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

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

Shofiya Bootwala
Gujarat Technological University
Ahmedabad, Gujarat, India

Jenna Fligor
jenna@fligor.net
University of Illinois
Urbana-Champaign
Dept. of Computer Science
Urbana, IL, USA

Kyrillos Ishak
Alexandria University
Alexandria, State, Egypt

Saleha Muzammil
National University of Computer and
Emerging Sciences
Lahore, Punjab, Pakistan

Asif Zubayer Palak
asif.zubayer.palak@g.bracu.ac.bd
BRAC University
Merul Badda, Dhaka, Bangladesh

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

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

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

ABSTRACT

Provenance tracing is the idea of capturing the *provenance* of computational artifacts, (e.g., what version of the program wrote this file). 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 collector that uses library interpositioning to achieve all three. We evaluate the performance of PROBE for scientific users.

ACM Reference Format:
Samuel Grayson, Shofiya Bootwala, Jenna Fligor, Kyrillos Ishak, Saleha Muzammil, Asif Zubayer Palak, Reed Milewicz, Daniel S. Katz, and Darko Marinov. 2024. Transparent, performant, non-privileged provenance tracing through library interposition. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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).

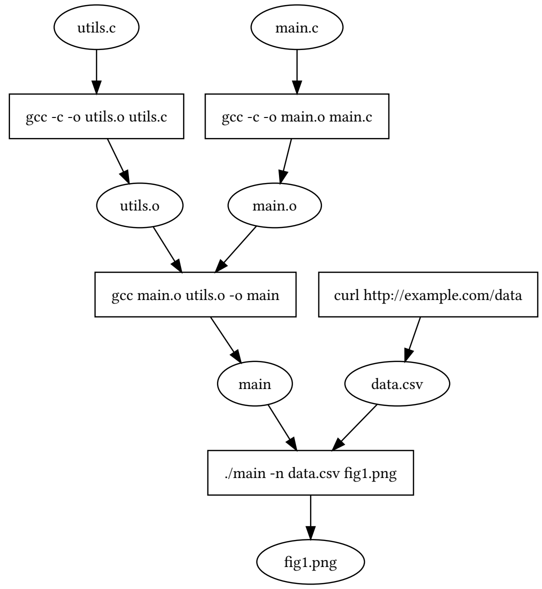
utils.c
main.c
gcc -c -o utils.o utils.c
gcc -c -o main.o main.c
utils.o
main.o
gcc main.o utils.o -o main
curl http://example.com/data
main
data.csv
./main -n data.csv fig1.png
fig1.png

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

Provenance can be **retrospective**, documenting processes that were run, or **prospective**, specifying what processes should be run. 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

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

but least general, as it only enables collection by that particular modified application [2].

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 [2].
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 [2].

This work focuses on system-level, retrospective provenance (SLRP). SLRP tracing has a number of applications:

1. **Reproducibility**. SLRP collects a description of how to generate each artifact, so it could inform users what commands are necessary to reproduce their work.
2. **Caching subsequent re-executions**. Computational science often involves an iterative process of developing code, executing it, changing the code, and repeating. SLRP could inform the user what commands have to be re-run after minor changes to the code. Other solutions like Make require the user to manually specify a dependency graph, which is often unsound (not all dependencies are present).
3. **Comprehension**. Provenance helps the user understand and document workflows and workflow results. An automated tool that consumes provenance can answer queries like "What version of the data did I use for this figure?" and "Does this workflow include FERPA-protected data?".

There are several design features of SLRP tracers:

1. **Transparency**: The user should not have to change their code to make it work in SLRP.
2. **Non-privilege**: A user should be able to use SLRP to trace their own processes without root-level access. Users of shared systems (e.g., HPC) would likely not have root access. Non-privilege may be relaxed to **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.
3. **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.
4. **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.

We present PROBE, an SLRP that is a non-privileged SLRP tracer that is reasonably complete and performant. We will explain how it works and document the limitations to completeness. A complete performance analysis of PROBE and related prior work is left for future work.

The rest of the work proceeds as follows:

- Sec. 1.1 summarizes prior SLRP and related prior works
- Sec. 2 documents the high-level design of PROBE.
- Sec. 3 documents low-level implementation details of PROBE.
- Sec. 4 quantifies the completeness of PROBE with respect to various information sources.
- Sec. 5 outlines future work we would like to do on PROBE.
- Sec. 5.1 outlines how we intend to analyze the performance of PROBE and related work.

1.1 Prior work

There have been several methods of collecting 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 [9].
- **Recompiling with instrumentation**: recompile, where the compiler or libraries insert instructions that log provenance data, e.g., [5]. This method is not transparent.
- **Static/dynamic binary instrumentation**: either before runtime (static) or while a binary is running (dynamic) change the binary to emit provenance data [4]. 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., [7]. This method is neither non-privileged nor user-level.
- **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 [3], SciUnit [8], Reprozip [1], RR [6]).
- **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, 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.

2 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.

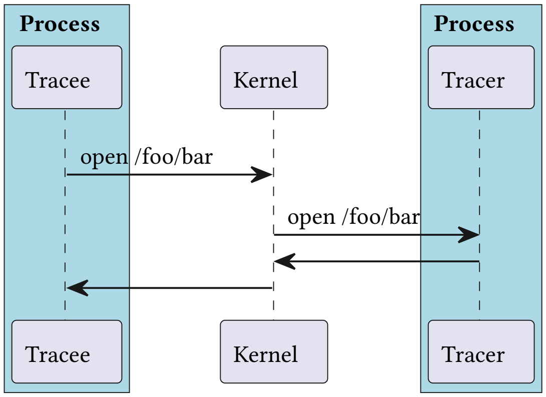

Figure 2: Sequence diagram of process with user-level debug tracing

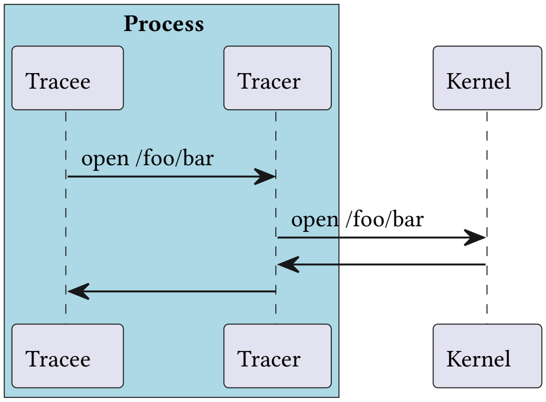

Figure 3: Sequence diagram of process with library interposition

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.

3 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. 2.

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.

4 COMPLETENESS ANALYSIS

We read the GNU C Library manual[1] and wrapped every function that appeared to do file I/O in the following chapters, with the exception of redundant functions:

- 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

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.

So far, we wrapped:

- file `open` and `close` familiy of functions (64-bit variants, `f*` variants, re-open, close-range, etc.)
- changing directories (so we can determine how paths are resolved)
- directory opens, closes, and iterations families
- 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

5 FUTURE WORK

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

5.1 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:

- 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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