Transparent, performant, non-privileged provenance tracing through library interposition

(Anonymous)

Abstract

System-level provenance tracing is the idea of automatically capturing how computational artifacts came to be (e.g., what version of the program wrote this file). While provenance is often discussed in the context of security, it also fills an important niche in computational science, providing data for reproducibility, incremental computation, and debugging. Unlike a security administrator, computational scientists do not necessarily have root-level access to the machine on which they want to trace provenance. Prior work proposes recompiling with instrumentation, ptrace, and kernel-based auditing, which at best achieves two out of three desirable properties: transparency, performance, and non-privilege. We present PROBE, a system-level provenance tracer that uses library interpositioning to achieve all three. We evaluate the performance of PROBE on system microbenchmarks and scientific applications. We also discuss the completeness of the provenance that PROBE collects compared to other provenance tracers.

1 Introduction

For computational artifacts, computational provenance (just **provenance** from here on) 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 has a number of use-cases discussed in prior work:

1. **Reproducibility** [2]. Provenance could aid in *automatic*

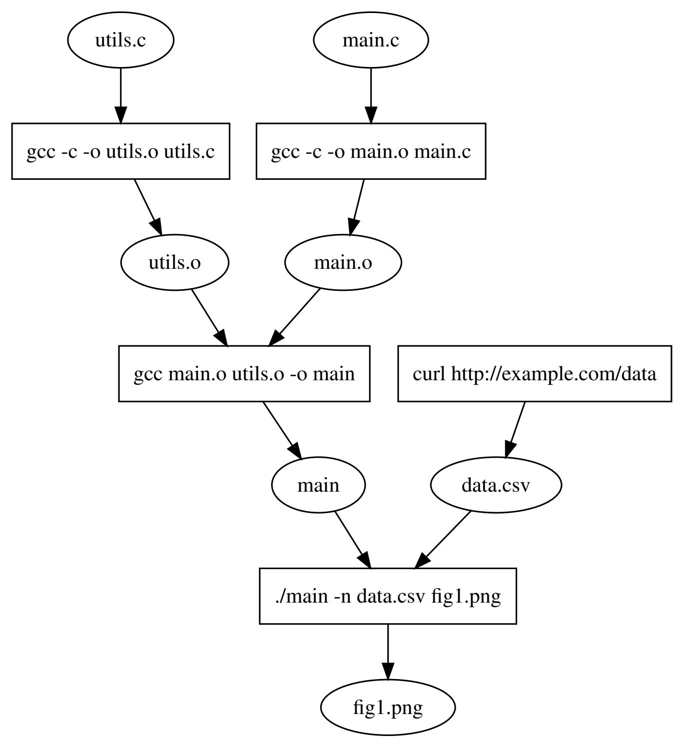

Figure 1: Example provenance graph of `fig1.png`. Artifacts are ovals; processes are rectangles.

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.

2. **Incremental computation** [13]. Iterative development cycles from developing code, executing it, and changing the code. A system like Make reads a provenance graph and determines which processes need to be re-executed.

3. **Comprehension** [8]. Provenance helps the user understand the flow of data in a complex set of processes, perhaps separately invoked. An tool that consumes provenance can answer queries like:

   - "Where did this data file come?" Answering, "this raw data processed by this list of commands".
   - "From what version of the script did this figure derive?" Answering with a copy of the script at the particular version which generated that output. While code is often versioned in VCS, large binary outputs like data and figures are more difficult.
   - "Does this output use FERPA-protected data (i.e., data located /path/to/ferpa)?"

4. **Differential debugging** [8]. 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?

5. **Intrusion-detection and forensics** [8] 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.

These features necessitate the following design features of provenance tracers:

1. **Non-privilege**: A user should be able to use SLRP to trace their own processes without root-level access. While appropriate for security use-cases, computational scientists would likely not have root-level access on shared systems.

2. **Performance**: SLRP should have a minimal performance overhead from native execution. If the performance overhead is noticeable, users may selectively turn it off, resulting in provenance with gaps in the history.

3. **Transparency**: Users should not have to change or recompile their code to track provenance.

Prior work misses at least one of these three:

- eBPF, Linux Audit framework, Linux Provenance Modules/Linux Security Modules [1], and CamFlow [10] use Linux kernel-level functionality violating non-privilege.

- ReproZip [2] and Sciunits [12] use ptrace, which has a significant performance overhead. Record/replay tools such as RR [9] and CDE [4] are similar to provenance tracers in this category. Record/replay tools seek automatic reproducibility but do not satisfy the other features of provenance tracers discussed above.

- PASSv2 [7] requires the user to instrument their code to emit provenance data to a colelctor, violating transparency.

We present a provenance tracer based on library interposition called PROBE, a non-privileged SLRP tracer that maintains performance and transparency. The rest of the work proceeds as follows:

- Sec. 3 summarizes prior SLRP and related prior works
- Sec. 4 documents the high-level design of PROBE.
- Sec. 5 documents low-level implementation details of PROBE.
- Sec. 6 quantifies the completeness of PROBE with respect to various information sources.
- Sec. 7 outlines future work we would like to do on PROBE.
- Sec. 7.1 outlines how we intend to analyze the performance of PROBE and related work.

2 Background

Provenance can be **retrospective**, tracing computational steps that *were run*, or **prospective**, determining what computational steps *should be run* [15]. Some programming systems permit determining prospective provenance, but it is easier to trace retrospective provenance dynamically.

Provenance can be collected at several different levels

1. **Application-level**: modify each application to emit provenace data. Application-level provenance is the most semantically rich but least general, as it only enables collection by that particular modified application [3].

2. **Language/workflow-level provenance**: modify the programming language or workflow language, and all programs written for that language would emit provenance data. Workflow engines are only aware of the dataflow, not higher-level semantics, so workflow-level provenance is not as semantically rich as application-level

provenance. However, workflow-level is more general than application-level provenance, as it enables collection in any workflow written for that modified engine [3].

3. **System-level provenance**: use operating system facilities to report the inputs and outputs that a process makes. 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 [3].

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

SLRP has the following design features:

- **Completeness**: The SLRP should trace as many sources of information from the host as possible, although there are some that may be too impractical to trace.
- **User-level**: SLRP should be able to implemented at a user-level as opposed to kernel-level. Non-privilege implies user-level, but user-level does not imply non-privilege.
- **TODO**

3 Prior work

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

- **Virtual machines**: running the application in a virtual machine that tracks information flow. This method is extremely slow; e.g., PANORAMA has 20x overhead [14].
- **Recompiling with instrumentation**: recompile, where the compiler or libraries insert instructions that log provenance data, e.g., [6]. This method is not transparent.
- **Static/dynamic binary instrumentation**: either before run-time (static) or while a binary is running (dynamic) change the binary to emit provenance data [5]. This method requires special hardware (Intel CPU) and a proprietary tool (Intel PIN).
- **Kernel modification**: modify the kernel directly or load a kernel module that traces provenance information, e.g., [10]. This method is neither non-privileged nor user-level.

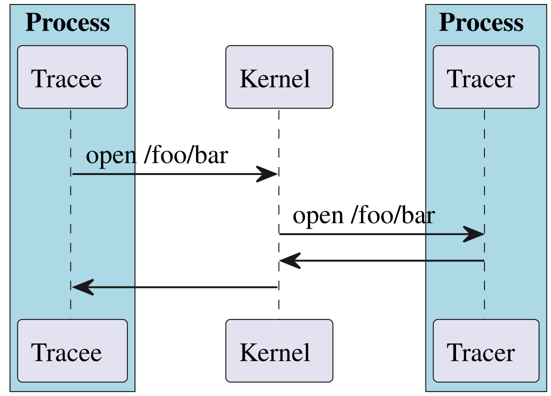

Figure 2: Sequence diagram of process with user-level debug tracing

- **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 non-privileged.
- **User-level debug tracing**: use user-level debug tracing functionality provided by the OS (e.g, Linux/ptrace used by strace, CDE [4], SciUnit [11], Reprozip [2], RR [9]).
- **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 non-privilege, transparency, no special hardware, and performance overhead less than 10-times are hard-requirements, the only possible methods are user-level debug tracing and library interposition.

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.

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 eliminates the extra context switches fig. 3.

TODO: Matrix:

- Each row is a group of prior works
- Each column is an SLRP tracer-feature

Prior works argue that library interposition is not appropriate for SLRP for the following reasons:

- **Bypassable by direct system calls**
- **Fragility due to variations in C libraries**

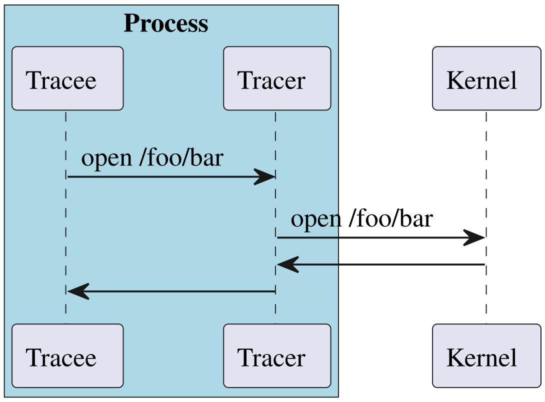

Figure 3: Sequence diagram of process with library interposition

- **Breaks other applications that use preloading**
- **Requires rebuilding or re-linking**
- **TODO**

  A common technique for intercepting system calls inprocess is to use dynamic linking to interpose wrapper functions over the C library functions that make system calls. In practice, we have found that method 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].

  An alternative implementation of whole-system provenance is interposition between system calls and libraries in user space, as in OPUS [9]. An argument in favour of such 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.

4 Concepts

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

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` (very common) or resetting to an arbitrary time with `utimes` (very 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". We consider it very 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.

5 Implementation

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. forward the call to the *true* libc implementation
3. log the underlying libc's returned value
4. return the underlying libc's returned value

There is no data shared between threads as the log is thread-local. The dynamic-trace consists of information that was collected at a thread-local level, but can be aggregated into a global-level dataflow graph as described in Sec. 4.

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. After the process dies, these log files can be stitched together into a single dynamic trace.

PROBE has a command-line interface. The `record` subcommand:

1. sets `LD_PRELOAD` to load `libprobe.so`
2. runs the user's provided command
3. stitches the PROBE data files into a single, readable log for other programs

There are also several subcommands that analyze or export the provenance. Those subcommands generally use the dataflow representation rather than the PROBE dynamic trace.

6 Completeness analysis

We read the GNU C Library manual[1] 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[2]:

- 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[3]
- `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

7 Future work

- Improve completeness: static binary rewriting
- Improve performance
- Multi-node and HPC cases

7.1 Performance analysis

We intend to do a performance analysis by studying commonly used scientific applications. We will sample popular projects from several repositories and run them in PROBE including:

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

[2] 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.

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

- Spack packages, filtering for packages that contain an executable, and using each package's project's GitHub repo's stars as a measure of popularity
- Kaggle notebooks, using the number of stars as popularity
- WorkflowHub, using the number of citations of the associated DOI as a measure of popularity

References

[1] 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. ISBN: 978-1-939133-11-3. Retrieved Aug. 25, 2023 from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/bates.

[2] 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.

[3] 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. interest: 97. DOI: 10.1109/MCSE.2008.79.

[4] 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.

[5] 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.

[6] 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.

[7] 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, (June 14, 2009), 10. DOI: 10.5555/1855807.1855817.

[8] 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. Accepted: 2015-12-07T19:07:43Z. https://dash.harvard.edu/handle/1/23853812.

[9] 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]. DOI: 10.48550/arXiv.1705.05937.

[10] 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.

[11] 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.

[12] 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.

[13] 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. Retrieved Aug. 23, 2023 from.

[14] 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.

[15] Yong Zhao, Michael Wilde, and Ian Foster. 2006. Applying the virtual data provenance model. In *Provenance and Annotation of Data* (Lecture Notes in Computer Science). Luc Moreau and Ian Foster, (Eds.) Springer, Berlin, Heidelberg, 148–161. ISBN: 978-3-540-46303-0. DOI: 10.1007/11890850_16.