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

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

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

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

ACM Reference Format:
Samuel Grayson, Faustino Aguilar, Daniel S. Katz, Reed Milewicz, and Darko Marinov. 2024. A benchmark suite and performance analysis of user-space provenance collectors. In . ACM, New York, NY, USA, 13 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 [56]. 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 [15, 32, 54, 63], package managers for fine-grained management of dependencies [25, 40], interactive notebooks and workflows[12, 22, 39], and online platforms for archiving and sharing computational experiments[18, 27, 67, 68]. In this work, we focus our attention 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 end result [24]; 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 independently developed. What has been lacking, however, is a rigorous comparison of those available tools and the extent to which they are practically usable in CSE application contexts[1]. In an effort 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 provenance tools*: There are scores of academic publications on system-level provenance (see Table 2), and we collated a list of available of provenance tools.
  [2] and classify them by *capture method* (e.g., does the provenance collector require you to load a kernel module or run your code in a VM?).
- *A benchmark suite*: Prior work does not use a consistent set of benchmarks; often publications use an overlapping set of benchmarks from prior work. We collate benchmarks used in prior work, add some unrepresented areas, and find a statistically valid subset of the benchmark.
- *A quantitative performance comparison of tools against this suite*: Prior publications often only compares the performance their provenance tool to the baseline, no-provenance performance, not to other provenance tools. It is difficult to compare provenance tools, given data of different benchmarks on different machines. We run a consistent set of benchmarks on a single machine over all provenance tools.
- *A predictive performance model for provenance tools*: The performance overhead of a single provenance collector varies from <1% to 23% [52] than without provenance depending on the application, so a single number for overhead is not sufficient. We develop a statistical model for predicting the overhead of $X application in $Y provenance collector based on $Y provenance collector's performance on our benchmark suite and $X application's performance characteristics (e.g., number of I/O syscalls).

[1] DSK: usable globally or perhaps in particular situations?
[2] DSK: as possible is problematic, probably need to rephrase

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The remainder of the paper is structured as follows. [^RMM: Outline paper structure here.]

2 BACKGROUND

Provenance tools and data have many potential applications, including the following from Pimentel et al. [58] and Sar and Cao [62]:

1. **Reproducibility**. A description of the inputs and processes used to generate a specific output can aid manual and automatic reproduction of that output[3]. Lack of reproducibility in computational experiments undermines the long-term credibility of science and hinders the day-to-day work of researchers. Empirical studies [20, 29, 72, 78] 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.
   - **Manual reproducibility**. Provenance data improves manual reproducibility, because users have a record of the inputs, outputs, and processes used to create a computational artifact.
   - **Automatic reproducibility**. 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". 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 process 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 a 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. 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?".
4. **Data cataloging**. Provenance data can help catalog, label, and recall experimental results based on the input parameters. For example, a user might have run dozens of different versions of their workflow, and they may want to ask an automated system, "show me the results I previously computed based on that data with this algorithm?".
5. **Space compression**. If the provenance of a particular artifact is known, the artifact may be able to be deleted to save space, and regenerated when needed. Historically, as computing systems has improved, a later regeneration takes less time than the original.

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 emit provenance data to report provenance data [24].

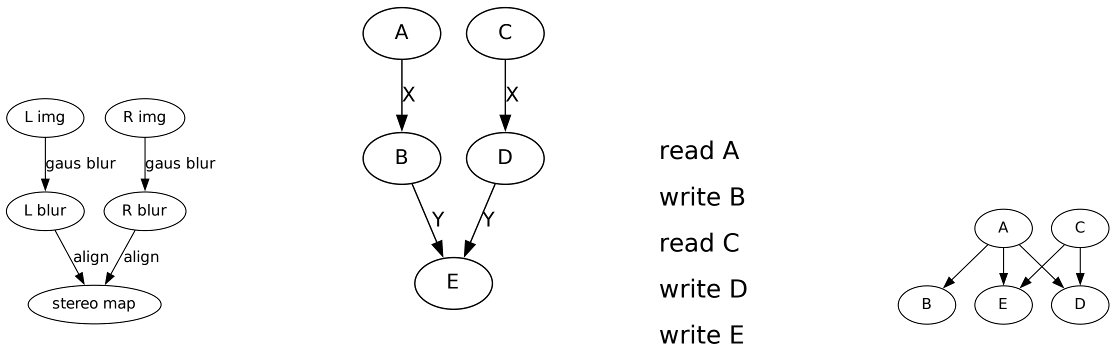
- **Application-level** provenance is the most semantically rich, since it knows the use of each input at the application-level (see Section 2), but the least general, since each application would have to be modified individually.
- **Workflow-level** or **language-level** provenance is a middle ground in semantic richness and generality; it only knows the use of inputs in a dataflow sense (see Section 2), but all applications using the provenance-modified workflow engine or programming language would emit provenance data without themselves being modified to emit provenance data.
- **System-level** is the most general, since all applications on the system would emit provenance data, but it is the least semantically rich, since observed dependencies may overapproximate the true dependencies (see Section 2 and Section 2). System-level provenance collectors may be implemented in **kernel-space** or in **user-space**. Since kernel-space provenance collectors modify internals of the Linux kernel, keeping them up-to-date as the kernel changes is a significant maintenance burden. High-security national labs may be wary of including a patched kernel. On the other hand, user-space collectors compromise performance in exchange for requiring less maintenance and less privilege.

One may imagine an abstract tradeoff curve between "enabling provenance applications such as reproducibility" as the horizontal axis increasing rightwards and "cost of implementation" that provenance data on the vertical axis increasing upwards). A typical status quo, not collecting any provenance data and not using workflows, is at the bottom left: no added cost and does nothing to enable provenance applications. System-level, workflow/language-level, and application-level are on a curve, increasing cost and enabling more provenance applications.

The implementation cost in adopting system-level provenance in a project which currently has no provenance is low because the user need not change *anything* about their application; they merely need to install some provenance tracer onto their system and run their code, without modifying it, in the tracer. [4] Perceived ease of use is a critical factor in the adoption of new technologies (formalized in the Technology Acceptance Model [21]). Although the user may eventually use more semantically rich provenance, low-initial-cost system-level provenance would get provenance's "foot in the door". While this data is less rich than that of the workflow or application

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

[4] DSK: what about the performance penalty? Since you talk about performance in contributions, I think you have to introduce it here. SAG: This is referring to the "cost of switching from no-prov to prov", which is low, and I'm only using this argument to explain why I look at system-level over the others. Performance overhead between system-level tools is a concern that I will address later on. DSK: maybe add a word ("implementation"?) before cost to say which cost is meant here?



(a) Application-level provenance has the most semantic information.

(b) Workflow-level provenance has an intermediate amount of semantic information.

(c) System-level log of I/O operations.

(d) System-level provenance, inferred from the log in Fig. 1c., has the least amount of semantic information

Figure 1: Several provenance graphs collected at different levels for the same application.

level, it may be enough to enable important applications such as reproducibility, caching, etc. 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.

While there is little added human overhead 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 that a program might execute per second.

Prior publications in system-level provenance usually contains some benchmark programs to evaluate the overhead imposed by the system-level provenance tool. However, the set of chosen benchmark programs are not consistent from one publication to another, and overhead can be extermely sensitive to the exact choice of benchmark, so these results are totally 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.

3 METHODS

3.1 Rapid Review

We began 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 [64] 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 [16, 17].

We conducted a rapid review with the following parameters:

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

We record the following features for each system-level provenance tool:

- **Capture method**: What method does the tool use to capture provenance?
  - **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 [49], enhanced Berkeley Packet Filter (eBPF) [2], kprobes [38], 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 [28].
  - **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.

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++), or benchmarks with an interactive component (e.g., GNU Midnight Commander).

We implemented the benchmarks as packages for the Nix package manager[5], 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.
- **Compilations**: Prior work uses compilation of Apache or of Linux. We added compilation of several other packages (any package in Spack) to our benchmark. Compiling packages is a good use-case for a provenance tracer, because a user might trial-and-error multiple compile commands and not remember the exact sequence of "correct" commands; the provenance tracker would be able to recall the commands which did not get overwritten, so the user can know what commands "actually worked". [6]

3.3 Performance Experiment

To get consistent measurements, we select as many benchmarks and provenance tracers as we reasonably can, and run a complete matrix (every tracer on every benchmark). Table 1 describes our experimental machine. We use BenchExec [13] 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 on the system.

3.4 Benchmark Subsetting

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

Following Yi et al. [76], we evaluate the benchmark subset in two different ways:

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

[6]DSK: this reminds me of VisTrails from Utah

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

1. **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 of a "non-negative" "linear regression" from the "standardized" features of selected benchmarks to the mean of features of the total set.
   - We use a "linear regression" to account for the possibility that the total set has unequal proportions of benchmark clusters. Suppose it contained 10 programs of type A, which all have similar performance, and 20 of type B: the benchmark subset need not contain two B programs and onerous A program. We would rather have one A, one B, and write the total performance as a weighted combination of the performance of A and B, perhaps $1 \cdot \mathrm{perf}_A + 2 \cdot \mathrm{perf}_B$). We normalize these weights by adding an ancillary constant feature, so $\mathrm{weight}_A + \mathrm{weight}_B + \cdots = 1$. Yi et al. [76] were attempting to subset with SPEC CPU 2006, which one can assume would already be balanced in these terms, so their analysis uses an unweighted average.
   - We require the linear regression to be "non-negative" so that the benchmark subset is monotonic; doing better on every benchmark in the subset should result in doing better on the total set.
   - "Standardized" means we transform raw features x into $z_x = (x - \bar{x})/\sigma_x$. 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 $x_{\mathrm{norm}} = x/x_{\mathrm{max}}$ 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.
2. **Representativeness.** How close are benchmarks in the original set to benchmarks in the subset? We will evaluate this by computing root mean squared error (RMSE) on the euclidean distance of standardized features from each benchmark in the original set to the closest benchmark in the selected subset.
   - We opt for RMSE over mean absolute error (MAE), used by Yi et al. [76], because RMSE punishes outliers more. MAE would permits some distances to be large, so long it is made up for by other distances are small. RMSE would prefer a more equitable distribution, which might be worse on average, but better on the outliers. We think this aligns more with the intent of "representativeness."

For features, we will use features that are invariant between running a program ten times and running it once. This gives long benchmarks and short benchmarks which exercise the same functionality similar feature vectors. In particular, we use:

- The log overhead ratio of running the benchmark in each provenance collectors. We use the logarithm of the ratio, rather than the ratio directly because the logarithm is symmetric with respect to overshooting and undershooting. Suppose $x provenance collector runs benchmark $y1 twice as fast and benchmark $y2 twice as slow; the average of the overhead would be $(2 + \frac{1}{2})/2 = 1.25$, whereas the average of the logarithms would be $\log 2 + \log \frac{1}{2} = \log 2 - \log 2 = 0$, meaning the 2x speedup "canceled out" the 2x slowdown on average). Note that the arithmetic mean of logarithms of a value is equivalent to the geometric mean of the value.
- The ratio of CPU time to wall time.
- The number of syscalls in each category per wall time second, where the categories consist of socket-related syscalls, syscalls that read file metadata, syscalls that write file metadata, syscalls that access the directory structure, syscalls that access file contents, exec-related syscalls, clone-related syscalls, exit-related syscalls, dup-related syscalls, pipe-related syscalls, close syscalls, pipe-related syscalls, and chdir syscalls.

In order to choose the subset, we will try clustering, preceded by optional dimensionality reduction. Once the benchmarks are grouped into clusters, we identify one benchmark from each of the k clusters to consist the benchmark subset. We will sweep across k. We tried the following clustering algorithms:

- **K-means.** K-means [48] greedily minimizes within-cluster variance, which is equivalent to the "representativeness" RMSE distance we want to minimize. Unfortunately, k-means can easily get stuck in local minima and needs to take the number of clusters, k, as a parameter. We use random restarts and initialize with k-means++ [8].
- **Agglomerative clustering (Ward linkage).** Agglomerative clustering [75] greedily minimizes a certain metric from the bottom up. All data points start out as singleton clusters, and the algorithm joins the "best" two clusters repeatedly. The Ward Linkage is a metric that joins the pair of clusters resulting in the smallest within-cluster variance, which is exactly what "representativeness" RMSE distance wants to minimize. Agglomerative clustering can output hierarchical clusters, which may be useful in other contexts.

Dimensionality reduction seeks transform points in a high-dimensional space to points in a low-dimensional space, while preserving some property or properties (often including pairwise distance). We experiment with no dimensionality reduction and with PCA dimensionality reduction, while sweeping on the number of target dimensions.

3.5 Performance Model

A related problem to subsetting is inferring a performance model. There are two motivations for inferring a performance model:

- A sysadmin may wish to provide a computational provenance capturing system to their institution, but getting approval to run new software on their system may be expensive (e.g., on highly secure systems, the sysadmin may need to acquire a security audit of the code before it can be approved for use). They may want to prospectively estimate the overhead of provenance collectors without having to install all the provenance collectors on their system, so they can select the optimal collector for their use-case.
- Inferring a provenance model may improve our understanding of the bottlenecks in provenance collectors.

A performance model should input features of a prospective workload and output the approximate overhead under different systems. 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 the following models:

- **Ordinary least-squares (OLS) linear regression**. We estimate the runtime of each benchmark on each provenance collector as a linear regression of the features of each benchmark, learning weights for each feature in each provenance collector using ordinary least-squares. This would create a model like $\text{weight}_1 \cdot \text{feature}_1 + \text{weight}_2 \cdot \text{feature}_2 + \cdots$ However, we can reduced its number of parameters, and thereby increase its out-of-domain generalizability, by the next two methods.
- **Low-rank linear regression.** To further reduce the number of parameters, we apply singular value decomposition (SVD) to create a lossily-compressed representation of the learned weights. TODO: describe this model
- **OLS on a subset of features.** This method proceeds like the OLS regression, except it only uses a subset of the features, ignoring the rest. This is like doing a LASSO regression, but with multiple linear predictors sharing the same set of features (LASSO is usually described as solving for just one linear predictor). Unfortunately, we do not know an efficient algorithm like ID for selecting this subset. We tried two algorithms: greedy, which picks one additional feature that decreases loss the most until it has k features, and random, which selects a random k-sized subset.

We use the same features as in Section 3.4, but with the addition of a constant term, for a provenance collectors which have a fixed startup cost.

We use cross-validation to estimate generalizability of the predictor. Cross-validation proceeds in the following manner, given n benchmarks and f features.

1. Separate the n benchmarks into αn "testing" benchmarks and $(1 - \alpha)n$ "training" benchmarks.
2. Learn to predict the log overhead ratio based on f features of each of the $(1 - \alpha)n$ training benchmarks.
3. Using the model learned in the previous step, predict the log overhead ratio on αn testing benchmarks.
4. Compute the RMSE of the difference between predicted and actual.
5. Repeat to 1 with a different, but same-sized test/train split.
6. Take the arithmetic average of all observed RMSE; this is an estimate of the RMSE of the predictor on out-of-sample data.

While cross-validation does punish model-complexity and overfitting to some extent, we will still take the number of parameters into account when deciding the "best" model in the interest of epistemic modesty. Preferring fewer parameters makes the model more generalizable on out-of-domain data, since even our full cross-validation data is necessarily incomplete.

4 RESULTS

4.1 Selected Provenance Collectors

Table 2 shows the provenance collectors we collected and their qualitative features. 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 which 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 [77]. The provenance systems we are interested in have overheads in the 1.01x – 3x range. [ˆDSK: maybe say instead that we don't use prov systems that have VM overheads over 3x?]
- **Requires recompilation.** Some provenance collectors require the user to recompile their entire application and library stack. This is prohibitively onerous and negates the otherwise low cost of switching to system-level provenance we are pursuing.
- **Requires special hardware.** Some methods require certain CPUs, e.g., Intel CPUs for a dynamic instrumention tool called Intel PIN. Being limited to certain 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, however, 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 which 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 require 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 [7]. However, we still could not get the system to run properly.
- **Needs more time (DTrace, SPADE, eBPF/bpftrace).** We simply needed more time to implement these provenance collectors.
- **Reproduced/excluded (ltrace, CDE).**
  - **ltrace**. ltrace is an off-the-shelf tool, available in most Linux package repositories. While we could run ltrace on some of our benchmarks, it crashed when processing on the more complex benchmarks. We localized the problem to the following code[8]:

```c
/* 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__);
    abort();
    return;
}
```

  - **CDE**. CDE is a research prototype proposed by Guo and Engler [30]. CDE can run some of our benchmarks, but crashes when trying to copy from the tracee process to the tracer due to `ret == NULL`[9]:

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

- **Reproduced (strace, fsatrace, rr, ReproZip, Sciunit2).** We reproduced this provenance collector on all of the benchmarks.

4.2 Implemented Benchmarks

Of these, Table 3 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 we believe have value in representing a computational science use-case.

- **HTTP/FTP servers/clients/traffic.** The most common benchmark class from prior work, 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 deprioritized implement additional FTP and HTTP clients and servers beyond the most common ones, because they would likely exhibit similar performance.
- **Compiling packages.** 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 LLVM takes more than twice as long as the longest benchmark, so we excluded LLVM specifically from the benchmark suite. We

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

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

[9] See https://github.com/usnistgov/corr-CDE/blob/v0.1/strace-4.6/cde.c#L2650. 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] URSprung depends on IBM Spectrum Scale to get directory change notifications, so it is not for a *generic* Linux system.

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

Table 2: Provenance collectors mentioned in primary and secondary studies in our search results.

Tool	Method	Status
strace	tracing	Reproduced
fsatrace	tracing	Reproduced
ReproZip [19]	tracing	Reproduced
Sciunit2 [71]	tracing	Reproduced
rr [55]	tracing	Reproduced
CDE [30]	tracing	Reproduced/excluded
ltrace	tracing	Reproduced/excluded
SPADE [26]	audit, FS, or compile-time	Needs more time
DTrace [1]	audit	Needs more time
eBPF/bpftrace	audit	Needs more time
OPUS [9]	lib. ins.	Not reproducible
CamFlow [57]	kernel ins.	Requires custom kernel
Hi-Fi [59]	kernel ins.	Requires custom kernel
LPM/ProvMon [11]	kernel ins.	Requires custom kernel
RecProv [35]	tracing	No source
LPROV [74]	kernel mod., lib. ins.	No source
S2Logger [69]	kernel mod.	No source
ProTracer [47]	kernel mod.	No source
FiPS [70]	FS	No source
PANDDE [23]	kernel ins., FS	No source
PASS/Pasta [53]	kernel ins., FS, lib. ins.	No source
PASSv2/Lasagna [52]	kernel ins.	No source
Lineage FS [62]	kernel ins.	No source
RTAG [34]	dyn./static bin. ins.	No source
BEEP [44]	dyn. bin. ins.	Requires HW
libdft [37]	dyn. bin., kernel, lib. ins.	Requires HW
RAIN [33]	dyn. bin. ins.	Requires HW
DataTracker [66]	compile-time ins.	Requires HW
MPI[46]	compile-time ins.	Requires recompilation
LDX [41]	VM ins.	Requires recompilation
Panorama [77]	VM ins.	VMs are too slow
PROV-Tracer [65]	audit	VMs are too slow
ETW [5]	audit	Not for Linux
Sysmon [50]	audit	Not for Linux
TREC [73]	tracing	Not for Linux
URSprung [60]	audit	Not for Linux[10]
Ma et al. [45]	audit	Not for Linux

Table 3: Benchmarks used in various provenance publications.

Publication	Benchmarks	Comparisons
TREC [73]	open/close, compile Apache, compile LaTeX doc	Native
PASS [53]	BLAST	Native ext2
Panorama [77]	curl, scp, gzip, bzip2	Native
PASSv2 [52]	BLAST, compile Linux, Postmark, Mercurial, Kepler	Native ext3, NFS
SPADEv2 [26]	BLAST, compile Apache, Apache	Native
Hi-Fi [59]	lmbench, compile Linux, Postmark	Native
libdft [37]	scp, tar, gzip, bzip2 x extract, compress	PIN
LogGC [43]	RUBiS, Firefox, MC, Pidgin, Pine, Proftpd, Sendmail, sshd, vim, w3m, wget, xpdf, yafc, Audacious, bash, Apache, mysqld	None[11]
LPM/ProvMon [11]	lmbench, compile Linux, Postmark, BLAST	Native
Ma et al. [45]	TextTransfer, Chromium, DrawTool, NetFTP, AdvancedFTP, Apache, IE, Paint, Notepad, Notepad++, simplehttp, Sublime Text	Native
ProTracer [47]	Apache, miniHTTP, ProFTPD, Vim, Firefox, w3m, wget, mplayer, Pine, xpdf, MC, yafc	Auditd, BEEP
LDX [41]	SPEC CPU 2006, Firefox, lynx, nginx, tnftp, sysstat, gif2png, mp3info, prozilla, yopsweb, ngircd, gocr, Apache, pbzip2, pigz, axel, x264	Native
PANDDE [23]	ls, cp, cd, lpr	Native
MPI [46]	Apache, bash, Evince, Firefox, Krusader, wget, most, MC, mplayer, MPV, nano, Pine, ProFTPd, SKOD, Tiny-HTTPd, Transmission, Vim, w3m, xpdf, Yafc	Audit, LPM-HiFi
CamFlow [57]	lmbench, postmark, unpack kernel, compile Linux, Apache, Memcache, redis, php, pybench	Native
BEEP [44]	Apache, Vim, Firefox, wget, Cherokee, w3m, ProFTPd, yafc, Transmission, Pine, bash, mc, sshd, sendmail	Native
RAIN [33]	SPEC CPU 2006, cp linux, wget, compile libc, Firefox, SPLASH-3	Native
Sciunit [71]	VIC, FIE	Native
LPROV [74]	Apache, simplehttp, proftpd, sshd, firefox, filezilla, lynx, links, w3m, wget, ssh, pine, vim, emacs, xpdf	Native
MCI [42]	Firefox, Apache, Lighttpd, nginx, ProFTPd, CUPS, vim, elinks, alpine, zip, transmission, lftp, yafc, wget, ping, procps	BEEP
RTAG [34]	SPEC CPU 2006, scp, wget, compile llvm, Apache	RAIN
URSPRING [60]	open/close, fork/exec/exit, pipe/dup/close, socket/connect, CleanML, Vanderbilt, Spark, ImageML	Native, SPADE

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 try compiling Python, Boost, HDF5, glibc, Apache HTTPd, and Perl.[12]

- **Browsers.** 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.
- **Un/archive.** 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.
- **I/O microbenchmarks (lmbench, postmark, custom).** These could be informative for explicating which I/O operations are most affected. Prior work uses lmbench [51], which focuses on syscalls generally, Postmark [36], which focuses on I/O operations, and custom benchmarks, for example running open/close in a tight loop. We use the specific lmbench cases from prior work, which is mostly the latency benchmarks with a few bandwidth benchmarks. Most provenance systems do not affect the bandwdith; it doesn't matter *how much* this process writes to that file, just *that* this process wrote to that file.
- **BLAST.** BLAST [7] is a search for a fuzzy string in a protein database. Many prior works use this as a file-read heavy benchmark, as do we. This code in particular resembles a computational science workload. However, unlike prior work, we split the benchmark into each of each subtasks; provenance may have a greater overhead on certain subtasks.
- **CPU benchmarks.** SPEC CPU INT 2006 [31] and SPLASH-3 [61] test CPU performance. 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. However, SPEC CPU INT 2006 is not free (as in beer), so we could only implement SPLASH-3.
- **Sendmail.** 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.

[12]TODO: Double check what are we compiling? Also update the table below, once that is nailed down.

Table 4: Benchmarks implemented by this work.

Prior works	This work	Benchmark group and examples from prior work
10	yes (5/7 servers)	HTTP server/traffic (Apache httpd, miniHTTP, simplehttp, lighttpd, Nginx, tinyhttpd, cherokee x apachebench)
9	yes (2/4 clients)	HTTP serer/client (simplehttp x curl, wget, prozilla, axel)
8	yes (3/5 orig + 4 others)	Compile user packages (Apache, LLVM, glibc, Linux, LaTeX document)
8	no	Browsers (Firefox, Chromium x Sunspider)
6	yes (1/6) + 2 others	FTP client (lftp, yafc, tnftp, skod, AdvancedFTP, NetFTP)
5	yes	FTP server/traffic (ProFTPd x ftpbench)
5	yes	Un/archive (compress, decompress x tar x nothing, bzip2, pbzip, gzip, pigz)
5	yes	I/O microbenchmarks (Postmark, lmbench, custom)
4	yes	BLAST
3	yes (1/2)	CPU benchmarks (SPEC CPU INT 2006, SPLASH-3)
3	yes	Coreutils and other utils (bash, cp, ls, procps)
2	no	Sendmail
1	yes	VCS checkouts (Mercurial)
1	no	Machine learning workflows (CleanML, Spark)
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	ImageML
1	no	ping
1	no	mp3info
1	no	ngircd
1	no	CUPS

- **VCS checkouts.** VCS checkouts are a common computational science operation. Prior work uses Mercurial, but we implemented Mercurial and Git. We simply clone a repository (untimed) and run `$vcs checkout` for random commits in the repository (timed).
- **Data processing/machine-learning Workflows.** VIC, FIE, ImageML, and Spark are real-world examples of scientific 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 [14] which FIE supports to understand the workflow, and write our own script which glues the operations together.
- **Utilities (bash, cp, ls, procps).** We did not see a huge representative value in these benchmarks that would not already be gleaned from lmbench, but due to its simplicity, we implemented it anyway. For `bash`, we do not know what workload prior works are using, but we test the speed of incrementing an integer and changing directories (`cd`).
- The rest of the 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.

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.

Table 5: 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 new execution takes 1% longer than the old. "Noprov" refers to a system without any provenance collection (native), for which the slowdown is 0 by definition. fsatrace appears to have a negative slowdown in some cases due to random statistical noise.

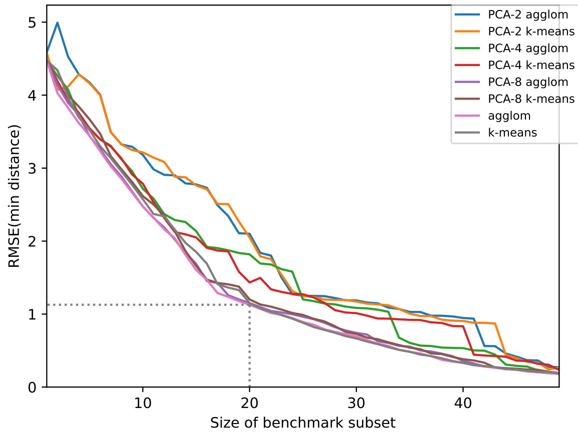
collector benchmark types	fsatrace	noprov	reprozip	rr	strace
archive	7	0	164	208	180
blast	-1	0	32	102	6
copy	48	0	7299	322	710
ftp client	-0	0	14	5	4
ftp server	1	0	58	-32	65
gcc	3	0	417	314	321
http client	-16	0	453	200	98
http server	6	0	791	965	516
lmbench	-14	0	31	15	5
notebook	-10	0	116	0	50
pdflatex	-10	0	290	19	79
postmark	9	0	2002	367	928
python	0	0	412	137	184
shell	18	0	4620	698	63
simple	23	0	977	1749	431
splash-3	-1	0	78	64	19
unarchive	6	0	179	190	177
vcs	3	0	453	169	185

4.3 Subsetted Benchmarks

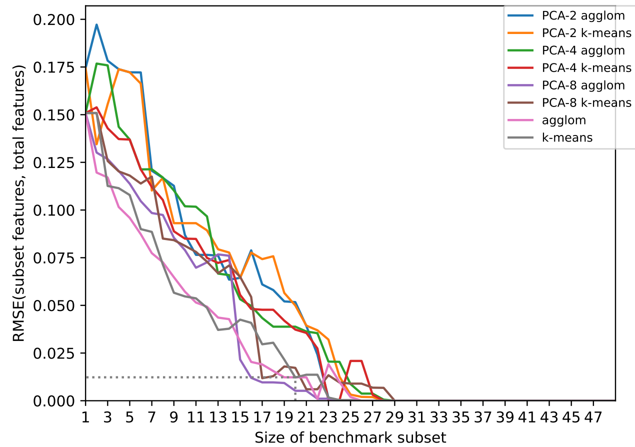
Figure 2 shows the performance of various algorithms on benchmark susbetting. We observe:

- The features are already standardized, so PCA has little to offer other than rotation and truncation. However, the truncation is throwing away potentially useful data. Since we have a large number of benchmarks, and the space of benchmarks is quite open-ended, the additional dimensions that PCA trims off appear be important for separating clusters of data.
- K-means and agglomerative clustering yield nearly the same results. They both attempt to minimize within-cluster variance, although by different methods.
- RMSE of the residual of linear regression will eventually hit zero because the k exceeds the rank of the matrix of features by benchmarks; The linear regression has enough degrees of freedom to perfectly map the inputs to their respective outputs.

It seems that agglomerative clustering with $k = 20$ has quite good performance, and further increases in k exhibit diminishing returns. We examine the generated clusters and benchmark subset in Figure 4 and Section 4.3.

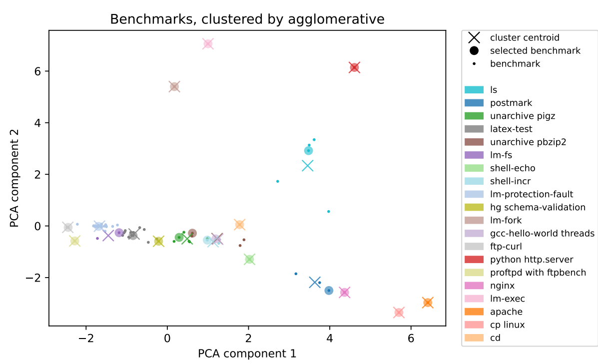


(a) Subsetting algorithms scored by the RMSE of the distance of each benchmark to the nearest selected benchmark. A dotted line shows the x- amd y-value of the point of diminishing return.

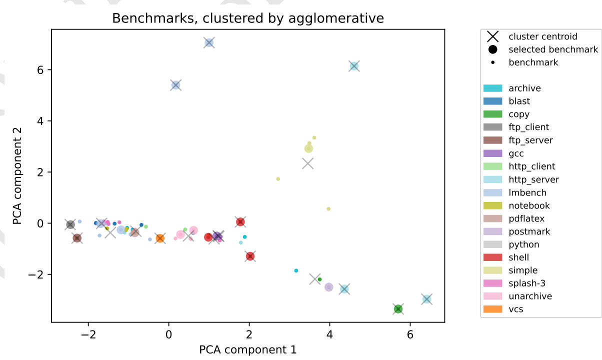


(b) Subsetting algorithms scored by the RMSE of the difference between (weighted) features of the subset and features of the original set. A dotted line shows the x- amd y-value of the point of diminishing return.

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



(a) Benchmark subset, where color shows a posteriori agglomerative clusters. Each cluster conceptually represents the same-color benchmarks being represented by a single benchmarks, which gives its name to the cluster.



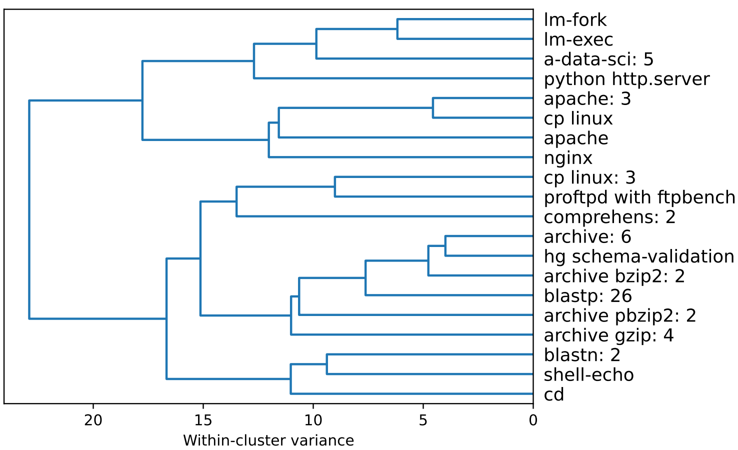
(b) Benchmark subset, where color shows a priori benchmark "type" (see Table 4).

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.

Section 4.3 shows the a posteriori clusters with colors. Section 4.3 shows a priori benchmark "types", similar but more precise than those in Table 4. From these two, we offer the following observations:

- 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 radial distance to each star.
- Many of the clusters are singletons, for example the `python http.server` near $(5, 6)$; this is surprising, but given there are not any other points nearby, it seems reasonable.
- We might expect that benchmarks of the same type would occupy nearby points in PCA space, but it seems they often do not. lmbench is particularly scattered with points at $(-1, 0)$ and $(0, 5)$, perhaps because it is a microbenchmark suite where each microbenchmark program tests a different subsystem.

To elucidate the structure of the clusters, we plotted a dendrogram (Section 4.3) and listed the members of each cluster (Section 4.3). We offer the following observations:



(a) Dendrogram showing the distance between clusters. We label each cluster by their "selected benchmark". If there is a colon and a number after the name, it indicates the number of benchmarks contained in that cluster. Otherwise, the cluster is a singleton.

Cluster representative	Weight (%)	Cluster members
ls	1.7	echo, hello, ps, true
postmark	0.0	archive, cp smaller
unarchive pigz	26.4	git setuptools_scm, python-hello-world, unarchive, unarchive gzip, wget
latex-test	13.0	archive bzip2, archive gzip, blastn, comprehens, curl, latex-test2, lm-fstat, lm-stat, minihttp, python-import, titanic-da, unarchive bzip2
unarchive pbzip2	7.9	archive pbzip2, archive pigz, lighttpd
lm-fs	3.9	lm-open/close
shell-echo	1.5	
shell-incr	3.1	splash-fft
lm-protection-fault	24.2	a-data-sci, blastp, blastx, lm-bw_file_rd, lm-bw_pipe, lm-bw_unix, lm-catch-signal, lm-getppid, lm-install-signal, lm-mmap, lm-page-fault, lm-read, lm-select-file, lm-select-tcp, lm-write, megabl, splash-cholesky, splash-lu, splash-ocean, splash-radiosity, splash-radix, splash-volrend, splash-water-nsquared, splash-water-spatial, tblast
hg schema-validation	0.0	
lm-fork	0.1	
gcc-hello-world threads	0.7	gcc-hello-world
ftp-curl	2.4	ftp-wget, lftp
python http.server	1.3	
proftpd with ftpbench	2.5	
nginx	1.5	
lm-exec	3.7	
apache	1.4	
cp linux	2.0	
cd	1.4	
all	98.8	

(b) A table showing cluster membership and weights. 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. Since these are coefficients, not proportions, the result need not add to 100%, although we insert an equation which should guide the solution there.

Figure 4: Figures showing the relationships between clusters and the members of each cluster.

- Fork and exec are close in feature-space, probably because programs usually do both.
- cd and shell-echo are near each other. I is surprising that blastn is also near cd and shell-echo, but they both have similar cputime-to-walltime ratios.
- Many of the CPU-heavy workloads are grouped together, under lm-protection-fault.
- Many of the un/archive benchmarks are grouped together with lighttpd, which also accesses many files.

4.4 Predictive Model

Figure 5 shows us the competition between predictive performance models. Note that linear regression does not permit sweeping over the number of parameters; it requires a $n_{\text{benchmarks}} n_{\text{features}}$ parameters. Matrix factorization methods use only $(n_{\text{benchmarks}} - k) \times (n_{\text{features}} - k) = n_{\text{benchmarks}} n_{\text{features}} - k(n_{\text{benchmarks}} + n_{\text{features}}) + k^2$ parameters. When k is low, matrix factorization is much fewer parameters than linear regression at the cost of some in-sample accuracy, but when k approaches n_{features}, it is less parameter-efficient than linear regression. Number of parameters is not truly an independent variable that can be directly swept over. Rather k is an independent variable, we sweep over k, and plot the number-of-parameters on the x-axis, since that is more directly interpretable. Models with a large number of parameters are more likely to overfit to spurious correlations on the test sample which generalize poorly on the train sample. Overgeneralization is appropriately punished by cross-validation.

We observe the following:

- When the number of parameters is large, all of the algorithms preform similarly; Even though greedy feature selection is more constrained than low-rank matrix factorization (every solution found by greedy is a candidate used by low-rank,

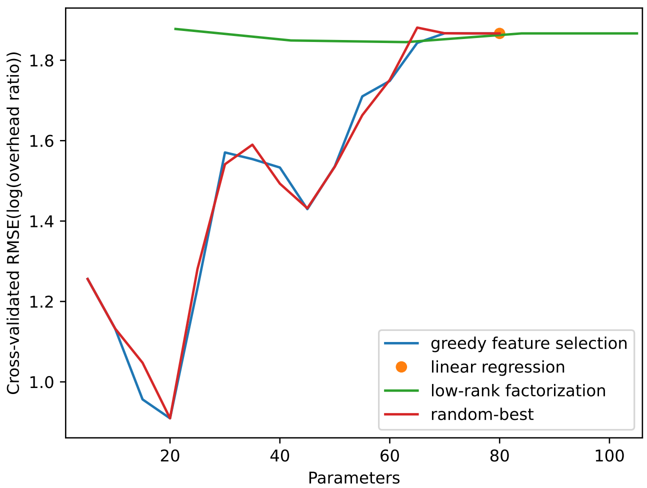


Figure 5: Competition between predictive performance models.

	metadata-reads per walltime second	constant fraction	cputime / walltime	execs-and-forks per walltime second
fsatrace	0.000003	-0.001236	-0.024958	0.000064
nnnoprov	0.000000	0.000000	0.000000	0.000000
reprozip	0.000043	-0.027311	0.266969	0.000438
rr	0.000021	-0.011208	0.404307	0.000878
strace	0.000029	-0.002243	0.229129	0.000312

Table 6: Linear regression, using benchmark subset to approximate the original benchmark.

but not vice versa), there are enough degrees of freedom to find similar enough candidates.

- Linear regression has equivalent goodness-of-fit to matrix factorization with a high k, as expected. When the compression factor is low, the compressed version is just as good as the original.
- Random-best usually does not do better than greedy feature selection. However, greedy is much easier to compute, evaluating n_{features} subsets of size 1, $n_{\text{features}} - 1$ subsets of 2, ... $n_{\text{features}} - k + 1$ subsets of size k; random has to evaluate a large number (1000 in our case) of subsets of size k. Greedy is not necessarily optimal, since a set of features may be individually outscored by other features but may have predictive value when taken as a set. Greedy would never pick that set, because it is bound to pick the best additional individual feature at every step, but random-best could. However, our problem domain may lack the complexity to generate these cases.

Greedy feature selection with 20 parameters (predicting the performance on 5 systems using only $k = 4$ of 16 features) seems to preform the best in cross-validation. On 19 out of 20 cross-validation splits, greedy feature selection with $k = 4$ chose the parameters in Table 6.

For example to estimate the overhead of fsatrace, we would use the first row of Table 6,

$$\log \frac{\text{walltime}_{\text{fsatrace}}}{\text{walltime}_{\text{noprov}}} = 3 \times 10^{-6} \left(\frac{\text{metadata reads}}{\text{walltime}_{\text{noprov}}}\right) - 0.001 \cdot \left(\frac{1}{\text{walltime}_{\text{noprov}}}\right) - 0.02 \cdot \left(\frac{\text{cputime}_{\text{noprov}}}{\text{walltime}_{\text{noprov}}}\right) + 6 \times 10^{-5} \cdot \left(\frac{\text{execs and forks}}{\text{walltime}_{\text{noprov}}}\right)$$

The system calls features can be observed using strace. The CPU time and wall time of noprov can be observed using GNU time. One need not complete an entire execution to observe the these fatures; one merely needs to record the features until they stabilize (perhaps after several iterations of the main loop).

4.5 Threats to Validity

5 FUTURE WORK

In the future, we plan to implement compilation for more packages, in particular xSDK [10] packages.

6 CONCLUSION

We hope this work serves as a part of a bridge from research to practical use of provenance collectors. As such, we address practical concerns of a user wanting to use a provenance collector. We identify the reproducible and usable provenance collectors from prior work, and we evaluate their performance on synthetic and real-world workloads.

A 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, 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/depaul-dice/sciunit/pull/36
- https://github.com/ipython-contrib/jupyter_contrib_nbextensions/pull/1649
- https://github.com/NixOS/nixpkgs/issues/268542

B 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] David Arthur and Sergei Vassilvitskii. 2007. K-Means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*. Society for Industrial and Applied Mathematics, USA, 1027–1035.

[9] 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)*.

[10] 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

[11] 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.

[12] 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.

[13] 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

[14] 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

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

[16] 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

[17] 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

[18] 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.

[19] 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

[20] 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

[21] Fred D. Davis. 1985. *A Technology Acceptance Model for Empirically Testing New End-User Information Systems: Theory and Results*. Thesis. Massachusetts Institute of Technology.

[22] 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.

[23] 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

[24] Juliana Freire, David Koop, Emanuele Santos, and Cl 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

[25] 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.

[26] 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

[27] 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.

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

[29] 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

[30] 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.

[31] 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

[32] 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.

[33] 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

[34] 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.

[35] 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

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

[37] 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

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

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

[40] 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.

[41] 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

[42] 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

[43] 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

[44] 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*.

[45] 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

[46] 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.

[47] 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

[48] J Macqueen. 1965. Some Methods for Classification and Analysis of Multivariate Observatiosn. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1. University of California Press, Los Angeles, CA, 281.

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

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

[51] 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.

[52] 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

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

[54] 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.

[55] 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]

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

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

[63] 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.

[64] 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

[65] 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.

[66] 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

[67] 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.

[68] 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.

[69] 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

[70] 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

[71] 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

[72] 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

[73] 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.

[74] 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

[75] Joe H. Ward Jr. 1963. Hierarchical Grouping to Optimize an Objective Function. *J. Amer. Statist. Assoc.* 58, 301 (March 1963), 236–244. https://doi.org/10.1080/01621459.1963.10500845

[76] 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

[77] 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

[78] 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