# Provenance tracing unmodified binaries in unprivileged mode with low overhead

*(Anonymous)*

## Abstract

System-level provenance tracing is the idea of automatically capturing how computational artifacts came to be, including what process created each file. While provenance is often discussed in the context of security, it also fills an important role in computational science, providing data for reproducibility, incremental computation, and debugging. Prior work proposes recompiling with instrumentation, using ptrace, or configuring kernel-based auditing, which at best achieves two out of three desirable properties: accepting unmodified binaries, running in unprivilege mode, and incurring low overhead.

We present PROBE, a system-level provenance tracer that uses library interposition to achieve all three. We evaluate the performance of PROBE on system microbenchmarks and scientific applications. PROBE has a less than 10% overhead on real-world scientific applications compared to 20% with the best prior, unprivileged provenance tracers for unmodified binaries. We also discuss the completeness of the provenance that PROBE collects compared to other provenance tracers.

## 1 Introduction

For computational artifacts, computational provenance (henceforth **provenance**) refers to the process which generated the artifact, the inputs to that process, and the provenance of those inputs. This definition permits a graph representation where the artifacts and processes become nodes; an edge indicates that an artifact was generated by a process or that a process used some artifact (for example, Fig. 1).

Provenance data has many applications including: **reproducibility**, documenting the processes needed to recompute a particular artifact, **comprehension**, visualizing how data
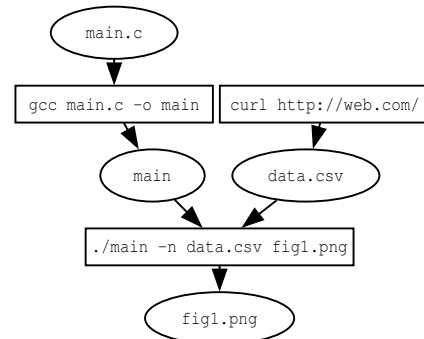


Figure 1: Example provenance graph of `fig1.png`. Artifacts are ovals; processes are rectangles.

flows in a pile-of-scripts, and **differential debugging**, determining how the process behind two computational outputs differ.

The reproducibility use-case is especially compelling due to the reproducibility crisis in computational science [20, 13, 10]. There are other approaches for improving reproducibility such as virtualized environments, package managers, and workflows, but these all involve tedious participation from the user. If a system can track provenance automatically, the user need only install or turn on this feature, continue their computational science experiments, and the system would keep track of how to reproduce each artifact *automatically*.

One barrier to adoption is the significant performance overhead imposed by collecting provenance data. User-level tracing involves asking the kernel to switch over every time the program being traced (henceforth, **tracee**) does a specific kind of action. However, context switching imposes a significant

overhead.

Prior provenance tracers avoid the overhead of context switching in two ways: by embedding themselves in the kernel or by embedding themselves in user code. Embedding in the kernel is problematic because it increases the attack surface, it is difficult to maintain, and many computational scientists will not have root-level access to the shared machines they operate on. Embedding in user code could happen at compile-time, but that involves the added burden of re-compiling all user code, including code that the user may not have compiled at all originally in the first place (due to binary package managers). Embedding could happen through binary rewriting, on the other hand, is difficult because the syscall instructions are smaller than the necessary instructions to facilitate a function call in x86 [30]. Another solution for could be to embed provenance tracing in library code at load-time by library interposition. This technique requires neither root-level access, recompiling user code, nor extra context-switching on every event.

Prior work on provenance tracing argues that library interposition is too incomplete, difficult, or fragile for their use-case (see sec. 7). We offer a counter-argument in the form of an implementation of a provenance tracer based on library interposition.

The contributions of this work are:

- a delineation of desirable properties of provenance tracers

- an implementation of a provenance tracer based on library interposition called PROBE: **P**rovenance for **R**eproducibility **OB**servation **E**ngine

- a suite of applications that consume its provenance, demonstrating the practical utility of provenance recorded in PROBE

- theoretical and empirical evaluation of the performance, completeness, and fragility of selected provenance tracers

The rest of the work proceeds as follows:

Sec. 2 defines kinds of provenance tracers, properties thereof, and use-cases thereof. Sec. 3 enumerates prior provenance tracers and discusses their properties. Sec. 4 documents the design of PROBE and its related applications. Sec. 5 discusses how provenance data can be used for real-world benefit and the applications we developed for PROBE. Sec. 6 presents an theoretical and empirical evaluation of selected provenance tracers. Sec. 7 is a general discussion of the evaluation results in the context of prior work. Sec. 8 discusses directions for future research with PROBE. Sec. A discusses the formal semantics of the terms used in PROBE and discusses its soundness as a system.

## 2  Background

Provenance can be collected at several different levels [8], which generally trade off less semantic value for higher generality or vice versa [12].

**Application-level** provenance the most semantically rich but least general, as it only enables collection by that particular modified application. **Language/workflow-level provenance** is less semantically rich but more general, as it enables provenance collection in any workflow or program written for that modified workflow engine or programming language. **System-level provenance** is the least semantically aware because it does not even know dataflow, just a history of inputs and outputs, but it is the most general, because it supports any process (including any application or workflow engine) that uses watchable I/O operations.

Operating system-level provenance tracing (henceforth **SLP**) is the most widely applicable form of provenance tracing; install an SLP, and all unmodified applications, programming languages, and workflow engines will be traced. This work focuses on system-level provenance (SLP).

Provenance has a number of use-cases discussed in prior work:

- **Reproducibility** [5] (manual or automatic). Provenance tracing aids manual reproducibility because it documents what commands were run to generate the particular artifact. While this can also be accomplished by documentation or making the structure of the code "obvious", in practice we accept it as an axiom that there are many cases where the authors don't have enough documentation or obvious structure to easily understand how to reproduce the artifact.

  Provenance could aid in *automatic* reproducibility, automatically replaying the processes with their recorded inputs, or *manual* reproducibility, showing the user the commands that were used and letting them decide how to reproduce those commands in their environment.

- **Comprehension** [21]. Provenance helps the user understand the flow of data in a complex set of processes, perhaps separately invoked. A tool that consumes provenance can answer queries like: "Does this output depend on FERPA-protected data (i.e., data located `/path/to/ferpa`)?".

- **Differential debugging** [21]. Given two outputs from two executions of similar processes with different versions of data and code or different systems, what is the earliest point that intermediate data from the processes diverges from each other?

- **Incremental computation** [29]. Iterative development cycles from developing code, executing it, and changing the code. Make and workflow engines require user to

specify a dependency graph (prospective provenance) by hand, which is often unsound in practice; i.e., the user may miss some dependencies and therefore types `make clean`. A tool could correctly determine which commands need to be re-executed based on SLP without needing the user to specify anything.

- **Intrusion-detection and forensics** [21] Provenance-aware systems track all operations done to modify the system, providing a record of how an intruder entered a system and what they modified once they were in. Setting alerts when certain modifications are observed in provenance forms the basis of intrusion detection.

The first four are applicable in the domain of computational science while the last is in security. This work focuses on provenance tracers for computational science.

We define the following "theoretical" properties of provenance tracers. They are theoretical in the sense that one does not need to do any experiments to determine these properties; only study their methods and perhaps their code. They are enumerated in a feature matrix in Table 1.

- **Runs in user-space**: SLP should run in user-space as opposed to kernel-space. Kernel modifications increase the attack surface and is more difficult to maintain than user-space code.

- **No privilege required**: A user should be able to use SLP to trace their own processes without accessing higher privileges than normal every time. Two motivations for this property are that code running in privileged mode increases the attack surface and presents a barrier to use for non-privileged users. Computational scientists would likely not have root-level access on shared systems and thus may not be able to use SLPs that require privilege to run. We do not distinguish between privileges required to install versus privileges required to run, since they are equivalent by setuid.

- **Ability to run unmodified binaries**: Users should not have to change or recompile their code to track provenance.

- **Not bypassable**: A tracee should not be able to read data in a way that will bypass detection by the provenance tracer.

- **Records data and metadata**: SLP tracers always record the metadata, e.g., which file was accessed. Some also record the data in the file that was accessed, at the cost of higher overhead. One could encompass the advantages of both groups by offering a runtime option to switch between faster/metadata-only or slower/metadata-and-data.

- **Replayable**: The SLP tool should export an archive that can be replayed. The replay may be mediated by the SLP tool itself or by an external tool, e.g. Docker, VirtualBox, QEMU. Recording data and metadata is required for replay.

- **Replay supports deviations**: The replay supports executing a different code path in the reconstructed environment, so long as the different code path does not access any files outside of those the original code path accessed (those are already in the reconstructed environment). For example, replay the recorded execution but replace one command-line flag, environment variable, or input file.

- **Constructs provenance graph**: The SLP tool should construct and export a graph representation of the provenance from the observed log of provenance events. Certain use-cases such as incremental computation, comprehension, differential debugging, and others require the graph representation while merely replaying does not. Constructing the graph from a log of events is difficult due to concurrency in the system.

## 3 Prior work

There have been several methods of tracing SLP proposed in prior work:

- **Virtual machines**: running the tracee in a virtual machine that tracks information flow. This method is extremely slow; e.g., PANORAMA has 20x overhead [31].

- **Recompiling with instrumentation**: recompile, where the compiler or libraries insert instructions that log provenance data, e.g., [18]. This method does not work with unmodified binaries.

- **Static/dynamic binary instrumentation**: either before run-time (static) or while a binary is running (dynamic) change the binary to emit provenance data [17]. These methods requires special hardware (e.g., Intel CPU), a proprietary tool (e.g., Intel PIN), and often root-level access (as Intel PIN does).

- **Kernel modification**: modify the kernel directly or load a kernel module that traces provenance information, e.g., [23]. This method is not in user-space.

- **Use kernel auditing frameworks**: use auditing frameworks already built in to the kernel (e.g., Linux/eBPF, Linux/auditd, Windows/ETW). This method is not unprivileged.

- **User-level debug tracing**: use user-level debug tracing functionality provided by the OS (e.g, Linux/ptrace used by `strace`, CDE [11], SciUnit [25], Reprozip [5], RR [22]).

- **Library interposition**: replace a standard library with an instrumented library that emits provenance data as appropriate. This could use the `LD_PRELOAD` of Linux and `DYLD_INSERT_LIBRARIES` on MacOS.

If unprivileged execution, unmodified binaries, and low performance overhead are hard-requirements, the only possible methods are user-level debug tracing and library interposition.

Using the results of a recent literature survey [9], we identify the user-level tracing provenance collectors: PTU [25], Sciunit [28], and ReproZip [5]. We did not identify any feasible library interposition provenance collectors. OPUS [4] is one example, but we were not able to replicate it. It was last developed almost a decade ago, and it uses end-of-life Python 2.7 and Java 1.6.

We also selected the following record/replay tools, which do not claim to be provenance tracers, but involve tracing provenance events with user-level tracing: CDE [11], CARE [14], and RR [22]. If they are performant enough, perhaps they could be converted to provenance tracers.

Lastly, we selected `strace`. `strace` captures the relevant provenance events with user-level debug tracing but does not copy data nor create a graph. We expect `strace` to be the upper-bound on user-level debug tracing performance, because it does not do any processing on the events it observes.

We examine the properties of these in Tbl. 1.

In user-level debug tracing, the tracer runs in a separate process than the tracee. Every time the tracee does a system call, control switches from the tracee to the kernel to the tracer and back and back Fig. 2. This path incurs two context switches for every system call.

O'Callahan et al. mitigate this by "inject[ing] into the recorded process a library that intercepts common system calls, performs the system call without triggering a ptrace trap, and records the results to a dedicated buffer shared with RR [the tracer program]" [22]. However, there are some system calls that RR cannot handle solely in the tracee's code, and those system calls will still cause two context switches.

On the other hand, in library interposition, the tracer code is part of a library dynamically loaded into the tracee's memory space. While this imposes restrictions on the tracer code, it obviates context switching on every provenance event Fig. 2.

## 4 Design

Even the most feature complete provenance tracers in Tbl. 1 use user-level debug tracing which can have a large overhead. Therefore, we set out to build a provenance tracer that offers similar features to the others in Tbl. 1 but uses library interposition for evaluation in Sec. 6.

Our provenance tracer is called PROBE: **P**rovenance for **R**eplay **OB**servation **E**ngine. PROBE has a recording phase, a transcription phase, and an exporting phase.
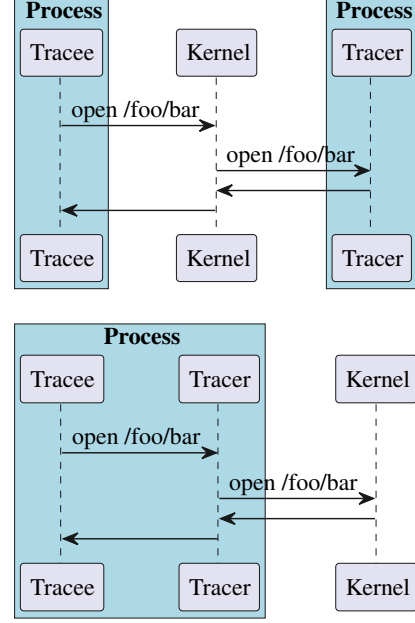


Figure 2: Sequence diagram of process with user-level debug tracing (top) and library interposition (bottom)

Users can **record** any shell command by writing `probe record` in front of the shell command. In the recording phase, PROBE will set `LD_PRELOAD` to load `libprobe.so` ahead of the systems usual C library and run the user's provided command

The core of PROBE is a library interposer for libc, called `libprobe.so`. `libprobe.so` exports wrappers for I/O functions like `open(...)`. The wrappers:

1. log the call with arguments

2. record the state of the system (e.g., during `open(..., O_RDWR)`, PROBE will make a copy of the file target)

3. forward the call to the *true* libc implementation

4. log the underlying libc's returned value

5. return the underlying libc's returned value

There is no data shared between threads as the log is thread-local. To make logging as fast as possible, the log is a memory-mapped file. If the logged data exceeds the free-space left in the file, `libprobe.so` will allocate a new file big enough for the allocation.

The **transcription phase** is run after the user's command terminates. In the transcription phase, PROBE combines the logged data into a single object that can be inserted into the provenance store or copied as a file to another host.

The PROBE log is conceptually a list of processes, where each process has a list of threads, and each thread has an ordered list of provenance events. An event may be reading a

4

| Name | User | No priv. | Unmod. bin. | Bypassable | Data & metadata | Replayable | Replay new exe. | Prov. graph |
|---|---|---|---|---|---|---|---|---|
| PROBE | yes | yes | yes | no | Optional | yes | yes | yes |
| ReproZip | yes | yes | yes | yes | Both | yes | yes | yes |
| PTU | yes | yes | yes | yes | Both | yes | yes | yes |
| CDE, Sciunit, CARE | yes | yes | yes | yes | Both | yes | yes | no |
| rr | yes | yes | yes | yes | Both | yes | no | no |
| strace, ltrace | yes | yes | yes | yes | Metadata | no | no | no |

Table 1: Feature matrix of provenance collectors and the properties described above. See Section 2 for explanation of the properties.

file, forking another process, executing a new process (exec syscall), or waiting on a process. By carefully analyzing the log of events, PROBE can construct a dataflow graph for the system. The semantics of events and their conversion to a graph is described more thoroughly in Sec. A. The dataflow can be inserted in the user's system-wide provenance store, allowing PROBE to track provenance of objects involved in multiple recordings.

In the **exporting phase**, PROBE or another program analyzes the provenance graph from the provenance store and produces some result or artifact.

The **provenance store** is implemented as a pair of SQLite tables: one table containing metadata regarding a file-version and another containing metadata regarding processes. Each process in the **process table** is associated with zero or one *parent processes*. The graph whose nodes are processes and edges point to the parent process is a tree.

Each file-version in the **file-version table** is associated with inodes (many-to-one) but zero or more paths due to symlinks and hardlinks. The assocation from file-version to inode is not invertible, since one multiple versions of one inode may have been observed. Each file-version is also associated with exactly one *creating process*, and zero or more *reading processes*, i.e., processes that are known to read the file. The dataflow graph is simply the graph whose nodes are processes and files and whose edges show creation or reading (see Sec. A). By its nature, it is acyclic and bipartite.

Finally, we developed wrapper programs for **scp** and **ssh**. One may want to run processes on remote hosts, usually because the remote has more computational resources or is "closer" to the data. Such an operation would usually be a boundary past which provenance cannot be tracked. However, we worked around this problem by deploying executables on the $PATH called ssh and scp ahead of the true ssh and scp. Our programs examine the arguments and wrap the true ssh and scp in a *provenance aware* manner.

- For ssh, we send a copy of PROBE to the remote, if it does not already have one, run the user's command in a shell with PROBE recording turned on, and insert the provenance collected on the remote to the provenance store.

- For scp, we send or receive the files as expected, but we

also attempt to send or receive the provenance associated with those files from the remote's provenance store.

## 5 Applications of provenance data

We developed applications that consume PROBE provenance to:

1. demonstrate PROBE collects "enough" provenance data for practical uses

2. motivate "always on" provenance tracers, which in turn motivates minimizing the overhead

Our applications include:

1. Visualizing the dataflow graph

2. Automatic Makefile or workflow conversion

3. Exporting a OCI or Docker container for re-execution

### 5.1 Visual representations

Some users may have a complex pile of scripts or spaghetti code that is difficult to analyze statically. If they simply run their scripts in PROBE, PROBE observes their provenance and can render it graphically.

TODO: Details on how we got this representation
TODO: Example graph

### 5.2 Automatic workflow conversion

A **workflow** is a directed acyclic graph[1] where each node represents a program and each edge is a data item, usually a file. Workflow systems like Galaxy [27], Snakemake [16], and Nextflow [7] are commonly used in domains such as bioinformatics, machine learning, and data science. Workflows are advantageous because:

- It may be easier for non-experts than Python

- Workflow languages are specialized for gluing together existing components

---

[1]There are some exceptions where the graph may be cyclic. TODO

- The workflow structure exposes parallelism, and many engines support distributed computing

- Many workflow engines implement incremental computation, so if one node changes, only the downstream artifacts need to be recomputed

However, it can be challenging to migrate from an *ad hoc* process or a pile of scripts to a structured workflow.

PROBE solves this problem by converting an *ad hoc* process to a structured workflow automatically. Users need only execute their process once by hand in PROBE, which captures the provenance. Then the user asks PROBE to export a workflow that will contain the commands used to write a particular output. PROBE supports generating Nextflow and Makefile workflows. Now, users can more easily switch to workflows and gain the benefits noted above.

## 5.3 Exporting a container

Containers are useful for automatically distributing a software environment. However, sometimes important code environments are not containerized. Therefore, containerization must be a non-trivial amount of work that some users, especially non-experts, do not have time for.

PROBE reduces the barrier to export containers by automatically containerizing the software environment. One simply runs their code in PROBE. PROBE collects the provenance of their process. We reconstruct a minimal, portable OCI image based on a provenance log. These files are then transferred into a new container's filesystem, and the container's configuration is set to replicate the original process's command, environment, and working directory. Finally, the container image is committed and optionally pushed to the Docker daemon for immediate use.

```
$ probe record ./run_script.sh
hello world

$ probe export docker-image my-image:1.0.0

$ docker run my-image:1.0.0
hello world
```

## 6 Empirical evaluation

We collected the benchmarks used in prior work in Tbl. 2. We could not find publications discussing the recording performance of ReproZip or CDE. Regarding CDE, Guo and Engler state "We have heard that ptrace interposition [the method used by CDE] can cause slowdowns of 10X or more, but we have not yet performed a rigorous performance stress test" [11]. Between the publications which do contain benchmarks, there is no overlap.

| Prov tracer | Gmean | Benchmarks used for recording |
|---|---|---|
| RR [22] | 1.58 | cp, compile, JavaScript, Firefox, Samba server |
| PTU [25] | 1.25 | geospatial task, natural language task |
| Sciunit [28] | 1.37 | hydrology task, data science task |

Table 2: Benchmarks used in publications on selected provenance tracers.

Grayson et al. give a representative benchmark suite based on benchmarks used in other provenance works including [9]:

- BLAST (multiomics application) [2] with a predefined set of queries [6]

- Apache under synthetic load

- lmbench (synthetic benchmark for I/O bandwidth and latency) [19]

- Postmark (synthetic benchmark for small file I/O) [15]

- Shell utilities in a tight loop

The performance of provenance tracing depends greatly on the ratio of provenance operations to other operations. Synthetic benchmarks that stress one subcomponent of a system (e.g., a million file reads with nothing else) will have virtually unaffected performance or significantly affected performance, depending on the subcomponent, and may not be representative of real-world performance. Therefore, we separate the synthetic benchmarks separately from the real applications in Tbl. 3.

BLAST is the only real-world application in the Grayson benchmark suite, so we added the following:

- Data science Jupyter notebooks from Kaggle.com, sorted by most votes. These notebooks read some data, create plots, and output a predicted dataset.

- Projects from Astrophysics Source Code Library [1], sorted by citations on OpenCitations [24]. From this set, we chose Quantum Espresso (calculates atomic properties) and SExtractor (extracts sources in an astronomical image).

- Projects from the Journal of Open Source Software, sorted by citations on OpenCitations [24]. From this set, we chose UMAP (data mining) and hdbscan (data mining).

strace merely logs the events but does not do any processing on the events (e.g., constructing a provenance graph). Therefore, strace is close to the theoretical minimal overhead for bare user-level debug tracing. Still, the overhead

of `strace` is significantly worse than PROBE. We enable logging on those syscalls relevant for provenance to make a fair comparison between `strace` and PROBE.

## 7 Discussion

Prior works [22, 23] argue that library interposition is not appropriate for SLP:

> In practice, we have found that method [dynamic linking to interpose wrapper functions] to be insufficient, due to applications making **direct system calls**, and **fragile**, due to variations in C libraries, and applications that require their own preloading [37, 3].

— O'Callahan et al. [22], emphasis ours

**Bypassability from direct syscalls**: While it is true that library interposition is bypassable by direct system calls, those cases are rare in practice. Syscall numbers are different on every operating system; however, `libc` call signatures are standardized in ANSI C and, with greater specificity, in POSIX. Source code written to call `libc` functions can be compiled on a variety of platforms. Therefore, `libc` becomes the *de facto* interface between application code and the host platform.

The default Ubuntu packages for Python, Java JRE, and Bash all use `libc`, so any programs written for those interpreters using the usual I/O mechanisms will not bypass library interposition. The GCC, Clang, Rust, and the Glasgow Haskell Compiler packages in Ubuntu also emit code that targets `libc`. The only major exceptions are Zig and Go lang.

One may at least detect that a potential bypass may occur, by analyzing the machine code of executables and libraries and searching for the `syscall` instruction. PROBE is in a primary position to do this, since it already hooks all `exec` calls, however we have not had enough time to implement this yet.

**Fragility**: Library interposition may interfere with other applications that also use library interposition. Each interposer would intercept the call and hands it off to the next library on the path, unaware if this is a true `libc` or an interposing `libc`, like a telescoping tube. It is possible that the behavior modifications introduced by one interposer violates some implicit contract expected by another, but it is equally possible that each passes along the call to the next without negative interference.

O'Callahan's citations refer to ASAN (Address SANitization) [26] and WINE (Wine Is Not an Emulator) [3]. We tested ASAN in PROBE. ASAN has to be informed to ignore that it is not at the *top* of the preload list by setting the environment variable `ASAN_OPTIONS=verify_asan_link_order=0`. After that, no errors occurred.

> An argument in favour of such [interposition] systems is that modifications to existing libraries are more likely to be adopted than modifications to the kernel. However, for this approach to work, all applications in the system need to be built against, or dynamically linked to, provenance-aware libraries, replacing existing libraries.

— Pasquier et al. [23]

**Recompiling binaries**: This is only true for *compile-time* linkage of library interposition. However, modern operating systems (Linux, *BSD, MacOS, Windows) provide for linking libraries at program load-time. PROBE does not require recompilation or re-linking of binaries.

## 8 Future work

- This Docker *image* should not be confused with a `Dockerfile`. A `Dockerfile` is much smaller and more convenient to send, but it does not necessarily build to a bit-wise reproducible Docker image.

- Improve completeness: static binary rewriting

- Improve performance

- Multi-node and HPC cases

## 9 Conclusion

## Acknowledgments

## Availability

## References

[1] Alice Allen, Kimberly DuPrie, Bruce Berriman, Robert J. Hanisch, Jessica Mink, and Peter J. Teuben. 2012. Astrophysics Source Code Library. (Dec. 9, 2012). arXiv: 1212.1916 [astro-ph]. Retrieved Jan. 13, 2025 from http://arxiv.org/abs/1212.1916. Pre-published.

[2] 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. 5, 1990), 403–410. DOI: 10.1016/S0022-2836(05)80360-2.

[3] Bob Amstadt and Michael K. Johnson. Wine Is Not an Emulator. (). Retrieved Jan. 13, 2024 from https://dl.acm.org/doi/fullHtml/10.5555/324681.324684.

[4] 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). Retrieved July 6, 2023 from https://www.usenix.org/conference/tapp13/technical-sessions/presentation/balakrishnan.

[5] 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, (June 26, 2016), 2085–2088. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2899401.

**Applications**

| workload | strace | probe | ptu | care |
|---|---|---|---|---|
| titanic-0 | 1.59 | 1.04 | 1.76 | 1.89 |
| titanic-1 | 1.26 | 1.01 | 1.32 | 1.44 |
| house-prices-0 | 1.21 | 1.01 | 1.24 | 1.31 |
| umap | 1.09 | 1.51 | 1.51 | 1.60 |
| hdbscan | 1.07 | 1.16 | 1.29 | 1.31 |
| blastn | 1.03 | 0.99 | 1.03 | 1.02 |
| blastx | 1.02 | 0.99 | 1.00 | 1.10 |
| blastp | 0.99 | 0.97 | 1.01 | 1.05 |
| **gmean** | 1.14 | 1.07 | 1.25 | 1.31 |

**Synthetic benchmarks**

| workload | strace | probe | ptu | care |
|---|---|---|---|---|
| shell | 0.87 | 0.90 | 0.88 | 0.94 |
| syscall | 1.03 | 0.96 | 1.00 | 0.97 |
| fork | 0.91 | 1.10 | 1.05 | 0.91 |
| fstat | 1.03 | 1.01 | 1.02 | 0.98 |
| exec | 0.90 | 0.92 | 0.81 | 0.93 |
| open/close | 0.98 | 0.98 | 1.00 | 1.11 |
| write | 1.18 | 1.00 | 1.03 | 0.97 |
| stat | 1.08 | 1.25 | 1.02 | 1.16 |
| read | 1.05 | 1.03 | 1.06 | 1.00 |
| postmark | 6.71 | 1.14 | 6.03 | 2.98 |
| **gmean** | 1.21 | 1.02 | 1.18 | 1.11 |

Table 3: Walltime overhead of applications in various provenance tracers as a multiple of native execution. 2.0 means that the program takes twice as long when running in the provenance tracer.

[6] George Coulouris and NIH Staff. 2016. Blast Benchmark. Fiehn Lab. Retrieved Dec. 4, 2023 from https://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/blast-benchmark.

[7] 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, (Apr. 2017), 316–319, 4, (Apr. 2017). DOI: 10.1038/nbt.3820.

[8] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10, 3, (May 2008), 11–21. DOI: 10.1109/MCSE.2008.79.

[9] 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 *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability*. ACM REP '24: ACM Conference on Reproducibility and Replicability. ACM, Rennes France, (June 18, 2024), 85–95. ISBN: 979-8-4007-0530-4. DOI: 10.1145/3641525.3663627.

[10] 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, (June 28, 2023), 74–84. ISBN: 979-8-4007-0176-4. DOI: 10.1145/3589806.3600037.

[11] Philip Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *2011 USENIX Annual Technical Conference*. USENIX Annual Technical Conference. USENIX, Portland, OR, USA, (June 14, 2011). https://www.usenix.org/legacy/events/atc11/tech/final_files/GuoEngler.pdf.

[12] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal*, 26, 6, (Dec. 1, 2017), 881–906. DOI: 10.1007/s00778-017-0486-1.

[13] Alexandre Hocquet and Frédéric Wieber. 2021. Epistemic issues in computational reproducibility: software as the elephant in the room. *European Journal for Philosophy of Science*, 11, 2, (Apr. 17, 2021), 38. DOI: 10.1007/s13194-021-00362-9.

[14] 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, (June 9, 2014), 1–7. ISBN: 978-1-4503-2951-4. DOI: 10.1145/2618137.2618138.

[15] Jeffrey Katcher. 2005. PostMark: A New File System Benchmark. TR3022. (Sept. 1, 2005). Retrieved Jan. 22, 2024 from https://web.archive.org/web/20050901112245/https://www.netapp.com/tech_library/3022.html.

[16] Johannes Köster and Sven Rahmann. 2012. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28, 19, (Oct. 1, 2012), 2520–2522. DOI: 10.1093/bioinformatics/bts480.

[17] 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*. Network and Distributed System Security (NDSS) 2017.

[18] 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. ISBN: 978-1-931971-40-9. Retrieved Aug. 23, 2023 from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ma.

[19] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*. USENIX 1996 ATC (Annual Technical Conference). USENIX, San Diego, CA, (Jan. 1996). https://www.usenix.org/legacy/publications/library/proceedings/sd96/full_papers/mcvoy.pdf.

[20] Marcin Miłkowski, Witold M. Hensel, and Mateusz Hohol. 2018. Replicability or reproducibility? On the replication crisis in computational neuroscience and sharing only relevant detail. *Journal of Computational Neuroscience*, 45, 3, (Dec. 1, 2018), 163–172. DOI: 10.1007/s10827-018-0702-z.

[21] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *2006 USENIX Annual Technical Conference*. 2006 USENIX Annual Technical Conference. https://dash.harvard.edu/handle/1/23853812.

[22] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. (May 16, 2017). arXiv: 1705.05937 [cs]. Retrieved Jan. 26, 2024 from http://arxiv.org/abs/1705.05937. Pre-published.

[23] 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, (Sept. 24, 2017), 405–418. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3129249.

[24] Silvio Peroni and David Shotton. 2020. OpenCitations, an infrastructure organization for open scholarship. *Quantitative Science Studies*, 1, 1, (Feb. 1, 2020), 428–444. DOI: 10.1162/qss_a_00023.

[25] 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). Retrieved Feb. 14, 2024 from https://www.usenix.org/conference/tapp13/technical-sessions/presentation/pham.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A Fast Address Sanity Checker. In 2012 USENIX Annual Technical Conference (USENIX ATC 12), 309–318. Retrieved Jan. 14, 2025 from https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[27] The Galaxy Community et al. 2024. The Galaxy platform for accessible, reproducible, and collaborative data analyses: 2024 update. *Nucleic Acids Research*, 52, W1, (July 5, 2024), W83–W94. DOI: 10.1093/nar/gkae410.

[28] 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)*. 2017 IEEE 13th International Conference on E-Science (e-Science). (Oct. 2017), 374–383. DOI: 10.1109/eScience.2017.51.

[29] 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, (June 15, 1998), 3.

[30] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. Zpoline: a system call hook mechanism based on binary rewriting. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), 293–300. ISBN: 978-1-939133-35-9. Retrieved Jan. 14, 2025 from https://www.usenix.org/conference/atc23/presentation/yasukata.

[31] 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, (Oct. 28, 2007), 116–127. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315261.

## A   Soundness of PROBE

The user supplies a **command**, such as python script.py -n 42, to PROBE.

PROBE runs command with certain environment variables set, resulting in a **process**.

The process may create **child processes** that will also get traced by PROBE.

If a process calls a syscall from the exec-family, a new process is created with the same PID. We call the pair of (PID, "number of times exec has been called"), an **exec epoch**. Each process has at least one exec epoch.

Each process can spawn kernel-level **threads** that provide concurrent control-flow in the same address-space identified by the triple (PID, exec epoch, TID).

Threads do **operations**, like "open file.txt for reading" or "spawn a new process", identified by (PID, exec epoch, TID, operation number), where operation number increments for every operation the thread does.

A **dynamic trace** of a command is an tuple of:

- a PID which is the root

- a mapping of processes to an ordered list of exec epochs

- a mapping of exec epochs to threads

- a mapping of threads to a list of operations

Dynamic traces are what PROBE actually records.

**Program order** is a partial order on operations where *A* precedes *B* in program order if *A* and *B* are in the same thread and *A*'s operation number is less than *B*'s.

**Synchronization order** is a partial order on operations where *A* precedes *B* in program order for specific special cases based on the semantics of the operation. PROBE currently tracks the following cases:

- *A* is an exec and *B* is the first operation of the next exec epoch for that process

- *A* is a process-spawn or thread-spawn and *B* is the first operation in the new process or thread.

- *A* is a process-join or thread-join and *B* is the last operation in the joined process (any thread of that process) or joined thread.

But the model is easily extensible other kinds of synchronization including shared memory locks, semaphores, and file-locks.

**Happens-before order**, denoted $\leq$, is a partial order that is the transitive closure of the union of program order and synchronization order.

We define a **dataflow** as a directed acyclic graph whose nodes are operations or versioned files. The edges are the union of happens-before edges and the following:

- If operation *A* opens a file at a particular version *B* for reading, $A \rightarrow B$.

- If operation *A* closes a file at a particular version *B* which was previously open for writing, $A \rightarrow B$.

Tracking the *versioned files* instead of files guarantees non-circularity.

Rather than track every individual file operation, we will only track file opens and closes. If processes concurrently read and write a file, the result is non-deterministic. Most working programs avoid this kind of race. If a program does have this race, the dataflow graph will still be sound, but it may be *imprecise*, that is, it will not have all of the edges that it could have had if PROBE tracked fine-grain file reads and writes.

A **file** is an inode. Defining a file this way solves the problem of *aliasing* in filesystems. If we defined a file as a path, we would be fooled by symlinks or hardlinks. When we observe file operations, it is little extra work to also observe the inodes corresponding to those file operations.

In practice, we use the pair modification times and file size as the version. Modification time can be manipulated by the user, either setting to the current time with `touch` (common) or resetting to an arbitrary time with `utimes` (uncommon). Setting to the current time creates a new version which does not threaten the soundness of PROBE. Setting to an arbitrary time and choosing a time already observed by PROBE does threaten its soundness. For this reason, we consider the file size as a "backup distinguishing feature". It is unlikely that a non-malicious user would accidentally reset the time to the exact time (nanosecond resolution) we already observed and have the exact same size.

In the event of a data race on a file write, the dataflow graph generated by our approach ensures that all potential dependencies are captured as edges. The child processes of a parent process inherit write access to a file opened by the parent and are treated as dependencies of the incremented final version of the file. This guarantees that no critical dependency is overlooked, ensuring the soundness of the dataflow graph. However, the approach may not achieve completeness, as some processes with write access may not represent true dependencies. Since we have limited information about the order in which the shared file is accessed by the processes and the exact change made to the file, this approach uses the access mode effectively to construct a graph that prioritizes soundness.

## A.1 Event coverage

We read the GNU C Library manual[2] and wrapped every function that does file I/O or changes how paths are resolved (e.g., `chdir`) in the following chapters, with the exception of redundant functions[3]:

- Chapter 12. Input/Output on Streams

- Chapter 13. Low-Level Input/Output

- Chapter 14. File System Interface

- Chapter 15. Pipes and FIFOs

- Chapter 16. Sockets

- Chapter 26. Processes

Our research prototype wraps:

- file `open` and `close` family of functions[4]

- `chdir` family of functions

- directory opens, closes, and iterations families of functions

- file `stat` and `access` families of functions

- file `chown`, `chmod`, and `utime` families of functions

- `exec` family of functions

- `fork`, `clone`, `wait` families of functions

- `pthread_create`, `pthread_join`, `thrd_create`, `thrd_join`, etc. functions

---

[2] https://www.gnu.org/software/libc/manual/html_node/

[3] One function, A, is redundant to another one B, if I/O through A necessitates a call through B. We need only wrap B to discover that I/O will occur. For example, we need only log file openings and closings, not individual file reads and writes.

---

[4] The "family of functions" includes 64-bit variants, `f*` variants (`fopen`), re-open, close-range