Practical Whole-System Provenance Capture

Thomas Pasquier[1], Xueyuan Han[1], Mark Goldstein[1], Thomas Moyer[2], David Eyers[3], Margo Seltzer[1] and Jean Bacon[4]

[1]Harvard University, [2]University of North Carolina at Charlotte, [3]University of Otago, [4]University of Cambridge

Abstract

Data provenance is a record of how data came to be in its present form, providing a *lineage* or *pedigree* for the data. Provenance includes the source of the data and all transformations that have been applied to it. There has been recent interest in whole-system provenance, to facilitate the comprehensive, systematic, and ubiquitous record of a system's behaviour. To date, none of these provenance systems have enjoyed widespread adoption, because they: A) impose too much overhead, B) are designed for long-outdated operating system kernel releases and are hard to port to more modern systems, and C) generate too much data. We address these shortcomings with CamFlow: A) by leveraging the latest kernel design advances to achieve efficiency; B) using a self-contained, easily maintainable implementation relying on a Linux Security Module, NetFilter, and other existing kernel facilities; and C) by providing a mechanism to tailor the provenance data that will be captured to the needs of the application.

Provenance Graph

A provenance graph represents *entities*, *activities* and *agents*. At the OS level, *entities* are typically kernel data objects: files, messages, packets, etc., but also xattributes, inode attributes, exec call parameters, network addresses, etc. *Activities* are processes carrying out manipulations on entities. *Agents* are persons or organisations (e.g., users, groups, etc.) on whose behalf the activities operate. Fig. 3 illustrates these concepts.

Processes exchange information via system calls. Some system calls represent information exchange at some discrete point in time e.g., read, write; others may create shared states, e.g., mmap. Previous implementations managed shared states by recording the action of "associating" the shared memory with a process (e.g., shmget, mmap_file). At query time, these shared states are either ignored or responsible for significant complexity.

CamFlow takes a different approach and conservatively assumes that any incoming or outgoing flow of information to/from a process implies an incoming or outgoing flow to/from a shared state (represented by shared_read, shared_write relationships).

System Overview

Our implementation builds upon and learns from previous OS provenance capture mechanisms, namely PASS, Hi-Fi, and LPM. The provenance data is captured through `Linux Security Module` hooks and `NetFilter` hooks. The provenance data is transferred to the user space through `relayfs`, where it can be stored or analysed. Applications can enrich system level provenance with application-specific details through a pseudofile interface. The provenance capture can be tailored to suit the needs of the application. This is done through pseudofiles and restricted to the owners of the capability `CAP_AUDIT_CONTROL`. We provide a library that, through an API, abstracts interactions with the pseudo-files and `relayfs`. This is shown in Fig. 1.

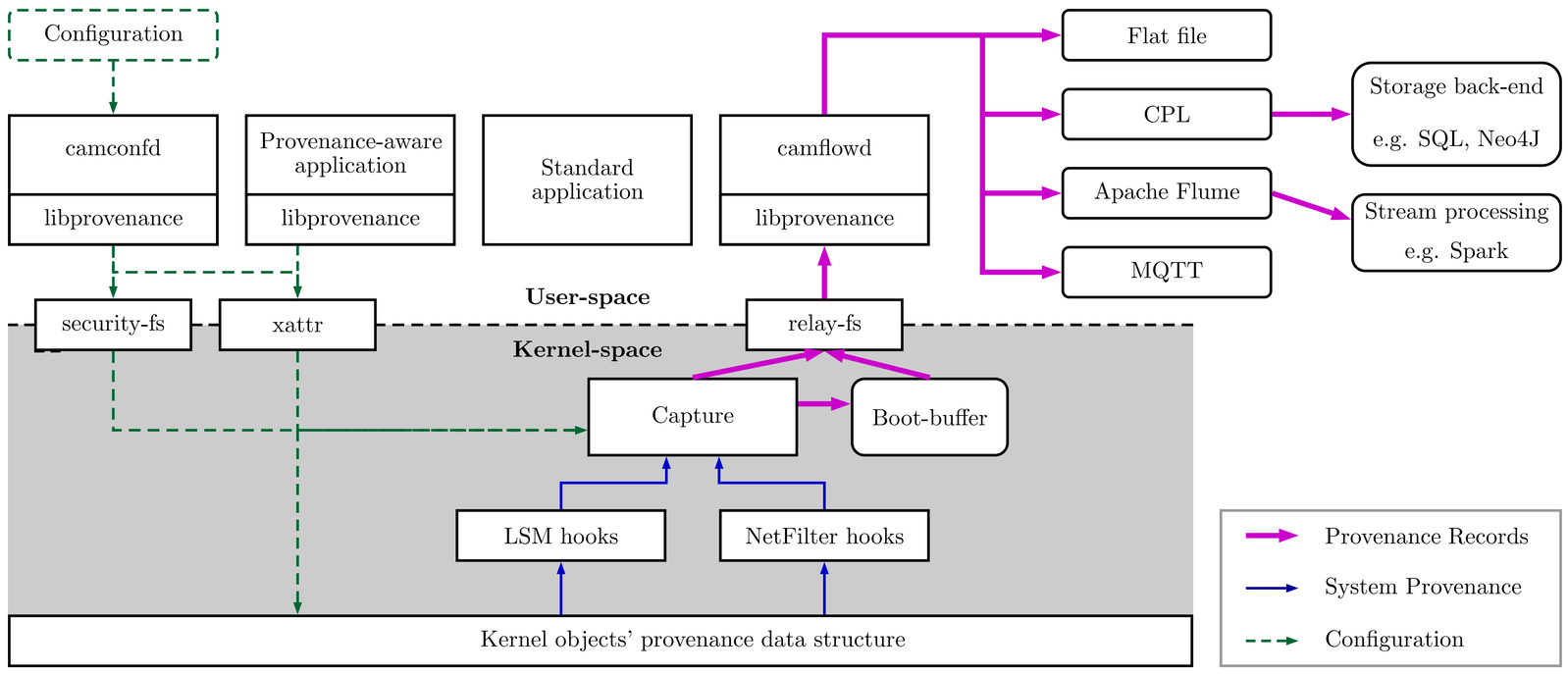

Figure 1: CamFlow architecture.

Provenance system	Headers	Codes	Total	LoC
(kernel version)	Number of files			
PASS (v2.6.27.46)	18	69	87	5100
LPM (v2.6.32 RHEL 6.4.357)	13	61	74	2294
CamFlow (v4.9.5)	8	1	9	2428

Table 1: Extent of modifications, number of files modified, and lines of codes (LoC). The number of files modified measurement is based on the last source code available from the PASS and Hi-Fi/LPM teams, and for CamFlow v0.2.1.

Test Type	vanilla	whole	overhead	selective	overhead
Execution time in seconds, smaller is better					
unpack	11.16	11.36	2%	11.24	<1%
build	433	442	2%	429	0%
4kB to 1MB file, 10 subdirectories, 4k5 simultaneous transactions, 1M5 transactions					
postmark	71	79	11%	75s	6%

Table 2: Macrobenchmark results.

Selective Capture

A major concern about whole-system provenance capture is that the amount of data generated can be extremely large. The performance of algorithms that consume provenance is dependent on the graph size. Most applications generally require a well-defined subset of the provenance graph; we provide mechanisms that allow capturing only this particular subset. It is possible to limit capture to specific edge/node types and/or flows from specific sources, such as:

- inodes,
- network interfaces,
- security contexts, and
- user IDs.

Disclosing Provenance

Provenance observed at the system layer may not be sufficient for all use cases. It may be necessary to disclose to the provenance infrastructure the inner workings of a process. PASS was the first system to introduce the possibility for an application to disclose provenance to complement provenance observed at the OS level, and we implement a similar mechanism. However, this process is complex and generally requires engineering effort.

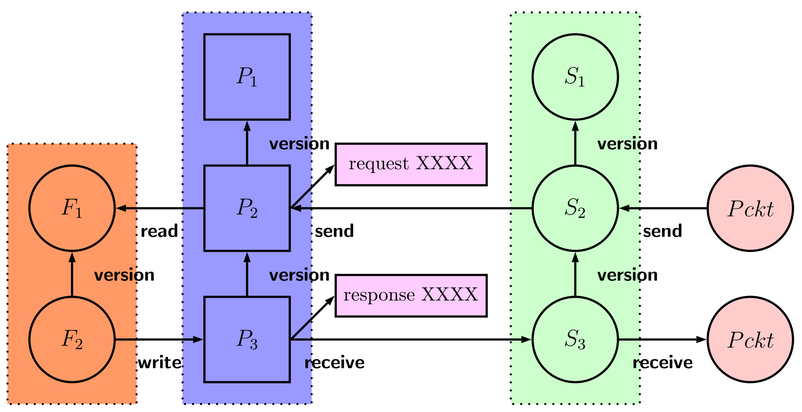

Figure 2: Annotated provenance graph.

To accommodate legacy applications we provide an additional mechanism to annotate the system provenance graph with log information (as illustrated in Fig. 2). Past research has demonstrated that internal provenance of, for example, web-servers can be derived from their logs. Therefore, we can understand an application's inner workings without complex modifications.

Test Type	vanilla	whole	overhead	selective	overhead
Process tests, times in μs, smaller is better					
NULL call	0.07	0.06	0%	0.05	0%
NULL I/O	0.09	0.14	56%	0.14	56%
stat	0.39	0.78	100%	0.50	28%
open/close file	0.88	1.59	80%	1.04	18%
signal install	0.10	0.10	0%	0.10	0%
signal handle	0.66	0.67	2%	0.66	0%
fork process	110	116	6%	112	2%
exec process	287	296	3%	289	<1%
shell process	730	771	6%	740	1%
File and memory latencies in μs, smaller is better					
file create (0k)	5.05	5.32	5%	5.12	1%
file delete (0k)	3.42	3.78	11%	3.79	11%
file create (10k)	9.81	11.41	16%	10.9	11%
file delete (10k)	5.70	6.12	7%	6.08	6%

Table 3: LMbench measurements.

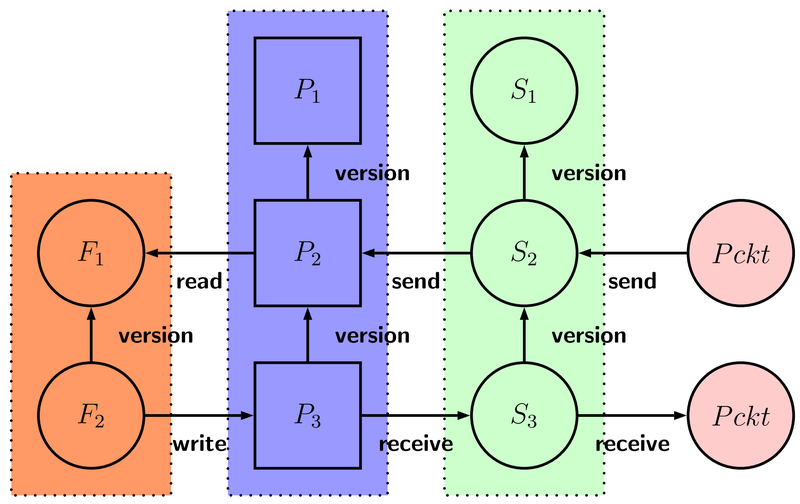

Figure 3: An example of a provenance graph.

Example Applications

Intrusion Detection: the provenance generated by CamFlow is used to detect abnormal system behaviour. Machine Learning algorithms are used over a stream of provenance data (see FRAPpuccino HotCloud'17 paper).

Regulation Compliance: the provenance generated by CamFlow can be used to continuously demonstrate compliance with regulations. The record of system execution is represented as a provenance graph. A number of relevant regulations can be expressed as information flow constraints that translate into properties of paths in the graph.

Download at:
`http://camflow.org`

Acknowledgements

This work was supported by the US National Science Foundation under grant SSI-1450277 *End-to-End Provenance*, and by the UK Engineering and Physical Sciences Research Council under grant EP/K011510 *CloudSafetyNet*.

Contact Information

Email: tfjmp@seas.harvard.edu

UNIVERSITY OF CAMBRIDGE

HARVARD School of Engineering and Applied Sciences

UNIVERSITY of OTAGO Te Whare Wānanga o Otāgo NEW ZEALAND

UNC CHARLOTTE