Expressiveness Benchmarking for System-Level Provenance

Sheung Chi Chan
University of Edinburgh
s1536869@inf.ed.ac.uk

Ashish Gehani
SRI International
ashish.gehani@sri.com

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Ripduman Sohan
University of Cambridge
ripduman.sohan@cl.cam.ac.uk

Hassaan Irshad
SRI International
hassaan.irshad@sri.com

Abstract

Provenance is increasingly being used as a foundation for security analysis and forensics. System-level provenance can help us trace activities at the level of libraries or system calls, which offers great potential for detecting subtle malicious activities that can otherwise go undetected. However, analysing the raw provenance trace is challenging, due to scale and to differences in data representation among system-level provenance recorders: for example, common queries to identify malicious patterns need to be formulated in different ways on different systems. As a first step toward understanding the similarities and differences among approaches, this paper proposes an *expressiveness benchmark* consisting of tests intended to capture the provenance of individual system calls. We present work in progress on the benchmark examples for Linux and discuss how they are handled by two different provenance collection tools, SPADE and OPUS.

Keywords completeness, correctness, versioning, benchmarking

1. Introduction

Provenance is increasingly being used as a basis for security, through forensic audit or online dynamic detection of malicious behaviour. There are several different systems in the literature, such as PASS (Muniswamy-Reddy et al. 2006), Hi-Fi (Pohly et al. 2012), SPADE (Gehani and Tariq 2012) (we refer to SPADEv2 in this paper), OPUS (Balakrishnan et al. 2013) and LPM (Bates et al. 2015) covering a variety of operating systems from Linux and BSD to Android and Windows. They have similar models for provenance recording, in which raw operating system events are processed into high-level provenance graphs that are believed to be suitable for forensic or online analysis. Some systems offer different "filters" or configuration options that govern what information is recorded, motivated by the need to trade off completeness against verbosity and cost of recording detailed information.

Analysts who wish to make use of such systems to detect malicious behaviour face a daunting task. One complicating factor is heterogeneity among different systems. At a syntactic level, standards such as W3C PROV can help different systems interoperate by establishing a common vocabulary for provenance-related data. However, such standards do not prescribe how information specific to the operating system security domain should be represented, or how the provenance record should relate to the actual behaviour of the monitored system. Moreover, formalizing the relationship between a modern operating system kernel and its audit logs or provenance records is non-trivial because the formal model is written imprecisely. (One exception is seL4 (Klein et al. 2009), but its development was a major undertaking that has not been reproduced for operating systems such as Linux.)

In the absence of a tractable approach to formalizing correct and complete behaviour for provenance recording systems, how are we to proceed? We propose a pragmatic approach inspired by previous community efforts such as the Provenance Challenge exercises (Moreau et al. 2008). This paper proposes an "expressiveness benchmark" for OS-level provenance recording systems. The goal of such a benchmark is to provide qualitative answers to the following questions about provenance recording systems:

- What does each node/edge in the graph tell us about actual events reported by sources? (correctness)
- What information is guaranteed to be captured and what are the blind spots? (completeness)
- How is state and change represented in the data? Are objects versioned precisely, imprecisely, or not at all?
- How do different systems, or differently configured systems, compare in these respects?

The goal of such a benchmark is not to produce an absolute ranking of systems, but rather to help place different systems in a landscape so that their relative capabilities can be better understood. We hope that the process of benchmarking and comparing provenance recording systems on comparable examples will help lead to greater uniformity and semantic interoperability among such systems, or to greater understanding of design decisions.

The structure of the rest of this paper is as follows. Section 2 provides some background on the two provenance collection tools considered, SPADE and OPUS. Section 3 presents the details of our proposed expressiveness microbenchmark. Section 4 presents the results of our microbenchmark using SPADE (Gehani and Tariq 2012) with Linux Audit. Section 5 presents the results of our microbenchmark using OPUS (Balakrishnan et al. 2013) under Linux. Section 6 compares the different approaches of the two systems. Finally, Section 7 concludes and discusses ongoing and future steps.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grants IIS-1116414 and ACI-1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. The authors also gratefully acknowledge travel support from GCHQ for Chan to visit SRI and Cambridge.

TaPP 2017, June 22-23, 2017, Seattle, Washington.

2. Background

SPADE and OPUS are two provenance collection tools that were originally developed with different scenarios in mind. In this section, we will discuss the different provenance collection approaches of the two targeted tools, running in a Linux environment.

First of all, we briefly review their architectures and other characteristics. SPADE is aimed at unified provenance collection in a distributed setting, and it focuses on relationships among processes and artifacts across distributed hosts. OPUS is more concerned with complete representation of operations on different files and artifacts and aims to abstract those operations from the operating system and make the provenance collection process portable. It also aims to explain how a given file came to be in its current state, and who has changed or handled it. The developers of OPUS provide a Provenance Versioning Model (PVM) as a formal model for recording both the history of artifacts and of associated meta-data. This can help to identify and preserve the version history of the artifacts.

SPADE works by observing audit events provided by different operating systems. SPADE provides multiple ways to record provenance by breaking its functionality into modules. Anyone can write custom modules for their platform to suit their own needs. Those modules may include filters, transformers, provenance collectors or storage handlers. This flexibility makes it possible to run SPADE across a distributed system. In the Linux environment, the major provenance collector supported by SPADE is the Audit Reporter, which depends on the Audit Daemon of the native Linux environment. When SPADE starts, it supplies audit rules (which can be defined by users) to the audit daemon. The Audit Dispatcher notifies SPADE (and any other monitoring processes) of any activities that fulfil the audit rules, and these are also recorded in the audit log. The audit information is recorded by the kernel, and for our purposes its completeness and correctness are assumed. SPADE also has the ability to identify different subactivities of a process ("units"), but we do not consider this capability in this paper.

OPUS works in a more intrusive way. It redirects standard library calls to pass through an additional layer for provenance collection. Most C library calls perform a combination of kernel system calls. The developers of OPUS have mapped most of the C library calls to syscall combinations, based on the PVM model. When a C library call passes through the additional provenance collection layer, OPUS can record the corresponding system calls in the resulting provenance graph without the need to monitor the underlying syscall operations. OPUS makes use of the ability offered in some operating systems (such as Linux) to alter the dynamic library linking process (i.e. `LD_PRELOAD`), which overrides the Global Offset Table (GOT) in the binary memory and points it to wrapped versions of library calls. Each wrapped call keeps the original call untouched and unaltered. It only provides an additional layer to record informations of the original call before passing it to the lower level. The recorded information is used to generate appropriate provenance in accordance with the PVM; it runs in the background so high-priority user tasks are not significantly slowed. Statically compiled binaries could be patched by adding in extra calls to OPUS's provenance collector, but this approach is not implemented yet. OPUS's approach does not require administrator privileges to run, and may be more portable because it only monitors high level library calls that are platform independent.

3. An Expressiveness Microbenchmark

The goal of expressiveness microbenchmarking is to identify the provenance patterns for the smallest meaningful units: syscalls operating on the OS kernel. Those syscalls combine together in different permutations to form high level processes. In order to understand what syscalls have been involved in the processes and their provenance patterns, we need to understand how each syscall contributes to the resulting provenance graph. That is, we need to understand the explicit patterns of provenance for each syscall. This mapping provides a microbenchmark for each syscall and its effect on the resulting provenance graph. With this understanding, we can use the patterns to generate rules to identify the existence of high level actions (containing different permutation of syscalls) in a large system by analysing the provenance graph describing its runtime behaviour. This can help to trace the existence and initiator of certain actions for security and forensic analysis.

In order to create a microbenchmark for syscalls in the kernel, we need to create some sample programs that use minimal syscalls in order to see those patterns without a large amount of noise. Most syscalls in Linux environment have a one to one mapping to C library calls, so those microbenchmark programs are written in C. We do not test all 300+ syscalls in the Linux operating system, only those syscalls generally believed to relate to sensitive actions have been used as a starting point. These syscalls have been summarized in Table 4 in the appendix. There is also a baseline program (Listing 1 in appendix), which does nothing, to form the control result of the benchmark process. It is used to demonstrate the basic provenance trace generated by starting a C program that does nothing. This can help to isolate and filter out the standalone pattern of each syscall.

There are some benchmark programs attached in the appendix. Each syscall is inserted into the control program separately for generating the unique patterns of provenance. Each time only one syscall (or two if we need to test close syscall) will be inserted into the control program and executed under the provenance collection tools to generate provenance patterns for that syscall. Two such example programs, Listing 2 and 3, together with a combined testing program for multiple syscalls, Listing 4, are also shown in the appendix. It contains multiple chosen syscalls to demonstrate provenance pattern combination and to act as the testing target for queries matching each syscall in a large application. The resulting graph of this program is shown in the appendix (Figure 1).

In this paper, we concentrate on two of the provenance collection tools, SPADE and OPUS. Both of them have the ability to output the provenance result in Neo4J graph database format that can be queried using the Cypher query language. The unique provenance patterns should identify each of the syscall actions. Those results should be enough to generate rules to determine the existence of certain syscall (or combination of syscalls) in a large system. The resulting provenance patterns are expected to behave the same when the syscalls happen in different environments.

4. Benchmarking SPADE in Linux

We executed each testing program using SPADE to record the resulting provenance graph and store it in a Neo4J database. The programs were executed using both static and dynamic loading; we have focused on the statically linked case because the resulting graphs involve less noise resulting from dynamic libraries loading. We compared the resulting graphs with the baseline and with each other and designed queries to detect the distinctive patterns associated with each syscall. In the rest of this section, we summarize the resulting observations about SPADE.

SPADE retrieves information about syscall events directly from the audit dispatcher, and it preserves the timestamps/event id ordering in Linux audit. Sometimes the Linux audit daemon uses multiple log records to keep track of a single syscall event because it may relate to more than one communication party. These log records are grouped together by event id and ordered by timestamps. Multiple audit log records with the same event id are treated as one audit event by SPADE. Multiple events can happen in a very short time (e.g. within 1 micro-second) so they might have the same timestamps. In this case, the event id provides additional information

about the order of events. The timestamps record the *starting time* of a syscall event, while event ids are assigned to syscall events in the order of their *completion*. This information helps discover patterns that are sensitive to event ordering.

In SPADE, artifacts can either be invariant – that is, the same instance represents an object as it is being modified – or versioned, in which a new epoch is created each time a related system call operates on the object. There are settings to track versions of files, sockets, pipes and other artifacts. Those syscalls that may have effects on versioning in SPADE provenance results are shown as **bold** in Table 1 and 2. There will be an extra operation 'update' displayed in the resulting provenance information between the new version and old version of the artifacts in the resulting provenance graph, this provides extra patterns for those syscalls.

SPADE does not record explicit information about some actions. Two representative examples are the `dup` and `mknod` syscalls, which do not touch the content of the artifacts. Instead they just create or clone the internal path mapping elements. In the viewpoint of SPADE, these actions do not affect the content of the artifacts and so they are not recorded explicitly. Nevertheless, they do indirectly affect the behaviour of subsequent syscalls. For example, `dup` creates a new file descriptor of an artifact. This action may have indirect effects on how subsequent syscalls are recorded since processes can communicate with the same artifact with different file descriptor. Another example is the `kill` command. It has no immediately observable effect shown in the provenance graph, but there should not be later events involving the killed process after the `kill` syscall. (SPADE does track `kill` if units are enabled, but we have not considered this configuration yet.) These examples show that provenance may not completely reflect all execution at runtime. Microbenchmarks can help to identify the mappings and understand how different tools handle each syscall execution.

mprotect	mmap_read	mmap_write	send	sendto
sendfile	sendmsg	tee	read	readv
pread	preadv	**recv**	**recvfrom**	**recvmsg**
write	**writev**	**pwrite**	**pwritev**	

Table 1: I/O related syscalls in SPADE

SPADE allows the use of filters (Gehani et al. 2011) to reduce the amount of graph information recorded. In a traditional operating system, I/O related events will contribute a large amount of noise due to background or graphical processes. To make it easier to identify patterns for I/O related syscalls from the provenance graph for microbenchmarking, we have used filters for eliminating part of the noise in the provenance collection process. Table 1 summarizes a list of I/O related syscall (which belong to category 2 in the classification) for which we generate benchmarking provenance patterns with the help of SPADE filtering. We consider file I/O as well as socket and memory related I/O.

Cat 1	Cat 2	Cat 2a	Cat 3	Cat 5
dup(2/3)	create	**fchmod**	execve	pipe
kill	close	**truncate**	(v)fork	pipe2
mknod(at)	open	**ftruncate**	clone	bind
setgid	openat	Cat 4	setuid	connect
setre(s)gid	unlink(at)	link(at)	setreuid	listen
chown	chmod	symlink(at)	setresuid	accept
fchown(at)	fchmodat	rename(at)	exit	accept4

Table 2: Syscalls classification for SPADE

We have classified the syscalls handled by SPADE based on the patterns they produce in the resulting graph. The classification is shown in Table 2. Category 1 includes all those syscalls with no observable patterns from the current implementation of SPADE when they are tested alone. Category 2 includes syscalls involving one process and one artifact. Category 2a contains special cases of category 2 syscalls that involve versioning information in the SPADE result. Category 3 includes syscalls that involve inter-process communications. Category 4 includes syscalls involving one process and two artifacts. Category 5 includes syscalls with irregular provenance patterns that do not fit any of the other categories.

We have written graph database query templates (using Cypher query language) for category 2, category 3 and category 4, and they are shown in the appendix (Listing 5, 6 and 7). There is also a special query (Listing 8 in appendix) for querying syscalls in category 2a. These patterns illustrate how SPADE generates provenance graphs for each of the syscall events captured by the Linux audit daemon. It can be used as a cross reference and testing guideline to validate the processing behaviour of SPADE.

These results highlight some interesting differences in how syscalls are handled in the current implementation. For example, `setuid` syscall group and `setgid` syscall group are similar; the only difference between them is that `setuid` calls target users and the `setgid` calls target the user groups. Intuitively, they should be treated similarly, but SPADE only records explicit provenance for `setuid` calls in our tested version of implementation. (The current version of SPADE records `setgid` calls.)

5. Benchmarking OPUS in Linux

We ran each benchmark program using OPUS for provenance collection. As mentioned earlier, OPUS currently cannot operate on statically linked programs because OPUS works by wrapping dynamically linked libraries. OPUS can only records the path of the binary and other environmental variables when handling static binaries. We therefore ran the benchmarks using dynamically linked binaries, but this yields larger, noisier provenance graphs. We have analysed the graphs to discern common patterns from the result set.

OPUS aims to provide completeness in two dimensions of provenance collection: runtime context and versioning. In order to provide more understanding about the state changes of objects in the operating system, OPUS maintains a framework for recording object versioning changes. When an object has changed and a new epoch of that object is created, the provenance information of that object splits into two series. The Provenance Versioning Model (PVM) acts as the backbone for the OPUS tools when collecting provenance in runtime. This makes OPUS more capable to collect provenance for version changes of an object, providing detailed information about the history of an object. OPUS groups the provenance around an artifact and its related epoch.

Consider a situation in which multiple processes access the same artifact concurrently. If a new syscall action is executed on this artifact, it may affect the artifact's status from the perspective of other processes working on it. If so, then a new version of the artifact should be created for the purpose of subsequent actions. Also, the PVM makes it possible to determine ordering relationships among actions because syscalls executing on older version of an artifact always execute earlier than syscalls executing on a newer version. On the other hand, OPUS does not follow a heavyweight version-on-write model; new versions are only created when necessary to reflect changes visible to other processes.

The classification of OPUS (shown in Table 3) is similar to SPADE, except the additional elements describing those versioning information. The classification is done by treating multiple version of the sample artifacts as one. Category 1 includes all syscalls with no observable patterns using OPUS. Category 2 includes syscalls that operate on artifacts directly. Category 2a includes category 2 syscalls which affects meta data and content of

artifacts, which produces standard versioning information. Category 3 includes syscalls that act on processes only. Category 4 includes syscalls affecting multiple artifacts and one process. Category 5 includes all other syscalls that affect the artifacts themselves but do not fit the patterns of the other categories. These categories match the OPUS team's own classification (OPUS Project).

Cat 1		Cat 2		Cat 2a	Cat 3
dup	pipe(2)	tee	pread(v)	chmod	setuid
dup2	listen	send	pwrite(v)	chown	setre(s)uid
dup3	connect	recv	sendto	(f)truncate	setgid
fork	accept	read	sendmsg	create	setre(s)gid
vfork	accept4	write	sendfile	close	Cat 5
clone	execve	readv	recvmsg	open(at)	rename(at)
kill	mprotect	writev	recvfrom	Cat 4	mknod(at)
exit	mmap_read	fchmod	fchmodat	symlink	link(at)
bind	mmap_write	fchown	fchownat	symlinkat	unlink(at)

Table 3: Syscalls classification for OPUS

Similar to SPADE result, we are only concerned with how the category 1 syscalls affect the later events so we do not provide a query for this category alone. For category 2, 2a, 3 and 4 a simple matching query can discover the associated processes and artifact or artifacts (taking versioning into account). Category 5 is the most difficult category to query because each syscall has a different pattern. Rules for identifying patterns for all categories are still an on-going task for the benchmarking process. Versioning behaviour for syscalls in category 2a, 3, 4 and 5 are all preserved by OPUS, while category 1 and 2 are not related to versioning.

6. Comparison between SPADE and OPUS

The provenance results of SPADE and OPUS are different because they aim to explain different perspectives of runtime behaviour. OPUS concentrates on changes in the file system. It maps multiple library calls into lower-level operations as specified by the PVM. This helps to minimise information since actions with no observable effect on the resulting provenance graph will be ignored. On the contrary, SPADE concentrates on communication between processes and artifacts across distributed hosts, and by default captures these operations without version updates. This setting illustrates the different goals of the two provenance collection tools, and the resulting provenance graphs show different information. SPADE's provenance graph shows processes' communications and actions on artifacts, while OPUS's provenance graph shows more details on which process has contributed to which version of each artifact and can answer more detailed questions on the ordering of actions or which process contributed to each change. On the contrary, OPUS lacks distributed environment handling and initiator recording so cannot answer questions regarding accountability of actions if those requests are from another trust domain.

It is also important to study the filtering functionality of the tools. The inclusion of all events in the audit log in the resulting provenance graph will make it large and hard to analyse. In order to avoid this problem, SPADE has provided two approaches to filter results and narrow down the resulting provenance graph to a suitable size for analysis. These approaches are named filters (Gehani et al. 2011) and transformers (Gehani et al. 2016). They share some characteristics, but operate in different stages within SPADE. Filters work at collection time which allows SPADE to ignore part of the audit log while generating the provenance graph. Transformers work at the querying stage after the provenance graph has been generated, and only the part concerned will be returned when querying for the result. On the contrary, OPUS handles noise in a different way. OPUS tries to abstract application behaviour from the underlying operating system and provides a set of transformations that define every operation. The definition aims to identify the key parts of each operation which contribute to its versioning behaviour. Noise with no effect on the artifact's versioning is ignored. However, the versioning from PVM may be inflexible and may lead to false alarms and result in missing information in the resulting provenance graph. We can see that the different approaches used by SPADE and OPUS aim to filter out different information and thus give different perspectives on runtime behaviour.

7. Conclusions and Future Work

In this paper we present an expressiveness microbenchmark approach for provenance-tracking systems in the Linux environment. The captured provenance patterns of syscalls provide a foundation for generating formal rules to identify complicated process patterns consisting of multiple syscalls in any permutations. By formulating such rules using queries, we can automatically determine whether such patterns exist in the runtime provenance of a large application. This constitutes a step toward formalization of mappings in different settings, such as the mapping from Linux Audit to SPADE provenance results or from C library calls to PVM operations in OPUS. Such formalisms could help to validate the provenance collection stages in these systems. In this paper, we targeted two provenance-aware systems (both running in user space) in order to demonstrate the feasibility of our approach. It could also be useful for understanding and comparing other systems that use kernel-based provenance recording such as Hi-Fi, LPM, or PASS. Lessons learned from benchmarking on OPUS and SPADE could be applied to improve such systems and make their behaviour more uniform, or to develop provenance models for new platforms; extending our approach to handle other tools may help improve the completeness of these systems or reveal further opportunities for aligning or "normalizing" provenance across systems.

References

N. Balakrishnan, T. Bytheway, R. Sohan, and A. Hopper. OPUS: A lightweight system for observational provenance in user space. In *TaPP 2013*, 2013.

A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security 2015*, pages 319–334, 2015.

A. Gehani and D. Tariq. SPADE: support for provenance auditing in distributed environments. In *Middleware 2012*, pages 101–120, 2012.

A. Gehani, D. Tariq, B. Baig, and T. Malik. Policy-based integration of provenance metadata. In *POLICY 2011*, pages 149–152, 2011.

A. Gehani, H. Kazmi, and H. Irshad. Scaling spade to "big provenance". In *TaPP 2016*, Washington, D.C., 2016. USENIX Association.

G. Klein et al. seL4: Formal verification of an OS kernel. In *SOSP 2009*, pages 207–220, New York, NY, USA, 2009. ACM.

L. Moreau et al. Special issue: The first provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):409–418, 2008.

K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56, 2006.

OPUS Project. POSIX mapping in PVM. `http://www.cl.cam.ac.uk/-research/dtg/fresco/opus/posix-pvm.pdf`.

D. J. Pohly, S. E. McLaughlin, P. D. McDaniel, and K. R. B. Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC 2012*, pages 259–268, 2012.

A. Syscall Table

dup	dup2	dup3	mknod	mknodat
kill	fork	vfork	clone	exit
open	openat	pipe	pipe2	execve
create	close	chmod	fchmod	fchmodat
link	symlink	linkat	symlinkat	setuid
setreuid	setresuid	setgid	setregid	setresgid
rename	renameat	truncate	ftruncate	unlink
unlinkat	send	sendto	sendmsg	sendfile
recv	recvfrom	recvmsg	tee	mprotect
mmap_read	mmap_write	chown	fchown	fchownat
bind	connect	listen	accept	accept4
read	readv	pread	preadv	
write	writev	pwrite	pwritev	

Table 4: Syscalls considered in this paper

B. The microbenchmark programs

Listing 1: Control Program

```
void main() {
        //syscall goes here
}
```

Listing 2: Testing Program (Open / Close)

```
#include <stdio.h>
#include <fcntl.h>

void main() {
  close(open("test.txt", O_RDWR));
}
```

Listing 3: Testing Program (Rename)

```
#include <stdio.h>

void main() {
  rename("test.txt", "newtest.txt");
}
```

Listing 4: Combined Program

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

void main() {
  if (fork()) {
    setuid(1000);
    close(creat("test.txt", S_IRWXU));
    rename("test.txt", "oldtest.txt");
    link("oldtest.txt", "test.txt");
    unlink("oldtest.txt");
    chmod("test.txt", S_IRUSR | S_IWUSR);
    int fp = open("test.txt", O_RDWR);
    write(fp, "TEST", 4);
    read(fp, NULL, 4);
    close(fp);
    truncate("test.txt", 0);
  }
}
```

C. Cypher query example

Listing 5: Cypher query for Category 2 of SPADE

```
MATCH
    (n1:VERTEX)-[r:EDGE]->(n2:VERTEX)
WHERE
    (
    (n1.type='Process' AND n2.type='Artifact')
    OR
    (n1.type='Artifact' AND n2.type='Process')
    )
    AND r.operation='<syscall>'
RETURN
    n1,n2,r
```

Listing 6: Cypher query for Category 3 of SPADE

```
MATCH
    (n1:VERTEX)-[r:EDGE]->(n2:VERTEX)
WHERE
    n1.type='Process'
    AND n2.type='Process'
    AND r.operation='<syscall>'
RETURN
    n1,n2,r
```

Listing 7: Cypher query for Category 4 of SPADE

```
MATCH
    (n1:VERTEX)-[r1:EDGE]->(n2:VERTEX),
    (n3:VERTEX)-[r2:EDGE]->(n1:VERTEX),
    (n3:VERTEX)-[r3:EDGE]->(n2:VERTEX)
WHERE
    n1.type='Process'
    AND n2.type='Artifact'
    AND n3.type='Artifact'
    AND r1.operation='<syscall>_read'
    AND r2.operation='<syscall>_write'
    AND r3.operation='<syscall>'
RETURN
    n1,n2,n3,r1,r2,r3
```

Listing 8: Cypher query for syscall with versioning of SPADE

```
MATCH
    (n1:VERTEX)-[r1:EDGE]->(n2:VERTEX),
    (n3:VERTEX)-[r2:EDGE]->(n1:VERTEX)
WHERE
    n1.type='Artifact'
    AND n2.type='Process'
    AND n3.type='Artifact'
    AND r1.operation='<syscall>'
    AND r2.operation='update'
RETURN
    n1,n2,n3,r1,r2
```

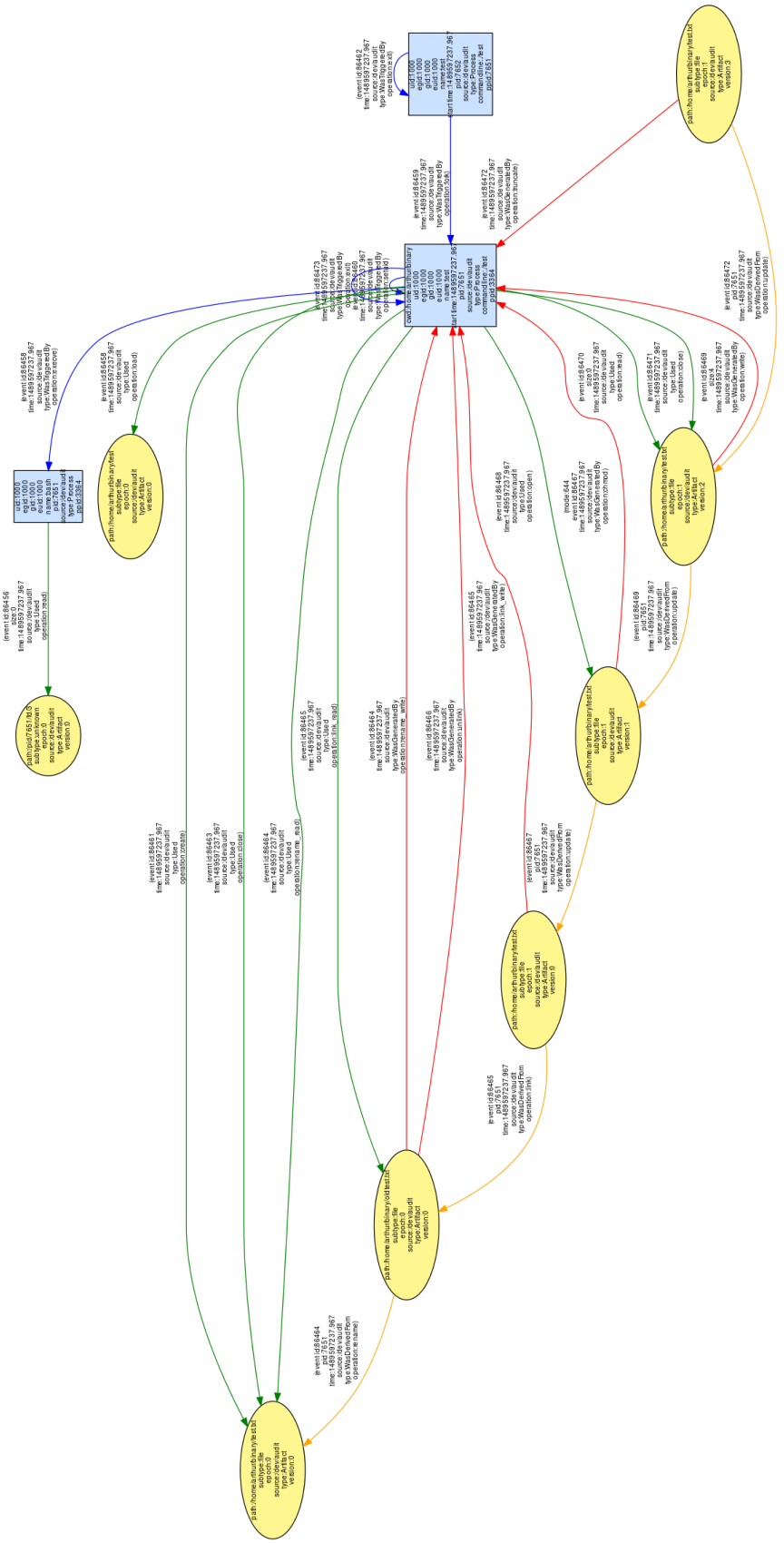

Figure 1: Provenance graph of Combined Program