ProvMark

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3361525.3361552

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Middleware '19 Proceedings of the 20th International Middleware Conference

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
We present an automated tool, ProvMark, that uses an existing vocabulary for provenance-related data, such standards do not specify how to record operating system-level behaviour, or even when such records are considered “accurate” or “complete”. Indeed, as we shall see, in practice there is little consensus about how specific activities (e.g., renaming a file) should be represented in a provenance graph. Additionally, different systems also work at different system layers (e.g., kernel space vs. user space), so some information may be unavailable to a given system.

To illustrate the problem, consider Figure 1, which shows three provenance graph representations of the same rename system call, as recorded by three different systems. These graphs clearly illustrate nontrivial structural differences in how rename is represented as communications between processes (blue rectangles) and artifacts or resources (yellow ovals).

Security analysts, auditors, or regulators seeking to use these systems face many challenges. Some of these challenges stem from the inherent size and complexity of provenance graphs: it is not unusual for a single day’s activity to generate graphs with millions of nodes and edges. However, if important information is missing, then these blind spots may render the records useless for their intended purpose. For example, if a provenance capture system does not record edges linking reads and writes to local sockets, then attackers can evade notice by using these communication channels. Such an omission could be due to a bug, but it could also result from misconfiguration, silent failure, or an inherent limitation of the recording system.

Provenance capture systems provide a broad-spectrum recording service, separate from the monitored applications, but (like any software system) they are not perfect, and can have their own bugs or idiosyncrasies. In order to rely on them for critical applications such as reproducible research, compliance monitoring, or intrusion detection, one must test their correctness and completeness. However, testing the correctness or completeness of provenance capture systems is challenging and currently done manually. In some cases there is not even a clear consensus about what behavior is correct. We present an automated tool, ProvMark, that uses an existing provenance graph structure recorded for a given activity, by a reduction to subgraph isomorphism problems handled by an external solver. ProvMark is a beginning step in the much needed area of testing and comparing the expressiveness of provenance systems. We demonstrate ProvMark’s usefulness in comparing three capture systems with different architectures and distinct design philosophies.
Kernel -> User Space

wards understanding what it means for system provenance to be volatile information such as timestamps and identifiers that vary how. Our main contributions are as follows:

- to show whether each system records the target activity, and if so
- system. Different runs of the same system can then be compared
- pressiveness benchmarking. ProvMark executes a given target op-
- complete and/or correct. It offers a systematic means to compare
- across runs, even when the underlying process is deterministic.
- Provenance records include volatile information such as timestamps and identifiers that vary across runs, even when the underlying process is deterministic. Furthermore, a process’s activity may include background process start-up activity that needs to be filtered out.

Expressiveness benchmarking is a first (but important) step towards understanding what it means for system provenance to be complete and/or correct. It offers a systematic means to compare different approaches. For example, how is a system call represented, and how do nodes and edges correspond to processes, files or file versions, or relationships among them? Are unsuccessful calls recorded? Are records of multiple similar calls aggregated together? By analyzing the same benchmark examples across different provenance capture systems, we can compare the information collected by different approaches to produce an understanding of the relative capabilities of the systems and their suitability for different tasks. Expressiveness benchmarking also enables regression testing since we can automatically compare benchmark graphs before and after a system change to detect differences.

We present ProvMark, an automated system for provenance expressiveness benchmarking. ProvMark executes a given target operation and records the resulting provenance captured by a given system. Different runs of the same system can then be compared to show whether each system records the target activity, and if so how. Our main contributions are as follows:

- We show how to generalize from multiple runs to obtain repeatable results, abstracting away volatile or transient data such as timestamps or identifiers.

2 Background and Related Work

Provenance [20] is a term originating in the art world. It refers to the chain of ownership of a work of art, including its original creator and each owner and location of the work from creation to the present. In the digital realm, provenance refers to information describing how an object came to be in its current form, typically including any other data from which it was derived and a description of the ways in which the input data was transformed to produce the output data. It is usually collected to allow post-hoc analysis to answer questions such as, "From where did this data come?", "On what objects does it depend?", or "What programs were used in its production?" Different applications of provenance will need different data, and systems are either tailored for specific applications or have a mechanism for selective capture, so that a user can obtain the right data for his/her intended use. Such flexibility makes it difficult to determine which system or configuration is most appropriate for a given task.

We apply ProvMark to three current provenance capture tools, namely SPADEV2 [12], OPUS [5] and CamFlow [26]. All three systems run under Linux. There are many other provenance capture systems. PASS is no longer maintained [23]; HiFi [31] was subsumed by LPM [6], which is similar to CamFlow, but less portable and not maintained. We focus on SPADE, OPUS and CamFlow as representative, currently available examples. Figure 2 shows how each system interacts with an application and the Linux kernel.

Figure 2. Architecture summary of the SPADE, OPUS, and CamFlow provenance recording systems

- We solve the required graph matching problems using an external solver, based on Answer Set Programming (ASP), a variation of logic programming.
- We evaluate ProvMark’s performance and effectiveness in testing and comparing provenance systems, highlighting several bugs or idiosyncrasies found so far.

ProvMark has been developed in consultation with developers of three provenance recording systems, SPADE, OPUS and CamFlow, several of whom are coauthors of this paper. They have validated the results and in some cases helped adapt their systems to ease benchmarking. The ProvMark system along with supplementary results is publicly available at http://provmark2018.github.io [8].
SPADEv2 (tag tc-e9) with the Linux Audit Reporter [15], which constructs a provenance graph using information from the Linux audit system (including the Audit service, daemon, and dispatcher). SPADE runs primarily in user space and provides many alternative configurations, including filtering and transforming the data, for example, to enable versioning or finer-grained tracking of I/O or network events, or to make use of information in proofs to obtain information about processes that were started before SPADE. We use a baseline configuration that collects data from the Audit Daemon (auditd), without versioning.

OPUS focuses on file system operations, attempting to abstract such operations and make the provenance collection process portable. It wraps standard C library calls with hooks that record provenance. The OPUS system is especially concerned with versioning support and proposes a Provenance Versioning Model, analogous to models previously introduced in the context of PASS [22] and later SPADE [13]. OPUS also runs in user space, but it relies on intercepting calls to a dynamically linked library (e.g., libc). Therefore, it is blind to activities that do not go through an intercepted dynamic library, but can observe the C library calls and userspace abstractions such as file descriptors.

CamFlow’s emphasis is sustainability through modularity, interfacing with the kernel via Linux Security Modules (LSM). The LSM hooks capture provenance, but then dispatch it to user space, via relays, for further processing. It strives for completeness and has its roots in Information Flow Control systems [29]. By default, CamFlow captures all system activity visible to LSM and relates different executions to form a single provenance graph; as we shall see, this leads to some complications for repeatable benchmarking. SPADE can also be configured to support similar behavior, while CamFlow can also be used (instead of Linux Audit) to report provenance to SPADE. Compared to SPADE and OPUS, which both run primarily in user space, CamFlow [26] monitors activity and generates the provenance graph from inside the kernel, via LSM and NetFilter hooks. This means the correctness of the provenance data depends on the LSM operation. As the rules are set directly on the LSM hooks themselves, which are already intended to monitor all security-sensitive operations, CamFlow can monitor and/or record all sensitive operations. CamFlow allows users to set filtering rules when collecting provenance.

Prior work [9, 21] on comparing different provenance systems has followed a manual approach. For example, the Provenance Challenge [21] proposed a set of scientific computation scenarios and solicited submissions illustrating how different capture systems handle these scenarios, to facilitate (manual) comparison of systems. More recently, Chan et al. proposed a pragmatic, but also manual, approach [9] to benchmarking OS-level provenance tracking at the level of individual system calls. However, these manual approaches are error-prone and not scalable. It is also worth mentioning that Pasquier et al. [27] perform static analysis showing a conservative over-approximation of the CamFlow provenance recorded for each call. However, these results are not yet automatically compared with actual run-time behavior.

Provenance expressiveness benchmarking is also related to the emerging topic of forensic-ready systems [2, 25, 34]. Work in this area considers how to add logging to an existing system to detect known classes of behaviors, particularly for legal evidence or regulatory compliance, or how to advise developers on how to add appropriate logging to systems in development. The provenance collection systems above strive for completeness so that previously-unseen behaviors can be detected, and proactively record information so that user applications do not need to be modified.

### 3 System Design and Methodology

ProvMark is intended to automatically identify the (usually small) subgraph of a provenance graph that is recorded for a given target activity. Target activities could consist of individual system calls (or syscalls), sequences of syscalls, or more general (e.g., concurrent) processes. For the moment, we consider the simplest case of a single syscall, but the same techniques generalize to deterministic sequential target activities; handling concurrency and nondeterminism are beyond the scope of this paper.

We call the targeted system call the `target call` and the corresponding subgraph the `target graph`. Naively, one might proceed by writing a simple C program for each target system call that just performs that call and nothing else. However, starting and ending a process creates considerable “boilerplate” provenance, including calls such as `fork`, `execve`, and `exit`, as well as accesses to program files and libraries and, sometimes, memory mapping calls. Furthermore, some target calls require other `prerequisite` calls to be performed first. For example, analyzing a `read` or `close` system call requires first performing an open. Process startup and prerequisite calls are both examples of `background activity` that we would like to elide.

In each benchmark, we use an `#ifdef TARGET` CPP directive to identify the target behavior of interest. ProvMark generates two executables for each such benchmark: a `foreground program` that includes all code in the benchmark program, including the target and any context needed for it to execute, and a `background program` that contains the background activities. The two binaries are almost identical; the difference between the resulting graphs should precisely capture the target behavior.

ProvMark includes a script for each syscall that generates and compiles the appropriate C executables and prepares a staging directory in which they will be executed with any needed setup, for example, first creating a file to run an `unlink` system call. We constructed these scripts manually since different system calls require different setup. The following code snippet illustrates the background program (including open) needed for the target `close` syscall (with `#ifdef` surrounding the target):

```c
// close.c
#include <fcntl.h>
#include <unistd.h>

void main() {
  int id = open("test.txt", O_RDWR);
  if (TARGET) close(id);
  #endif
}
```

Figure 3 provides an overview of ProvMark, which is composed of four subsystems: (1) recording, (2) transformation, (3) generalization, and (4) comparison. Users can select which provenance capture system to use, which benchmark to run, and other configuration settings, such as the number of trials. Before presenting the details of the four subsystems, we outline some common use cases for ProvMark.
Figure 3. ProvMark system overview. The recording stage (1) uses one of the provenance recorders to compute background graphs $bg_1$, $bg_2$ and foreground graphs $fg_1$, $fg_2$. (The same recorder is used for all four graphs.) The transformation stage (2) maps these graphs to a uniform Datalog format. The generalization stage (3) identifies the common structure of $bg_1$ and $bg_2$, resulting in $bg$, and likewise $fg_1$ and $fg_2$ are generalized to $fg$. Finally, $bg$ and $fg$ are compared (4); structure corresponding to $bg$ in $fg$ is removed, yielding the benchmark result.

3.1 Use Cases

People are beginning to build critical distributed applications, particularly security applications, using system-level provenance. However, it is difficult for them to know how to interpret results or implement queries to detect activity on different systems. Both potential users and developers of all three considered provenance recording tools have participated in the design of ProvMark. To clarify when, and to whom, ProvMark is useful, in this section we outline several common use cases. In each case, ProvMark automates a central, labor-intensive step: namely, running tools to extract provenance, graphs and analyzing the graphs produced by a tool to identify a target activity. The first two use cases are real (although fictionalized) situations where we have used ProvMark. The other two are more speculative, but illustrate how ProvMark could be used for testing or exploration.

Tracking failed calls  Alice, a security analyst, wants to know which provenance recorders track syscalls that fail due to access control violations, since these calls may be an indication of an attack or surveillance attempt. She can write small benchmark programs that capture various access control failure scenarios; most only take a few minutes to write, by modifying other, similar benchmarks for successful calls. Once this is done, ProvMark can run all of them to produce benchmark results. For example, Alice considers what happens if a non-privileged user unsuccessfully attempts to overwrite /etc/passwd by renaming another file.

By default SPADE installs Linux Audit rules that only report on successful system calls, so SPADE records no information in this case. OPUS monitors system calls via intercepting C library calls, so it knows whether a call is being attempted, and typically generates some graph structure even for unsuccessful calls. For example, the result of a failed rename call has the same structure as shown in Figure 1, but with a different return value property of -1 instead of 0. Finally, CamFlow can in principle monitor failed system calls, particularly involving permission checks, but does not do so in this case. Alice concludes that for her application, OPUS may provide the best solution, but resolves to discuss this issue with the SPADE and CamFlow developers as well.

Configuration validation  Bob, a system administrator, wants to make sure that an installation of SPADE is configured correctly to match a security policy. SPADE (like the other systems) has several configuration parameters, to allow greater or lesser levels of detail, coalescing similar operations (such as repeated reads or writes), or enabling/disabling versioning. These also affect performance and system overhead, so Bob wants to ensure that enough information is recorded to enable successful audits, while minimizing system load.

Bob can use ProvMark to benchmark alternative configurations of SPADE. For example, SPADE provides a flag `simplify` that is enabled by default. Disabling `simplify` causes setresgid and setresuid (among others) to be explicitly monitored, and Bob wants to ensure these calls are tracked. However, on doing so, Bob also uncovered a minor bug: when `simplify` is disabled, one of the properties of a background edge is initialized to a random value, which shows up in the benchmark as a disconnected subgraph. The bug was reported to the SPADE developers and quickly fixed.

SPADE also provides various filters and transformers which perform pre-processing or post-processing of the stored provenance respectively. Bob also experimented with one of SPADE’s filters, 10Runs, which controls whether runs of similar read or write operations are coalesced into a single edge. Surprisingly, Bob found that in the benchmarked version of SPADE, enabling this filter had no effect. This turned out to be due to an inconsistency in the property names used by the filter vs. those generated by SPADE. This has also now been fixed.

Regression testing  Charlie, a developer of provenance recording tool XYZTrace, wants to be able to document the level of completeness of XYZTrace to (potentially skeptical) users. ProvMark can be used for regression testing, by recording the graphs produced in a given benchmarking run, and comparing them with

Provenance systems: CamFlow, OPUS, SPADE, etc.
Graph handlers: Neo4j, GraphViz, ProvJSON, etc.
General Datalog format for bg and fg graphs
Graph comparison using Clingo
the results of future runs, using the same code for graph isomorphism testing ProvMark already uses during benchmarking. Charlie writes a simple script that stores the benchmark graphs (as Datalog) from previous runs, and whenever the system is changed, a new benchmarking run is performed and the results compared with the previous ones. When differences are detected, if the changes are expected then the new version of the graph replaces the old one; if the changes are unexpected, this is investigated as a potential bug.

**Suspicious activity detection** Dora, a security researcher, wants to identify patterns in provenance graphs that are indicative of a potential attack on the system. She compiles a set of scripts that carry out different kinds of attacks, and configures CamFlow on a virtual machine. She is particularly interested in detecting privilege escalation events where an attacker is able to gain access to new resources by subverting a privileged process. She instruments the scripts to treat the privilege escalation step as the “target activity”. Using ProvMark, Dora can obtain example provenance graphs that include the target activity or exclude it. If the graphs are not too large (e.g. hundreds rather than thousands of nodes), ProvMark’s default behavior will also be able to compute the differences between the graphs with and without the target activity.

### 3.2 Recording

The recording subsystem runs the provenance capture tools on test programs. This subsystem first prepares a staging directory that provides a consistent environment for test execution. The recording subsystem then starts the provenance capture tool with appropriate settings, captures the provenance generated by the tool, and stops the tool afterwards.

The recording subsystem is the only one to interact directly with the target provenance capture tools. For each tool, we implemented a script that configures the tool to capture the provenance of a specific process, rather than recording all contemporaneous system events. Recording multiple runs of the same process using CamFlow was challenging in earlier versions because CamFlow only serialized nodes and edges once, when first seen. The current version (0.4.5) provides a workaround to this problem that re-serializes the needed structures when they are referenced later. We also modified its configuration slightly to avoid tracking ProvMark’s own behavior. We otherwise use the default configuration for each system; thus the recording subsystem aims to manage the collecting tools by inserting and managing timeouts between the sessions. For example, we usually obtain stable results from SPADE by inserting timeouts to wait for successful completion of SPADE’s graph generation process; even so, we sometimes stop too early and obtain inconsistent results leading to mismatched graphs. Similarly, using CamFlow, we sometimes experience small variations in the size or structure of the results for reasons we have not been able to determine. In both cases we deal with this by running a larger number of trials and retaining the two smallest consistent results (as discussed later). For OPUS, any two runs are usually consistent, but starting OPUS and loading data into or out of Neo4j are time-consuming operations.

#### 3.3 Transformation

Different systems output their provenance graphs in different formats. For example, SPADE supports Graphviz DOT format and Neo4j storage (among others). OPUS also supports Neo4j storage, and CamFlow supports W3C PROV-JSON [18] as well as a number of other storage or stream processing backends. CamFlow also can be used instead of Linux Audit as a reporter to SPADE, though we have not yet experimented with this configuration. To streamline the remaining stages, we translate these three formats to a common representation. Unlike the recording stage, this stage really is straightforward, but we will describe the common format in detail because it is important for understanding the remaining stages.

The common target format is a logical representation of property graphs, in which nodes and edges can have labels as well as associated properties (key-value dictionaries). Specifically, given a set $Σ$ of node and edge labels, $Γ$ of property keys, and $D$ of data values, we consider property graphs $G = (V, E, src, tgt, lab, prop)$ where $V$ is a set of vertex identifiers and $E$ a set of edge identifiers; these are disjoint ($V \cap E = \emptyset$). Further, $src, tgt : E \rightarrow V$ maps each edge $e$ to its source and target nodes, respectively, $lab : V \cup E \rightarrow Σ$ maps each node or edge $x$ to its label $lab(x) \in Σ$, and $prop : (V \cup E) \times Γ \rightarrow D$ is a partial function such that for a given node or edge $x$, then $prop(x, p)$ (if defined) is the value for property $p \in Γ$. In practice, $Σ$, $Γ$, and $D$ are each sets of strings.

For provenance graphs, following the W3C PROV vocabulary, the node labels are typically `entity`, `activity` and `agent`, edge labels are typically relations such as `wasGeneratedBy` and `used`, and properties are either PROV-specific property names or domain-specific ones, and their values are typically strings. However, our representation does not assume the labels and properties are known in advance; it works with those produced by the tested system.

We represent property graphs as sets of logical facts using a Prolog-like syntax called Datalog [1], which is often used to represent relational data in logic programming (as well as databases [1] and networking [14]).

The generic form of the Datalog graph format we use is shown as Listing 1. We assume a fixed string gid used to uniquely identify a given graph as part of its node, edge and label relation names. Each node $v \in V$ is represented as a fact $n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle$. Likewise, each edge $e = (v, w) \in E$ is represented as a fact $e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$. 

```plaintext
Listing 1. Datalog Graph Format
Node n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle
Edge e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle
Property p\langle gid\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle
```

---

Listing 1. Datalog Graph Format

Node $n\langle \text{gid}\rangle\langle \text{nodeID}, \text{label}\rangle$

Edge $e\langle \text{gid}\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$

Property $p\langle \text{gid}\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle$

---

For provenance graphs following the W3C PROV vocabulary, the node labels are typically `entity`, `activity` and `agent`, edge labels are typically relations such as `wasGeneratedBy` and `used`, and properties are either PROV-specific property names or domain-specific ones, and their values are typically strings. However, our representation does not assume the labels and properties are known in advance; it works with those produced by the tested system.

We represent property graphs as sets of logical facts using a Prolog-like syntax called Datalog [1], which is often used to represent relational data in logic programming (as well as databases [1] and networking [14]).

The generic form of the Datalog graph format we use is shown as Listing 1. We assume a fixed string `gid` used to uniquely identify a given graph as part of its node, edge and label relation names. Each node $v \in V$ is represented as a fact $n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle$. Likewise, each edge $e = (v, w) \in E$ is represented as a fact $e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$. 

```plaintext
Listing 1. Datalog Graph Format
Node n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle
Edge e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle
Property p\langle gid\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle
```

---

Listing 1. Datalog Graph Format

Node $n\langle \text{gid}\rangle\langle \text{nodeID}, \text{label}\rangle$

Edge $e\langle \text{gid}\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$

Property $p\langle \text{gid}\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle$

---

For provenance graphs following the W3C PROV vocabulary, the node labels are typically `entity`, `activity` and `agent`, edge labels are typically relations such as `wasGeneratedBy` and `used`, and properties are either PROV-specific property names or domain-specific ones, and their values are typically strings. However, our representation does not assume the labels and properties are known in advance; it works with those produced by the tested system.

We represent property graphs as sets of logical facts using a Prolog-like syntax called Datalog [1], which is often used to represent relational data in logic programming (as well as databases [1] and networking [14]).

The generic form of the Datalog graph format we use is shown as Listing 1. We assume a fixed string `gid` used to uniquely identify a given graph as part of its node, edge and label relation names. Each node $v \in V$ is represented as a fact $n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle$. Likewise, each edge $e = (v, w) \in E$ is represented as a fact $e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$.

```plaintext
Listing 1. Datalog Graph Format
Node n\langle gid\rangle\langle \text{nodeID}, \text{label}\rangle
Edge e\langle gid\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle
Property p\langle gid\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle
```

---

Listing 1. Datalog Graph Format

Node $n\langle \text{gid}\rangle\langle \text{nodeID}, \text{label}\rangle$

Edge $e\langle \text{gid}\rangle\langle \text{edgeID}, \text{srcID}, \text{tgtID}, \text{label}\rangle$

Property $p\langle \text{gid}\rangle\langle \text{nodeID}/\text{edgeID}, \text{key}, \text{value}\rangle$
We wish to identify a single, representative and general graph for representative will be at least two. The third subsystem performs structure being found in the difference. leads to failure if the extra background structure is not found in also seems to work; the choice seems arbitrary. However, picking of graphs whose size is smallest. Picking the two largest graphs that are only similar to themselves, and consider these to be failed representative graphs, we first consider all of the trial runs and a strategy for identifying an appropriate pair below. To obtain two other. Identifying two representative runs is complicated by the structure being found in the difference.

Moreover, we say that \( h \) is isomorphic to \( G_1 \) and (as we shall see) the ASP solver can efficiently find solutions. Because the graphs and specification can be represented concisely, requirements of graph similarity. ASP is well-suited to this problem because the graphs and specification can be represented concisely, and (as we shall see) the ASP solver can efficiently find solutions. The problem specification is a logic program defining when a model is a possible solution. In our setting, the models consist of matching relations between two graphs, satisfying the requirements of graph similarity. ASP is well-suited to this problem because the graphs and specification can be represented concisely, and (as we shall see) the ASP solver can efficiently find solutions. Given two similar graphs, the generalization stage identifies the property values that are consistent across the two graphs and removes the transient ones. The generalization stage searches for a matching of nodes and edges of the two graphs that minimizes the number of different properties. We assume that the differences are transient data and discard them. We would like to minimize the number of differences, that is, match as many properties as possible.

Similarity and generalization are instances of the graph isomorphism problem, whose status (in P or NP-complete) is unknown [4]. We solve these problems using an Answer Set Programming (ASP) [11] specification that defines the desired matching between the two graphs. ASP is a decidable form of logic programming that combines a high-level logical specification language with efficient search and optimization techniques analogous to SAT solving or integer linear programming. An answer set program specifies a set of possible models, which correspond to solutions to a problem. The specification is a collection of logical formulas that define when a model is a possible solution. In our setting, the models consist of matching relations between two graphs, satisfying the requirements of graph similarity. ASP is well-suited to this problem because the graphs and specification can be represented concisely, and (as we shall see) the ASP solver can efficiently find solutions.

The problem specification is a logic program defining when a binary relation forms an isomorphism between the graphs. The code in Listing 3 defines graph isomorphism in ASP. ASP specifications consist of rules that constrain the possible solution set. The first four lines specify that \( h \) can relate any node in \( G_1 \) to any node in \( G_2 \) and vice versa, and similarly for edges. The next two lines constrain \( h \) to be a 1-1 function. (Rules of the form \( \neg \) must be read as “It cannot be the case that \( A, B, C \) hold.”) The remaining three pairs of lines specify that \( h \) must preserve node and edge labels and the source and targets of edges.

This specification can be solved using clingo, an efficient ASP solver [11]. As ASP is a kind of logic programming, it helps ProvMark to determine an optimal matching solution among two graphs by searching for a model that satisfies the specification. We use the resulting matching to determine which properties are common across both graphs and discard the others. We perform generalization independently on the foreground and background graphs. The
We present a demonstration of using ProvMark to investigate and compare the behavior of the different capture systems. These results pertain to SPADev2 (tag tc-e3) and OPUS version 0.1.0.26 running on Ubuntu 16.04 and CamFlow version 0.4.5 running on Fedora 27.

Unix-like operating systems support over three hundred system calls. We focus on a subset of 22 common system call families shown in Table 1, including the most common file and process management calls. We prepared benchmarking programs for these syscalls, along with tests for each one to ensure that the target behavior was performed successfully. Note that the same system call may display different behavior using different parameters and system states; we focus on common-case behavior here, but handling other scenarios such as failure cases is straightforward as outlined in Section 3.1.

We cannot show all of the benchmark graphs, nor can we show the graphs at a readable size; the full results are available for inspection online [8]. Table 2 summarizes the results. In this table, “ok” means the graph is correct (according to the system’s developers), and “empty” means the foreground and background graphs were similar and so the target activity was undetected. The table includes notes indicating the reason for emptiness. Also, Table 3 shows selected graphs (again, images are links to scalable online images with full details) for the benchmark result. Although the property keys and values are omitted, we can compare the graph structure and see that the three tools generate very different structures for the same system calls. This demonstrates one of the important motivations for expressiveness benchmarking. In these graph results, the yellow ovals represent artifacts which includes files, pipes or other resources. The blue rectangles represent processes involved in the system calls. The green or gray ovals represent dummy nodes which stand for pre-existing parts of the graph. We retain these dummy nodes to make the result a complete graph.

In this section we present and discuss the benchmarking outputs for several representative syscalls, highlighting minor bugs found and cases where the behavior of individual tools or the variation among different tools was surprising.

### 4.1 File system

File system calls such as creat, open, close are generally covered well by all of the tools, but they each take somewhat different approaches to representing even simple behavior. For example, for the open call, SPADE adds a single node and edge for the new file, while CamFlow creates a node for the file object, a node for its path, and several edges linking them to each other and the opening process, and OPUS creates four new nodes including two nodes corresponding to the file. On the other hand, reads and writes appear similar in both SPADE and CamFlow, while by default, OPUS does not record file reads or writes. For close, CamFlow records the underlying kernel data structures eventually being freed, but ProvMark does not reliably observe this.

### Listing 4. Approximate subgraph isomorphism

\[
(h(X,Y) : n2(Y,\_)) \iff 1 : = n1(X,\_).
\]

\[
(h(X,Y) : e2(Y,\_,\_)) \iff 1 : = e1(X,\_,\_).
\]

\[
\begin{align*}
& X \not\iff Y, h(X, Z), h(Y, Z). \\
& X \not\iff Y, h(Z, Y), h(Z, X).
\end{align*}
\]

\[
\begin{align*}
& X \not\iff Y, h(X, L), h(Y, X), \not\iff n2(Y, L). \\
& X \not\iff Y, h(E1, E2), \not\iff e2(E2, \_, \_, L).
\end{align*}
\]

\[
\begin{align*}
& e1(E1, E2), h(E1, E2), e2(E2, \_, \_, \_), \not\iff h(X, Y). \\
& e1(E1, \_, \_, \_), h(E1, E2), \not\iff e2(E2, \_, \_, \_), \not\iff h(X, Y).
\end{align*}
\]

\[
\begin{align*}
& \text{cost}(X, K, \theta) : = p1(X, K, Y), h(X, Y), p2(Y, K, V), V \not\iff W. \\
& \text{cost}(X, K, 1) : = p1(X, K, Y), h(X, Y), p2(Y, K, W), V \not\iff W.
\end{align*}
\]

\[
\begin{align*}
& \text{cost}(X, K, 1) : = p1(X, K, Y), h(X, Y), p2(Y, K, W), V \not\iff W.
\end{align*}
\]

\[
\begin{align*}
& \#\text{minimize} \{ PC, X, K : \text{cost}(X, K, PC) \}.
\end{align*}
\]
The dup[2, 3] syscall duplicates a file descriptor so that the process can use two different file descriptors to access the same open file. SPADE and CamFlow do not record this call directly, but the changes to the process’s file descriptor state can affect future file system calls that are monitored. OPUS does record dup[2, 3]. The two added nodes are not directly connected to each other, but are connected to the same process node in the underlying graph. One component records the system call itself and the other one records the creation of a new resource (the duplicated file descriptor).

The link[at] call is recorded by all three systems but symlink[at] is not recorded by CamFlow 0.4.5. Likewise, mknod is only recorded by OPUS, and mknodat is not recorded by any of the systems. The rename[at] call, as shown in the introduction, illustrates how the three systems record different graph structure for this operation. SPADE represents a rename using two nodes for the new and old filenames, with edges linking them to each other and to the process that performed the rename. OPUS creates around a dozen nodes, including nodes corresponding to the rename call itself and the new and old filename. CamFlow represents a rename as adding a new path associated with the file object; the old path does not appear in the benchmark result.

### 4.2 Process management

The process management operations start processes ([v]fork, clone), replace a process’s memory space with a new executable and execute it (execve), or terminate a process (kill, exit).

Process creation and initialization calls are generally monitored by all three tools, except that OPUS does not appear to monitor clone. Inspecting the results manually shows significant differences, however. For example, the SPADE benchmark graph for execve is large, but has just a few nodes for OPUS and CamFlow. On the other hand, the fork and vfork graphs are small for SPADE and CamFlow and large for OPUS. This may indicate the different perspectives of these tools: SPADE relies on the activity reported by Linux Audit, while OPUS relies on intercepting C library calls outside the kernel.

Another interesting observation is that the SPADE benchmark results for fork and vfork differ. Specifically, for vfork, SPADE represents the forked process as a disconnected activity node, i.e. there is no graph structure connecting the parent and child process. This is because Linux Audit reports system calls on exit, while the parent process that called vfork is suspended until the child process exits. SPADE sees the vforked child process in the log executing its own syscalls before it actually sees the vfork that created the child process.

The exit and kill calls are not detected because they deviate from the assumptions our approach is based on. A process always has an implicit exit at the end, while killing a process means that it does not terminate normally. Thus, the exit and kill benchmarks are all empty. We plan to consider a more sophisticated approach that can benchmark these calls in future work.

### 4.3 Permissions

We included syscalls that change file permissions (such as [f]chown[at], [f]chmod[at]) or process permissions in this group. According to its documentation, SPADE currently records [f]chmod[at] but not [f]chown[at]. OPUS does not monitor fchmod or fchown because from its perspective these calls only perform read/write activity and do not affect the process’s file descriptor state, so as for other read/write activity OPUS does not record anything in its default configuration. CamFlow records all of these operations.

In its default configuration, SPADE does not explicitly record seteuid or setregid, but it does monitor changes to these process attributes and records observed changes. Our benchmark result for seteuid is nonempty, reflecting an actual change of

<table>
<thead>
<tr>
<th>Group</th>
<th>syscall</th>
<th>SPADE</th>
<th>OPUS</th>
<th>CamFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>close</td>
<td>ok</td>
<td>ok</td>
<td>empty (LP)</td>
</tr>
<tr>
<td>1</td>
<td>creat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>dup</td>
<td>empty (SC)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>dup2</td>
<td>empty (SC)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>dup3</td>
<td>empty (SC)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>link</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>linkat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>symlink</td>
<td>ok</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>symlinkat</td>
<td>ok</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>mknod</td>
<td>empty (NR)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>mknodat</td>
<td>empty (NR)</td>
<td>empty (NR)</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>1</td>
<td>open</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>openat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>read</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>pread</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>rename</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>renameat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>truncate</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>ftruncate</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>unlink</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>unlinkat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>writeat</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>2</td>
<td>clone</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>2</td>
<td>execve</td>
<td>empty (LP)</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>2</td>
<td>exit</td>
<td>empty (LP)</td>
<td>empty (LP)</td>
<td>empty (LP)</td>
</tr>
<tr>
<td>2</td>
<td>fork</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>2</td>
<td>kill</td>
<td>empty (LP)</td>
<td>empty (LP)</td>
<td>empty (LP)</td>
</tr>
<tr>
<td>2</td>
<td>vfork</td>
<td>ok (DV)</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>chmod</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>fchmod</td>
<td>ok</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>fchmodat</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>chown</td>
<td>empty (NR)</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>fchown</td>
<td>empty (NR)</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>fchownat</td>
<td>empty (NR)</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>setgid</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>setregid</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>seteuid</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>3</td>
<td>setresuid</td>
<td>ok (SC)</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
<tr>
<td>4</td>
<td>pipe</td>
<td>empty (NR)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>4</td>
<td>pipe2</td>
<td>empty (NR)</td>
<td>ok</td>
<td>empty (NR)</td>
</tr>
<tr>
<td>4</td>
<td>tee</td>
<td>empty (NR)</td>
<td>empty (NR)</td>
<td>ok</td>
</tr>
</tbody>
</table>

**Table 2. Summary of validation results**

The dup[2, 3] syscall duplicates a file descriptor so that the process can use two different file descriptors to access the same open file. SPADE and CamFlow do not record this call directly, but the changes to the process’s file descriptor state can affect future file system calls that are monitored. OPUS does record dup[2, 3]. The two added nodes are not directly connected to each other, but are connected to the same process node in the underlying graph. One component records the system call itself and the other one records the creation of a new resource (the duplicated file descriptor).

The link[at] call is recorded by all three systems but symlink[at] is not recorded by CamFlow 0.4.5. Likewise, mknod is only recorded by OPUS, and mknodat is not recorded by any of the systems. The rename[at] call, as shown in the introduction, illustrates how the three systems record different graph structure for this operation. SPADE represents a rename using two nodes for the new and old filenames, with edges linking them to each other and to the process that performed the rename. OPUS creates around a dozen nodes, including nodes corresponding to the rename call itself and the new and old filename. CamFlow represents a rename as adding a new path associated with the file object; the old path does not appear in the benchmark result.

### 4.2 Process management

The process management operations start processes ([v]fork, clone), replace a process’s memory space with a new executable and execute it (execve), or terminate a process (kill, exit).

Process creation and initialization calls are generally monitored by all three tools, except that OPUS does not appear to monitor clone. Inspecting the results manually shows significant differences, however. For example, the SPADE benchmark graph for execve is large, but has just a few nodes for OPUS and CamFlow. On the other hand, the fork and vfork graphs are small for SPADE and CamFlow and large for OPUS. This may indicate the different perspectives of these tools: SPADE relies on the activity reported by Linux Audit, while OPUS relies on intercepting C library calls outside the kernel.

Another interesting observation is that the SPADE benchmark results for fork and vfork differ. Specifically, for vfork, SPADE represents the forked process as a disconnected activity node, i.e. there is no graph structure connecting the parent and child process. This is because Linux Audit reports system calls on exit, while the parent process that called vfork is suspended until the child process exits. SPADE sees the vforked child process in the log executing its own syscalls before it actually sees the vfork that created the child process.

The exit and kill calls are not detected because they deviate from the assumptions our approach is based on. A process always has an implicit exit at the end, while killing a process means that it does not terminate normally. Thus, the exit and kill benchmarks are all empty. We plan to consider a more sophisticated approach that can benchmark these calls in future work.

### 4.3 Permissions

We included syscalls that change file permissions (such as [f]chown[at], [f]chmod[at]) or process permissions in this group. According to its documentation, SPADE currently records [f]chmod[at] but not [f]chown[at]. OPUS does not monitor fchmod or fchown because from its perspective these calls only perform read/write activity and do not affect the process’s file descriptor state, so as for other read/write activity OPUS does not record anything in its default configuration. CamFlow records all of these operations.

In its default configuration, SPADE does not explicitly record seteuid or setregid, but it does monitor changes to these process attributes and records observed changes. Our benchmark result for seteuid is nonempty, reflecting an actual change of
Table 3. Example benchmark results for SPADE, OPUS and CamFlow. Images are links to scalable online images with property labels.

<table>
<thead>
<tr>
<th>Open</th>
<th>Read</th>
<th>Write</th>
<th>Dup</th>
<th>Setuid</th>
<th>Setresuid</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPADE</td>
<td><img src="open.png" alt="Image" /></td>
<td><img src="read.png" alt="Image" /></td>
<td><img src="write.png" alt="Image" /></td>
<td>Empty</td>
<td><img src="setuid.png" alt="Image" /></td>
</tr>
<tr>
<td>OPUS</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td><img src="setuid.png" alt="Image" /></td>
<td>Empty</td>
</tr>
<tr>
<td>CamFlow</td>
<td><img src="open.png" alt="Image" /></td>
<td><img src="read.png" alt="Image" /></td>
<td><img src="write.png" alt="Image" /></td>
<td>Empty</td>
<td><img src="setuid.png" alt="Image" /></td>
</tr>
</tbody>
</table>

user id, while our benchmark for setresgid just sets the group id attribute to its current value, and so this activity is not noticed by SPADE. OPUS also does not track these two calls, while CamFlow again tracks all of them.

### 4.4 Pipes
Pipes provide efficient interprocess communication. The `pipe[2, 3]` call creates a pipe, which can be read or written using standard file system calls, and `tee` duplicates information from one pipe into another without consuming it. Of the three systems, only OPUS records `pipe[2, 3]` calls, while only CamFlow currently records `tee`.

### 5 System Evaluation

In this section, we will evaluate ProvMark. We claim that the ProvMark system makes it easy to test and compare the behavior of provenance tracking systems and to assess their completeness and correctness. The previous section gave some evidence that the results are useful for understanding the different systems and what information they do or do not collect. In this section, we will evaluate the performance and extensibility of ProvMark itself. For a fair comparison, all the experiments are done in a virtual machine with 1 virtual CPU and 4GB of virtual memory. Dependencies for all three provenance collecting tools, including Neo4J and some needed Python libraries, are also installed. All virtual machines were hosted on an Intel i5-4570 3.2GHz with 8GB RAM running Scientific Linux 7.5.

#### 5.1 System Performance

We first discuss the performance of the ProvMark system. Some performance overhead is acceptable for a benchmarking or testing tool; however, we need to establish that ProvMark’s strategy for solving NP-complete subgraph isomorphism problems is effective in practice (i.e. takes minutes rather than days). In this section, we will show that the extra work done by ProvMark to generate benchmarks from the original provenance result is acceptable.

We first report measurements of the recording time. Since the number of trials varies, we report the average time needed per trial. For SPADE, recording took approximately 20s per trial. For OPUS, the recording time was around 28s per trial, and for CamFlow each trial took around 10s. We wish to emphasize that the recording time results are not representative of the recording times of these systems in normal operation — they include repeatedly starting, stopping, and waiting for output to be flushed, and the waiting times are intentionally conservative to avoid garbled results. No conclusions about the relative performance of the recording tools when used as intended should be drawn.

In Figures 5–7, we summarize the time needed for ProvMark to run five representative syscalls using SPADE, OPUS, and CamFlow respectively. The bars are divided into three portions, representing the transformation, generalization and comparison subsystems. Note that the x-axes are not all to the same scale: in particular the transformation, generalization and comparison times for OPUS are much higher than for the other tools because of database startup and access time, and because the graphs extracted from the database are larger. Modifying OPUS to circumvent the database and serialize provenance directly would avoid this cost, but we have chosen to use the tools as they are to the greatest extent possible.

From the data in Figures 5–7 we can see that the transformation stage is typically the most time-consuming part. The transformation stage maps the different provenance output formats to Datalog.
format. The transformation time for OPUS is much longer than for
the other two systems. This appears to be because OPUS produces
larger graphs (including recording environment variables), and
extracting them involves running Neo4j queries, which also has a
one-time JVM warmup and database initialization cost.

The time required for the generalization stage depends mainly
on the number of elements (nodes and edges) in the graph, since
this stage matches elements in pairs of graphs to find an isomorphic
graph matching. As we can see from the results, the generalization
of OPUS graphs again takes significantly longer than for SPADE
and CamFlow, probably because the graphs are larger. For SPADE,
generalization of the execve benchmark takes much longer than
for other calls (though still only a few seconds).

The time required for the comparison stage is usually less than
for generalization (recall that in generalization we process two pairs
of graphs whereas in comparison we just compare the background
and foreground graphs). Comparison of OPUS graphs again takes
longest, while comparison of CamFlow graphs again takes slightly
longer than for SPADE, perhaps because of the larger number of
properties.

In general, we can conclude that the time needed for ProvMark’s
transformation, generalization, and comparison stages is acceptable
compared with the time needed for recording. Most benchmarks
complete within a few minutes at most, though some that produce
larger target graphs take considerably longer; thus, at this point
running all of the benchmarks takes a few hours. This seems like an
acceptable price to pay for increased confidence in these systems,
and while there is clearly room for improvement, this compares
favorably with manual analysis, which is tedious and would take
a skilled user several hours. In addition, we have monitored the
memory usage and found that ProvMark never used more than 75%
of memory on a 4GB virtual machine, indicating memory was not
a bottleneck.

5.2 Scalability
Our experiments so far concentrated on benchmarking one syscall
to a target activity consists of multiple syscalls,
the time needed for ProvMark to process results and solve the
resulting subgraph isomorphism problems will surely increase. The
generated provenance graph will also increase in size and number
of elements. The NP-complete complexity of sub-graph isomor-
phism means that in the worst case, the time needed to solve larger
problem instances could increase exponentially. This section in-
vestigates the scalability of ProvMark when the size of the target
action increases.

In figure 8-10, the charts show the time needed for the three pro-
cessing subsystems of ProvMark in handling some scalability test
cases on SPADE, OPUS and CamFlow respectively. The scalability
test cases range from scale1 to scale8. In test case scale1, the target
action sequence is simply a creation of a file and another deletion
of the newly created file. In test case scale2, scale4 and scale8, the
same target action is repeated twice, four times, and eight times
respectively. The results show that the time needed initially grows
slowly for SPADE, but by scale8 the time almost doubles compared
to scale1. For OPUS, the time increases are dwarfed by the high
overhead of transformation, which includes the one-time Neo4j
startup and access costs as discussed above. For CamFlow, the time
needed approximately doubles at each scale factor. Although fur-
ther scalability experiments are needed to consider much larger
design a tool that handles short sequences of 10-20 syscalls
without problems.

5.3 Modularity and Extensibility
ProvMark was designed with extensibility in mind. Only the first
two stages (recording and generalization) depend on the details
of the recording system being benchmarked, so to support a new
system, it suffices to implement an additional recording module that
uses the new system to record provenance for a test executable, and
(if needed) an additional translation module that maps its results
to Datalog.

As discussed earlier, it was non-trivial to develop recording mod-
ules for the three systems that produce reliable results. In particular,
supporting CamFlow required several iterations and discussion
with its developers, which have resulted in changes to CamFlow
to accommodate the needs of benchmarking. This architecture
has been refined through experience with multiple versions of the
SPADE and CamFlow systems, and we have generally found the
changes needed to ProvMark to maintain compatibility with new
versions to be minor. Developing new recording modules ought to
be straightforward for systems that work in a similar way to one
of the three currently supported.

If a provenance tool generates data in a format not currently
supported by ProvMark, an additional module for handling this type
of provenance data is needed. The need for new transformations
should decrease over time as more tools are introduced to ProvMark
since there are a limited number of common provenance formats.
As shown in Table 4, none of the three recording or transformation modules required more than 200 lines of code (Python 3 using only standard library imports). We initially developed ProvMark with support for SPADE/DOT and OPUS/Neo4j combinations; thus, adding support for CamFlow and PROV-JSOn only required around 330 lines of code.

Finally, throughout the development period of ProvMark, the candidate tools have all been updated and include new features and behaviour. Over time, we have updated ProvMark to react to changes in both SPADE and CamFlow, with few problems.

5.4 Discussion and Limitations

While we have argued that ProvMark is effective, fast enough to produce timely results, and easy to extend, it also has some limitations, which we now discuss.

At this stage, ProvMark has only been applied to provenance tracking at the operating-system level; in particular, for SPADE, OPUS and CamFlow running under Linux. We believe the same approach can be adapted to other operating systems or other layers of distributed systems, assuming an extension to deal with nondeterministic or concurrent activity as described below.

We have evaluated ProvMark using a small subset of individual syscalls. Creating benchmarks currently takes some manual effort and it would be helpful to automate this process to ease the task of checking all of the possible control flow paths each syscall can take, or even generate multi-step sequences. In addition, the analysis and comparison of the resulting benchmark graphs requires some manual effort and understanding.

We have shown ProvMark is capable of handling benchmark programs with a small number of target system calls. Scaling up to larger amounts of target activity or background activity appears challenging, due to the NP-completeness and worst-case exponential behavior of subgraph isomorphism testing. However, for deterministic activity it may be possible to take advantage of other structural characteristics of the provenance graphs to speed up matching: for example, if matched nodes are usually produced in the same order (according to timestamps), then it may be possible to incrementally match the foreground and background graphs.

Lastly, ProvMark currently handles deterministic activity only. Nondeterminism (for example through concurrency) introduces additional challenges: both the foreground and background programs might have several graph structures corresponding to different schedules, and there may be a large number of different possible schedules. It also appears necessary to perform some kind of fingerprinting or graph structure summarization to group the different possible graphs according to schedule. We may also need to run larger numbers of trials and develop new ways to align the different structures so as to obtain reliable results. It appears challenging to ensure completeness, that is, that all of the possible behaviors are observed, especially since the number of possible schedules can grow exponentially in the size of the program.

6 Conclusions and Future Work

This paper presents ProvMark, an automated approach to benchmarking and testing the behavior of provenance capture systems. To the best of our knowledge it is the first work to address the unique challenges of testing and comparing provenance systems. We have outlined the design of ProvMark, and showed how it helps identify gaps in coverage, idiosyncrasies and even a few bugs in three different provenance capture systems. We also showed that it has acceptable performance despite the need to solve multiple NP-complete graph matching subproblems. ProvMark is a significant step towards validation of such systems and should be useful for developing correctness or completeness criteria for them.

There are several directions for future work, to address the limitations discussed in Section 5.4. First, additional support for automating the process of creating new benchmarks, or understanding and interpreting the results, would increase the usefulness of the system, as would extending it to provenance tracking at other levels, such as distributed system coordination layers. Second, although we have evaluated scalability up to 10–20 target system calls, realistic applications such as suspicious behavior analysis would require dealing with much larger target activities and graphs. Finally, ProvMark cannot deal with nondeterminism, including concurrent I/O and network activity, and extending its approach to handle nondeterministic target activity is a focus of current work.

Acknowledgments

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. Cheney was also supported by ERC Consolidator Grant Skye (grant number 682315). This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contract FA8650-15-C-7557 and the National Science Foundation under Grant ACI-1547467 and SII-1450277 (End-to-End Provenance). 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.

We are grateful to Ajitha Rajan for comments on a draft of this paper and to our shepherd Boris Koldehofe and anonymous reviewers for insightful comments.

References

A ProvMark Documentation

ProvMark is a fully automated system that generates system level provenance benchmarks for different provenance collection tools and systems, such as SPADE, OPUS and CamFlow. It is the main tool mentioned in the paper. This section provides documentation for how to use it and access the source code of ProvMark.

A.1 ProvMark source and release page

The source code of ProvMark is stored in a Github repository and the link is here\(^1\). ProvMark is still undergoing development. There are many new updates after the publication of this paper. In order for the reader of this paper to use or test ProvMark with all the features mentioned in this paper, we have provided a release version for this Middleware submission. The link is here\(^2\). In this release page, documentation on how to install, configure and use the ProvMark tool has been provided. There is also a tarball with a version of ProvMark that we are mentioning in this paper. In addition, a set of sample results and additional experimental timing results is provided in this page.

A.2 Directory structure of ProvMark source

- **benchmarkProgram**: Contains sample c programs for the collection of provenance information of different syscalls
- **clingo**: Contains the clingo code
- **config**: Contains the configuration profile of different tool choices for stage 1 and stage 2
- **documentation**: Contains the documentation for ProvMark
- **genClingoGraph**: Contains code to transform graph format
- **processGraph**: Contains code to handle graph comparison and generalization
- **sampleResult**: Contains sample benchmark result on our trial
- **startTool**: Contains tools to handle provenance collecting tools currently supported and retrieve result from them
- **template**: Contains html template for result generation
- **vagrant**: Contains vagrant files for those provenance collecting tools currently supported

A.3 ProvMark Installation

Installing ProvMark is simple, just clone the whole git repository. The current stable version corresponding to this paper is tagged by the provenance systems. It can increase the accuracy for the resulting benchmark, but it will decrease the efficiency.

Each profile defines one supporting tool and its configuration. If new tools are supported in ProvMark, a new profile will be created here.

A.3.1 Vagrant installation

In the vagrant folder, we have prepared the Vagrant\(^3\) script for the three provenance collecting tools currently supported. If you have vagrant (v2.2.2 or greater) and virtual box (v6.0 or greater) installed in your system, you can follow the steps below to build up a virtual environment which everything (tools and ProvMark) are installed.

A.3.2 Vagrant script for SPADE

```bash
cd ./vagrant/spade
vagrant plugin install vagrant-vbguest
vagrant up
vagrant ssh
```

A.3.3 Vagrant script for OPUS

```bash
cd ./vagrant/opus
vagrant plugin install vagrant-vbguest
vagrant up
vagrant ssh
```

To run OPUS, you also need a source or binary distribution for the OPUS system itself, which is available here\(^4\).

A.3.4 Vagrant script for CamFlow

```bash
cd ./vagrant/camflowv045
vagrant plugin install vagrant-vbguest
vagrant up
vagrant halt
vagrant up
vagrant ssh
```

It is necessary to reboot the VM (halt / up) so that the camflow-enabled kernel will be used. This kernel should be highlighted as the first entry by the boot loader but if not, it should be selected.

After the above steps, you should be given a ssh connection to the virtual machine from which you can start ProvMark on your chosen tools directly. Note: the installation process can take an extended amount of time depending on your configuration.

A.4 ProvMark configuration

Configuration of ProvMark is done by modifying the config/config.ini file. Changing this file should not be necessary in common cases.

In the ProvMark system, the first two stages collect provenance information from provenance collecting tools and transform the provenance result into a clingo graph. Different tools need different process handling and type conversion. These configuration parameters are stored in the config.ini file.

Each profile starts with the name and includes three settings as follows:

- **stage1tool**: define the script (and parameter) for stage 1 handling when this profile is chosen
- **stage2handler**: define which graph handler is used to handle the raw provenance information when this profile is chosen
- **filtergraphs**: define if graph filtering is needed. The default value for SPADE and OPUS is false and true for CamFlow. The graph filtering is a mechanism provided by ProvMark to filter out obviously incomplete or incorrect graphs generated by the provenance systems. It can increase the accuracy for the resulting benchmark, but it will decrease the efficiency.

Each profile defines one supporting tool and its configuration. If new tools are supported in ProvMark, a new profile will be created here.

---

\(^1\)https://github.com/arthurscchan/ProvMark/
\(^2\)https://github.com/arthurscchan/ProvMark/releases/tag/Middleware2019
\(^3\)https://www.vagrantup.com/
\(^4\)https://github.com/DTG-FRESCO/opus
A.5 ProvMark usage

A.5.1 Parameters

Currently Supported Tools:
- spg SPADE with Graphviz storage
- spn SPADE with Neo4j storage
- opu OPUS
- cam CamFlow

Tools Base Directory:
Base directory of the chosen tool, it is assumed that if you want to execute this benchmarking system on certain provenance collecting tools, you should have installed that tools with all dependencies required by the tools. If you build ProvMark with the given Vagrant script, the default tools base directory is shown as follows.

SPADE /home/vagrant/SPADE
OPUS /home/vagrant/opus (or where OPUS was manually installed)
CamFlow ./ (this parameter is ignored but some value must be provided)

Benchmark Directory:
Base directory of the benchmark program.
Point the script to the syscall choice for the benchmarking process.

Number of trials (Default: 2):
Number of trials executed for each graph for generalization.
More trials will result in longer processing time, but provide a more accurate result as multiple trials can help to filter out uncertainty and unrelated elements and noise.

Result Type:
- rb benchmark only
- rg benchmark and generalized foreground and background graph only
- rh html page displaying benchmark and generalized foreground and background graph

A.5.2 Single Execution
This command will only call a specific benchmark program and generate benchmark with the chosen provenance system.
Usage:
./fullAutomation.py <Tools> <Tools Base Directory> <Benchmark Directory> [<Trial>]

Sample:
./fullAutomation.py cam ./ /benchmarkProgram/baseSyscall/grpCreat/cmdCreat 2

A.5.3 Batch Execution
Automatically execute ProvMark for all syscalls currently supported. The runTests.sh script will search for all benchmark programs recursively in the default benchmarkProgram folder and benchmark them one by one. It will also group the final result and post process them according to the given result type parameter.
Usage:
./runTests.sh <Tools> <Tools_Path> <Result Type>

Sample:
./runTests.sh spg /home/vagrant/SPADE rh

For more examples of the usage please refer to the documentation provided on the release page or the README file at the parent directory of the source code.

A.6 Sample Output

For the generation of the sample output, we have used the provided Vagrant script to build up the environment for the three provenance systems and execute a batch execution in each of the built virtual machine. The following command is used in each virtual machine respectively.

A.6.1 SPADE
./runTests.sh spg /home/vagrant/SPADE rh

A.6.2 OPUS
./runTests.sh opu /home/vagrant/opus rh

A.6.3 CamFlow
./runTests.sh cam . rh 11

The results on successful completion of this script are accessible in finalResult/index.html.

A.6.4 Full Timing Result

From the data in Figures 5–7 we can see the time needed for ProvMark to process some of the system calls using specific provenance systems. The full list of timing result containing all system calls included in the above batch execution can be found in the following path inside the source tarball from the release page. They are separated by the chosen provenance systems.

SPADE sampleResult/spade.time
OPUS sampleResult/opus.time
CamFlow sampleResult/camflow.time

Each line in those result files representing one system call execution and all parameters in each line are separated by a comma. The first two parameters represent the provenance system and system calls chosen for this line of result. The remaining four floating-point numbers represent the time needed (in seconds) for each ProvMark subsystem to process that system call (with the chosen provenance system) in order. ProvMark will automatically time all the process and attached a line of timing result at the end of /tmp/time.log for each system call execution. You must have the privilege to write to that file to get the timing result.