the six-frame translations of a nucleotide sequence database).

[11]See https://github.com/selectel/ftpbench

[12]See https://www.nas.nasa.gov/software/npb.html

- Prior work uses several **CPU benchmarks**: SPEC CPU INT 2006 [35], SPLASH-3 [77], SPLASH-2 [92] and HPCG [36]. While we do not expect CPU benchmarks to be particularly enlightening for provenance collectors, which usually only affect I/O performance, it was used in three prior works, so we tried to implement both. SPLASH-3 is an updated and fixed version of the same benchmarks in SPLASH-2. However, SPEC CPU INT 2006 is not free (as in beer), so we could only implement SPLASH-3.
- **Sendmail** is a quite old mail server program. Mail servers do not resemble a computational science workload, and it is unclear what workload we would run against the server. Therfore, we deprioritized this benchmark and did not implement it.
- **VCS checkouts** are a common computational science operation. We simply clone a repository (untimed) and run `$vcs checkout $commit` for random commits in the repository. CVS does not have a notion of global commits, so we use Mercurial and Git.
- VIC, FIE, ImageML, and Spark are real-world examples of **Data processing** and **machine-learning workflows**. We would like to implement these, but reproducing those workflows is non-trivial; they each require their own computational stack. For FIE, in particular, there is no script that glues all of the operations together; we would have to read the publication [13] which FIE supports to understand the workflow, and write our own script which glues the operations together.
- We did not see a huge representative value in **coreutils and friends (bash, cp, ls, procps)** that would not already be gleaned from lmbench, but due to its simplicity and use in prior work, we implemented it anyway. For `bash`, we do not know what exact workload prior works are using, but we test the speed of incrementing an integer and changing directories (`cd`).
- The **other** benchmark programs are mostly specific desktop applications used only in one prior work. These would likely not yield any insights not already yielded by the benchmarks we implemented, and for each one we would need to build it from source, find a workload for it, and take the time to run it. They weigh little in the argument that our benchmark suite represents prior work, since they are only used in one prior work.

F REPRODUCING

We split this into three steps:

1. Getting the software environment.
2. Running the benchmarks.
3. Running the analysis.

F.1 Getting the software environment with Nix

Nix package manager[13] is a user-level package manager available on many platforms. Nix uses build-from-source and a binary cache; this is more resilient than Dockerhub because if the binary cache goes down or removes packages for any reason, the client can always build them from source, so long as the projects don't disappear from GitHub. We considered creating a Docker image, but BenchExec, the tool we use to get consistent running times, manipulates cgroups and systemd, and we did not have enough time to figure out how to run in Docker or Podman.

Install Nix with:

[13]See https://nixos.org/guides/how-nix-works

```
$ curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | sh -s -- install
```

This installer also enables "flakes" and the "nix-command". If you installed Nix by another method, see this page to enable flakes and the nix-command.

```
$ git clone https://github.com/charmoniumQ/prov-tracer
$ cd prov-tracer/benchmark
```

Use Nix to build. We used `--max-jobs` to enable parallelism. This step takes about an hour on a modest machine with residential internet.

```
$ nix build --print-build-logs --max-jobs $(nproc) '.#env'
```

F.2 Running the benchmarks

Note that we use the Python from our software environment, not from the system, to improve determinism. We wrote a front-end to run the scripts called `runner.py`.

Run with `--help` for more information. Briefly, it takes a `--collectors`, `--workloads`, `--iterations`, and `--parallelism` arguments, which specify what to run. For the paper, we ran

```
./result/bin/python runner.py --collectors working --workloads working --iterations 3 --parallelism 1
```

Multiple `--collectors` and `--workloads` can be given, for example,

```
./result/bin/python runner.py --collectors noprov --collectors strace --workloads lmbench --workloads postmark
```

See the bottom of `prov_collectors.py` and `workloads.py` for the name-mapping.

G RUNNING THE ANALYSIS

The analysis is written in a Jupyter notebook stored in `notebooks/cross-val.ipynb`. The notebook is commented. It begins by checking for anomalies in the data, which we've automated as much as possible, but please sanity check the graphs before proceeding.

The notebook can eb launched from our software environment by:

```
env - PATH=$PWD/result/bin/jupyter notebook
```

H OPEN SOURCE CONTRIBUTIONS

The actual benchmark set and statistical analysis are open-source:

- https://github.com/charmoniumQ/prov-tracer/

This work necessitated modifying Spack, Sciunit, PTU, jupyter-contrib-nbextensions, Nixpkgs, ftpbench, and benchexec. Where appropriate, we submitted as pull-requests to the respective upstream projects.

The following are merged PRs developed as a result of this work:

- https://github.com/depaul-dice/sciunit/pull/35
- https://github.com/spack/spack/pull/42159
- https://github.com/spack/spack/pull/42199
- https://github.com/spack/spack/pull/42114
- https://github.com/selectel/ftpbench/pull/5
- https://github.com/selectel/ftpbench/pull/4
- https://github.com/sosy-lab/benchexec/pull/984
- https://github.com/NixOS/nixpkgs/pull/263829
- https://github.com/NixOS/nixpkgs/pull/257396

The following are open PRs developed as a result of this work:

- https://github.com/spack/spack/pull/39902
- https://github.com/spack/spack/pull/42131
- https://github.com/spack/spack/pull/41048
- https://github.com/sosy-lab/benchexec/pull/990
- https://github.com/depaul-dice/sciunit/pull/36
- https://github.com/depaul-dice/provenance-to-use/pull/4
- https://github.com/depaul-dice/provenance-to-use/pull/5
- https://github.com/ipython-contrib/jupyter_contrib_nbextensions/pull/1649
- https://github.com/NixOS/nixpkgs/issues/268542

I REFERENCES

REFERENCES

[1] [n. d.]. About DTrace.

[2] [n. d.]. BPF Documentation. https://docs.kernel.org/bpf/index.html.

[3] [n. d.]. FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[4] [n. d.]. Ptrace. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[5] 2021. Event Tracing - Win32 Apps. https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-portal.

[6] ACM Inc. staff. 2020. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[7] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic Local Alignment Search Tool. *Journal of Molecular Biology* 215, 3 (Oct. 1990), 403–410. https://doi.org/10.1016/S0022-2836(05)80360-2

[8] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2013. {OPUS}: A Lightweight System for Observational Provenance in User Space. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*.

[9] Roscoe Bartlett, Irina Demeshko, Todd Gamblin, Glenn Hammond, Michael Allen Heroux, Jeffrey Johnson, Alicia Klinvex, Xiaoye Li, Lois Curfman McInnes, J. David Moulton, Daniel Osei-Kuffuor, Jason Sarich, Barry Smith, James Willenbring, and Ulrike Meier Yang. 2017. xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit. *Supercomputing Frontiers and Innovations* 4, 1 (Feb. 2017), 69–82. https://doi.org/10.14529/jsfi170104

[10] Adam Bates, Dave (Jing) Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy {Whole-System} Provenance for the Linux Kernel. In *24th USENIX Security Symposium (USENIX Security 15)*. 319–334.

[11] Marijan Beg, Juliette Taka, Thomas Kluyver, Alexander Konovalov, Min Ragan-Kelley, Nicolas M Thiéry, and Hans Fangohr. 2021. Using Jupyter for reproducible scientific workflows. *Computing in Science & Engineering* 23, 2 (2021), 36–46.

[12] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable Benchmarking: Requirements and Solutions. *Int J Softw Tools Technol Transfer* 21, 1 (Feb. 2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[13] Mirza M. Billah, Jonathan L. Goodall, Ujjwal Narayan, Bakinam T. Essawy, Venkat Lakshmi, Arcot Rajasekar, and Reagan W. Moore. 2016. Using a Data Grid to Automate Data Preparation Pipelines Required for Regional-Scale Hydrologic Modeling. *Environmental Modelling & Software* 78 (April 2016), 31–39. https://doi.org/10.1016/j.envsoft.2015.12.010

[14] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.

[15] A.N. Burton and P.H.J. Kelly. 1998. Workload Characterization Using Lightweight System Call Tracing and Reexecution. In *1998 IEEE International Performance, Computing and Communications Conference. Proceedings (Cat. No.98CH36191)*. 260–266. https://doi.org/10.1109/PCCC.1998.659975

[16] S.P. Callahan, J. Freire, E. Santos, C.E. Scheidegger, C.T. Silva, and Huy T. Vo. 2006. Managing the Evolution of Dataflows with VisTrails. In *22nd International Conference on Data Engineering Workshops (ICDEW'06)*. 71–71. https://doi.org/10.1109/ICDEW.2006.75

[17] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2018. The Role of Rapid Reviews in Supporting Decision-Making in Software Engineering Practice. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018 (EASE '18)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3210459.3210462

[18] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2020. Rapid Reviews in Software Engineering. In *Contemporary Empirical Methods in Software Engineering*, Michael Felderer and Guilherme Horta Travassos (Eds.). Springer International Publishing, Cham, 357–384. https://doi.org/10.1007/978-3-030-32489-6_13

[19] Kyle Chard, Niall Gaffney, Matthew B Jones, Kacper Kowalik, Bertram Ludäscher, Jarek Nabrzyski, Victoria Stodden, Ian Taylor, Matthew J Turk, and Craig Willis. 2019. Implementing computational reproducibility in the Whole Tale environment. In *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems*. 17–22.

[20] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the*

2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 2085–2088. https://doi.org/10.1145/2882903.2899401

[21] Christian Collberg and Todd A. Proebsting. 2016. Repeatability in Computer Systems Research. *Commun. ACM* 59, 3 (Feb. 2016), 62–69. https://doi.org/10.1145/2812803

[22] George Coulouris and NIH Staff. 2016. Blast Benchmark. https://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/blast-benchmark.

[23] Dong Dai, Yong Chen, Philip Carns, John Jenkins, and Robert Ross. 2017. Lightweight Provenance Service for High-Performance Computing. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 117–129. https://doi.org/10.1109/PACT.2017.14

[24] David Devecsery, MIchael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 525–540. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/devecsery

[25] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature biotechnology* 35, 4 (2017), 316–319.

[26] Daren Fadolalkarim, Asmaa Sallam, and Elisa Bertino. 2016. PANDDE: Provenance-based ANomaly Detection of Data Exfiltration. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. Association for Computing Machinery, New Orleans Louisiana USA, 267–276. https://doi.org/10.1145/2857705.2857710

[27] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. Provenance for Computational Tasks: A Survey. *Comput. Sci. Eng.* 10, 3 (May 2008), 11–21. https://doi.org/10.1109/MCSE.2008.79

[28] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R De Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[29] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Middleware 2012 (Lecture Notes in Computer Science)*, Priya Narasimhan and Peter Triantafillou (Eds.). Springer, Berlin, Heidelberg, 101–120. https://doi.org/10.1007/978-3-642-35170-9_6

[30] Jeremy Goecks, Anton Nekrutenko, and James Taylor. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology* 11, 8 (2010), 1–13.

[31] Gooch. [n. d.]. Overview of the Linux Virtual File System. https://docs.kernel.org/filesystems/vfs.html.

[32] Samuel Grayson, Darko Marinov, Daniel S. Katz, and Reed Milewicz. 2023. Automatic Reproduction of Workflows in the Snakemake Workflow Catalog and Nf-Core Registries. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability (ACM REP '23)*. Association for Computing Machinery, New York, NY, USA, 74–84. https://doi.org/10.1145/3589806.3600037

[33] Philip Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *2011 USENIX Annual Technical Conference*. USENIX, Portland, OR, USA.

[34] Runzhou Han, Suren Byna, Houjun Tang, Bin Dong, and Mai Zheng. 2022. PROV-IO: An I/O-Centric Provenance Framework for Scientific Data on HPC Systems. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/3502181.3531477

[35] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[36] Michael Allen Heroux, Jack Dongarra, and Piotr Luszczek. 2013. *HPCG Benchmark Technical Specification.* Technical Report SAND2013-8752. Sandia National Lab. https://doi.org/10.2172/1113870

[37] Yves Janin, Cédric Vincent, and Rémi Duraffort. 2014. CARE, the Comprehensive Archiver for Reproducible Execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering (TRUST '14)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/2618137.2618138

[38] Christoph Jansen, Jonas Annuscheit, Bruno Schilling, Klaus Strohmenger, Michael Witt, Felix Bartusch, Christian Herta, Peter Hufnagl, and Dagmar Krefting. 2020. Curious Containers: A framework for computational reproducibility in life sciences with support for Deep Learning applications. *Future Generation Computer Systems* 112 (2020), 209–227.

[39] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/3133956.3134045

[40] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling Refinable {Cross-Host} Attack Investigation with Efficient Data Flow Tagging and Tracking. In *27th USENIX Security Symposium (USENIX Security 18)*. 1705–1722.

[41] Yang Ji, Sangho Lee, and Wenke Lee. 2016. RecProv: Towards Provenance-Aware User Space Record and Replay. In *Provenance and Annotation of Data and Processes (Lecture Notes in Computer Science)*, Marta Mattoso and Boris Glavic (Eds.). Springer International Publishing, Cham, 3–15. https://doi.org/10.1007/978-3-319-40593-3_1

[42] Jeffrey Katcher. 2005. *PostMark: A New File System Benchmark.* Technical Report TR3022.

[43] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042

[44] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. [n. d.]. Kernel Probes (Kprobes). https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[45] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. 2008. Provenance trails in the Wings/Pegasus system. *Concurrency and Computation: Practice and Experience* 20, 5 (2008), 587–597. https://doi.org/10.1002/cpe.1228 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1228

[46] Johannes Köster and Sven Rahmann. 2012. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (2012), 2520–2522.

[47] Markus Kowalewski and Phillip Seeber. 2022. Sustainable packaging of quantum chemistry software with the Nix package manager. *International Journal of Quantum Chemistry* 122, 9 (2022), e26872.

[48] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 503–515. https://doi.org/10.1145/2872362.2872395

[49] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. 2018. MCI : Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23306

[50] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1005–1016. https://doi.org/10.1145/2508859.2516731

[51] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 2017 Network and Distributed System Security (NDSS) Symposium*.

[52] Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang. 2021. h5bench: HDF5 I/O kernel suite for exercising HPC I/O patterns. In *Proceedings of Cray User Group Meeting, CUG*, Vol. 2021.

[53] Fang (Cherry) Liu, Mehmet Belgin, Nuyun Zhang, Kevin Manalo, Ruben Lara, Christopher P. Stone, and Paul Manno. 2022. ProvBench: A performance provenance capturing framework for heterogeneous research computing environments. *Concurrency and Computation: Practice and Experience* 34, 10 (2022), e6820. https://doi.org/10.1002/cpe.6820 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6820

[54] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC '15)*. Association for Computing Machinery, New York, NY, USA, 401–410. https://doi.org/10.1145/2818000.2818039

[55] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. {MPI}: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*. 1111–1128.

[56] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2016.23350

[57] Ashish Bharadwaj Madabhushana. 2021. Configure Linux System Auditing with Auditd.

[58] markruss. 2023. Sysmon - Sysinternals. https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon.

[59] John R. Mashey. 2004. War of the benchmark means: time for a truce. *SIGARCH Comput. Archit. News* 32, 4 (sep 2004), 1–14. https://doi.org/10.1145/1040136.1040137

[60] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*. USENIX,

San Diego, CA.

[61] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, USA, 10. https://doi.org/10.5555/1855807.1855817

[62] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *2006 USENIX Annual Technical Conference*.

[63] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong! *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 265–276. https://doi.org/10.1145/2528521.1508275

[64] Yuta Namiki, Takeo Hosomi, Hideyuki Tanushi, Akihiro Yamashita, and Susumu Date. 2023. A Method for Constructing Research Data Provenance in High-Performance Computing Systems. In *2023 IEEE 19th International Conference on E-Science (e-Science)*. 1–10. https://doi.org/10.1109/e-Science58273.2023.10254932

[65] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J Eglen, Tim Head, Tony Hirst, and Benjamin D Evans. 2020. Ten simple rules for writing Dockerfiles for reproducible data science. , e1008316 pages.

[66] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. https://doi.org/10.48550/arXiv.1705.05937 arXiv:1705.05937 [cs]

[67] The National Academies of Sciences, Engineering, & Medicine. 2019. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC. https://doi.org/10.17226/25303

[68] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 405–418. https://doi.org/10.1145/3127479.3129249

[69] Jeffrey M Perkel. 2021. Ten computer codes that transformed science. *Nature* 589, 7842 (2021), 344–349.

[70] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using Provenance for Repeatability. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*.

[71] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. 2007. Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Archit. News* 35, 1 (mar 2007), 69–76. https://doi.org/10.1145/1241601.1241616

[72] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2019. A Survey on Collecting, Managing, and Analyzing Provenance from Scripts. *ACM Comput. Surv.* 52, 3 (June 2019), 47:1–47:38. https://doi.org/10.1145/3311955

[73] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 259–268. https://doi.org/10.1145/2420950.2420989

[74] Vara Prasad, William Cohen, Frank Eigler, Martin Hunt, Jim Keniston, and Brad Chen. 2005. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the Linux Symposium*, Vol. 2. kernel.org, Ottawa, Ontario, Canada, 49–64.

[75] Eric D Ragan, Alex Endert, Jibonananda Sanyal, and Jian Chen. 2015. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 31–40.

[76] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. 2020. Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3354–3368. https://doi.org/10.14778/3415478.3415556

[77] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 101–111. https://doi.org/10.1109/ISPASS.2016.7482078

[78] Can Sar and Pei Cao. [n. d.]. Lineage File System. ([n. d.]).

[79] Mahadev Satyanarayanan, Jan Harkes, and James Blakley. 2023. Towards Reproducible Execution of Closed-Source Applications from Internet Archives. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability*. 15–26.

[80] Holger J. Schünemann and Lorenzo Moja. 2015. Reviews: Rapid! Rapid! Rapid! …and Systematic. *Systematic Reviews* 4, 1 (Jan. 2015), 4. https://doi.org/10.1186/2046-4053-4-4

[81] Hongzhang Shan and John Shalf. 2007. Using IOR to analyze the I/O Performance for HPC Platforms. In *Conference: Cray User Group Conference*. https://www.osti.gov/biblio/923356

[82] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Decoupling Provenance Capture and Analysis from Execution. In *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance (TaPP'15)*. USENIX Association, USA, 3.

[83] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Looking Inside the Black-Box: Capturing Data Provenance Using Dynamic Instrumentation. In *Provenance and Annotation of Data and Processes (Lecture Notes in Computer Science)*, Bertram Ludäscher and Beth Plale (Eds.). Springer International Publishing, Cham, 155–167. https://doi.org/10.1007/978-3-319-16462-5_12

[84] Victoria Stodden, Christophe Hurlin, and Christophe Pérignon. 2012. RunMyCode. org: A novel dissemination and collaboration platform for executing published computational results. In *2012 IEEE 8th International Conference on E-Science*. IEEE, 1–8.

[85] Victoria Stodden, Sheila Miguez, and Jennifer Seiler. 2015. Researchcompendia. org: Cyberinfrastructure for reproducibility and collaboration in computational science. *Computing in Science & Engineering* 17, 1 (2015), 12–19.

[86] Chun Hui Suen, Ryan K.L. Ko, Yu Shyang Tan, Peter Jagadpramana, and Bu Sung Lee. 2013. S2Logger: End-to-End Data Tracking Mechanism for Cloud Data Provenance. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. 594–602. https://doi.org/10.1109/TrustCom.2013.73

[87] Salmin Sultana and Elisa Bertino. 2013. A File Provenance System. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY '13)*. Association for Computing Machinery, New York, NY, USA, 153–156. https://doi.org/10.1145/2435349.2435368

[88] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable Research Objects. In *2017 IEEE 13th International Conference on E-Science (e-Science)*. 374–383. https://doi.org/10.1109/eScience.2017.51

[89] Ana Trisovic, Matthew K. Lau, Thomas Pasquier, and Mercè Crosas. 2022. A Large-Scale Study on Research Code Quality and Execution. *Sci Data* 9, 1 (Feb. 2022), 60. https://doi.org/10.1038/s41597-022-01143-6

[90] Amin Vahdat and Thomas Anderson. 1998. Transparent Result Caching. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '98)*. USENIX Association, USA, 3.

[91] Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2018. Lprov: Practical Library-aware Provenance Tracing. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 605–617. https://doi.org/10.1145/3274694.3274751

[92] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (S. Margherita Ligure, Italy) *(ISCA '95)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

[93] Cong Xu, Shane Snyder, Vishwanath Venkatesan, Philip Carns, Omkar Kulkarni, Suren Byna, Roberto Sisneros, and Kalyana Chadalavada. 2017. *DXT: Darshan eXtended Tracing*. Technical Report. Argonne National Lab. (ANL), Argonne, IL (United States).

[94] Joshua J. Yi, Resit Sendag, Lieven Eeckhout, Ajay Joshi, David J. Lilja, and Lizy K. John. 2006. Evaluating Benchmark Subsetting Approaches. In *2006 IEEE International Symposium on Workload Characterization*. 93–104. https://doi.org/10.1109/IISWC.2006.302733

[95] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

[96] Jun Zhao, Jose-Manuel Gomez-Perez, Khalid Belhajjame, Graham Klyne, Esteban Garcia-cuesta, Aleix Garrido, Kristina Hettne, Marco Roos, David De Roure, and Carole Goble. 2012. Why Workflows Break — Understanding and Combating Decay in Taverna Workflows. In *2012 IEEE 8th International Conference on E-Science (e-Science)*. IEEE, Chicago, IL, 9. https://doi.org/10.1109/eScience.2012.6404482