IPAPI: Designing an Improved Provenance API

Lucian Carata, Ripduman Sohan, Andrew Rice, and Andy Hopper

Computer Laboratory, University of Cambridge
{*firstname.lastname*}@*cam.ac.uk*

Abstract

We investigate the main limitations imposed by existing provenance systems in the development of provenance-aware applications. In the case of disclosed provenance APIs, most of those limitations can be traced back to the inability to integrate provenance from different sources, layers and of different granularities into a coherent view of data production.

We consider possible solutions in the design of an Improved Provenance API (IPAPI), based on a general model of how different system entities interact to generate, accumulate or propagate provenance. The resulting architecture enables a whole new range of provenance capture scenarios, for which available APIs do not provide adequate support.

1 Introduction

Capturing provenance metadata by *observing* the data production process (an approach taken by systems like ES3 [1] or PASS [2]) is limited in terms of recovering the exact semantics of each operation across multiple abstraction levels (application, OS, network). This may lead to the existence of false positives when describing dependencies between different data items: for example, a system that observes provenance at the OS level will link each output of a process to all of its inputs, even if a particular output is actually derived at application level from only a few of the inputs. Consequently, the utility of recorded provenance diminishes as one looks further into the history of a particular data item (false relationships accumulate), and real causal dependencies become harder to distinguish for long derivation chains. In addition, systems that observe provenance require the use of provenance-aware kernels, filesystems or runtime environments across all the nodes involved in the processing of data, which hinders their wide adoption in distributed environments.

It is possible to overcome those drawbacks by requiring applications to explicitly *disclose* provenance. Doing so trades off the transparency of provenance capture for the possibility of recording it in a semantically accurate way, across various layers. Workflow management systems are particularly suitable for this approach, and implementations like Kepler [3] or VisTrails [4] take advantage of the data flow and dependencies disclosed in the workflow's definition to automatically determine provenance.

Extending this to the general case where data can be derived through any process requires making applications provenance-aware: modifying source code to call specialized provenance APIs for disclosing relationships between pieces of data. Such APIs have already been proposed, either as part of observed-provenance systems, as is the case of DPAPI [5], or as general purpose provenance libraries - the case of CPL [6]. We aim to systematically discuss the issues that limit the generality of those APIs, and explore possible solutions in the design of an improved provenance API (IPAPI).

Current API limitations: There are four major scenarios for which current available APIs fail to provide adequate support.

1. Tracking provenance at *granularities smaller than file level*. The existing solutions (both DPAPI and CPL) are able to create arbitrary provenance objects and annotate them with key/value pairs as the process transforms data. However, the main challenge is recovering the identity of those objects starting from output data. It is not currently possible to search for the provenance of some values in a file, as there is no way to determine which provenance objects hold the corresponding information. The same problem appears for operations that do not directly interact with files, but still generate provenance (as is the case with copy-pasting text between two editors).

2. Exploring the *semantics* of disclosed provenance. Existing APIs fail to consider how the key/value annotations can be consumed in automated ways (for example, by applications using provenance to reason about data quality). As long as the meaning of each key/value remains opaque, it is difficult to build applications that use provenance irrespective of its source.

3. Use in a *distributed environment*. One of the common aspects of existing provenance capturing systems is their orientation towards centralized storage. However, it would be useful to store provenance close to the data and easily keep them together when transferring between hosts or responsibility domains. Repositories similar to

the ones used by distributed versioning systems (like git) would be more appropriate in this scenario, replacing huge provenance databases with structures that are more easily synchronized and managed.

4. Leveraging *existing data* as provenance. It is important to recognize that many applications already output information which could be considered provenance (e.g as part of logs or standard IO). Current provenance APIs cannot use this information directly, forcing the developer to disclose it twice when making the application provenance-aware (once as part of normal application output, and subsequently when creating provenance objects). Recognizing existing data as provenance would enable applications to play an active role within a provenance-aware system without having to be modified or recompiled.

Why IPAPI? We develop IPAPI as a base for experimenting with various design solutions of a general purpose disclosed provenance system that is complete and can be used in practice with few constraints, even in heterogeneous environments (considering that not all applications will expose provenance or use IPAPI to do so).

When compared to DPAPI or CPL, the effort required from the application developer is not increased yet a whole range of new use cases become possible. The research question we are tackling when considering those new scenarios asks how disclosed provenance from different sources, layers and of different granularities can be combined to understand the behavior of a data transformation process.

2 Classifying Provenance-aware Entities

Before looking at the core of the API, it is important to get a better understanding of the fundamental entities that have a role in generating, accumulating or propagating provenance metadata. Existing data provenance models (such as PROV-DM, OPM or Provenir) address this at a high-level, making it difficult to clarify the relationships between interacting system entities (processes, files, pipes, etc) and the provenance they produce. Instead, we consider a lower-level model, which allows a direct mapping between system object types and our API data structures.

The provenance structure and properties for different entities will vary depending on their type. For example, the provenance of some entities needs to take into account versioning (e.g. files), but this is not necessary in other cases (processes, pipes) or might be requested on demand (data structures). Existing APIs prefer a uniform approach instead.

As illustrated in Figure 1, a distinction is first made between *active* and *passive* entities, based on a simple criteria: active objects are the ones through which data

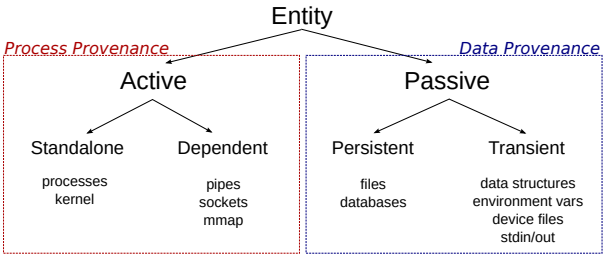

Figure 1: Provenance-aware entity types

"flows", while passive objects just store information. This can be directly mapped on the existing classification of process provenance (active) and data provenance (passive). However, in order to obtain a complete typing of provenance, we need to distinguish subcategories within active and passive entities.

Among **active entities**, the *standalone* ones are computational: they can be instantiated from a passive entity (the binary) to produce, derive or transform data. This includes processes and OS kernels, but could also refer to computational abstractions such as middleware services.

The provenance of such entities is dual: on one hand, it is linked to the provenance of the underlying passive entity, which describes the process used to obtain the executable (the build process, compiler parameters, etc). On the other hand, standalone entities have provenance related to each particular run (command line arguments, environment variables), and are uniquely identifiable during their lifetimes.

In the other subcategory, *dependent* entities are only instantiated within the remit of another active standalone entity. They typically represent communication primitives such as pipes, sockets or memory mappings.

Passive entities map data storage abstractions and are categorized depending on how they change. The *persistent* ones are accessed or modified through fixed system interfaces (read, write) and store data for longer periods of time. *Transient entities* on the other hand have limited lifetimes, and might change without the knowledge of the OS. Typically, they live in volatile memory, even though sometimes they might be presented to the end-user as files.

We are now in the position to give a high level description of how our API considers the accumulation and propagation of provenance: When standalone entities are instantiated, they will map the data from various passive objects (inputs, context) into local data structures (transient entities). They will then proceed to apply transformations, create new data structures, or instantiate other active objects as helpers (for further processing or communication). As dictated by internal control flow, the standalone entity will then map the results back

into passive objects (files, standard output, etc). Provenance needs to track the hierarchy of active objects and the two mappings (input→transient entities and transient entities→output).

3 Design and Implementation

We have designed IPAPI based on the described entity model, in a way which allows it to scale in both directions: from tracking provenance of individual operations to tracking provenance across multiple hosts and enterprise domains.

The API is written in C++ and packaged as a library to which applications link either statically or dynamically. For minimum functionality, the application developer just needs to include the IPAPI header file. At runtime, the library self-initializes, overriding part of the program startup sequence. In the process, a number of provenance objects corresponding to the standalone entity that linked to the library are automatically generated, storing information about the running process, its parent and the active context (command line arguments, environment). This means that even with minimal application changes, we manage to track basic process provenance. As in existing APIs, arbitrary key/value pair annotations can be added to provenance objects when required.

For data provenance, the developer needs to explicitly create those provenance objects representing passive entities, and then disclose the relationships between them and other provenance objects by calling either the `obj_relation` or the `key_relation` member function. The first one discloses actual data flow between two objects (as is the case with basic input-output relationships), while the second enables associations between different key/value pairs (playing the role of a foreign key relationship).

The `key_relation` function also allows for higher levels of provenance abstraction (provenance of provenance). Unlike CPL, we define provenance of provenance as more than just a way to keep track of the context in which provenance was generated. Instead, we consider higher order provenance as a way to explain existing provenance relationships. Take the example of an application that reads the name of its input file from a configuration file. First-order provenance will identify a link between the application and two particular input files. Second order provenance can explain the relationship more abstractly: namely, that the name of the second input depends on a value read from the configuration file.

Provenance Repositories: All resulting metadata is persisted in decentralized provenance repositories, grouping data and its associated provenance as a single manageable unit. The provenance objects which are not directly linked to any persistent entity (like the provenance objects of data passing through a pipe) are stored in the location of the active entity that produced them. Also, provenance from one repository can reference objects from other repositories – a key aspect of being able to scale our system across multiple hosts.

Similar to versioning systems, provenance repositories are managed using a dedicated tool, `prov`. Its purpose is to maintain correct provenance when entities are relocated (moved locally, transferred to other systems, etc).

3.1 Namespacing and Identity

We use a custom naming scheme allowing global provenance object identification across multiple granularities. Each object is part of a hierarchical namespace which is partially controlled by the API, with further levels customizable by application developers. For example, consider the namespace `OS::ping::sockets`. Each level identifies a particular provenance granularity, going from coarse (OS) to fine (sockets). The first two levels are defined by the API when the provenance-aware ping starts execution. The `sockets` level is then defined by the developer to hold the provenance of dependent entities used by the application.

The identity of a new provenance object within a namespace is defined by binding each namespace level except the last to an existing provenance object and then assigning a namespace-unique ID to the new object we are creating. In our example, when disclosing the provenance for a particular socket, we need to first bind the `OS` and `ping` namespace levels to existing objects (representing provenance metadata for the host machine and the instance of the ping application), then give an ID to our socket. The whole process is simplified by the automatic binding of API-managed namespace levels.

Using this scheme, we can start from any provenance object and understand it at different granularities by considering the bindings from each level in its namespace.

3.2 Granularity Control

We have seen how namespaces can be used to demarcate granularity boundaries. However, the sub-file granularity issue presented in the introduction is not completely solved. In order to determine the identity of a provenance object that is linked to certain values in a file, one needs to define correspondences between fragments of a passive object and their provenance.

The `map` function implements this functionality, allowing developers to link provenance objects to specific locations in passive entities. We currently support mapping continuous (possibly overlapping) regions defined by [start position, end position], but we plan generic support based on the explicit declaration of the output data format in the near future.

3.3 Retrieving Provenance Data

The key requirement for integrating multiple sources of provenance is an extensible provenance retrieval API. As part of IPAPI, we define an interface for fetching provenance objects (the `pquery` function), which can use plugins along the provenance retrieval path. Those plugins can be easily developed on a per-application basis and have access to the data returned by scanning the provenance repositories. They can augment or modify this data, parsing output files, connecting to other provenance systems' databases or by mapping from one key/value dictionary to another before returning the requested provenance objects.

Consider the example of a service that evaluates the quality of data based on certain key/value pairs from provenance. In order to integrate a new provenance-aware application with this service, all we have to do is write a plugin that takes existing disclosed provenance key/value pairs from the application and maps them to the format expected by the service. This provides a solution to the variable semantics of data provenance.

4 Use Case

We highlight a typical use case not supported by existing APIs, in which IPAPI manages to keep track of provenance with minimal developer effort. The example is based on four provenance-aware applications: a shell, `provsh`, a "sensor" application (which generates data values), a "filter" application and a "plotter" application. The applications are started from provsh, and communicate through pipes (sensor | filter | plotter).

The sensor application launches a number of threads, and each thread starts outputting data values in the same range. Some of the threads are picked at random to consistently emit values outside the range (simulating defective sensors). Next along the processing chain, the filter application passes all the inputs that meet certain criteria (for example, the ones that are above a certain threshold) to its output. In the plotting application, we look at a graphical display of the values and would like to know from what sensor a particular implausible value came from (we are trying to identify the malfunctioning sensors based on recorded provenance). Even though the applications do not know about each other (they are loosely coupled), their I/O interactions are not based on persistent files (but on pipes), and the requested provenance granularity is that of individual values, the source of the outliers can be successfully identified.

5 Limitations

The limitation of any provenance API is the reliance on the correctness of developer-disclosed information. This is a problem when using provenance for security related tasks, such as intrusion detection: a virus might choose to disclose false provenance to cover its tracks and make it impossible to determine which parts of the system it has affected. We believe that the way forward is to combine disclosed provenance with a low-overhead observed provenance system, and check for provenance consistency between the two. A trust-based model that classifies active provenance entities could also be a viable solution, but would require more user input.

6 Conclusion

We have highlighted the problems and limitations of current provenance APIs, and addressed those problems in the design of IPAPI. We are using IPAPI to understand more about provenance in distributed (heterogeneous) environments, where applications might want to use provenance in computations or automated inferences.

Availability

In order to encourage discussion and use, we have open-sourced IPAPI, under a BSD License. It is available together with its full documentation from `http://github.com/lc525/ipapi`

Acknowledgments

The authors would like to thank George Coulouris, Sherif Akoush and the other members of the FRESCO project for their insights and feedback on the contents of this research.

References

[1] J. Frew, D. Metzger, and P. Slaughter, "Automatic capture and reconstruction of computational provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 485–496, 2008.

[2] K. Muniswamy-Reddy, D. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the 2006 USENIX Annual Technical Conference*, pp. 43–56, 2006.

[3] I. Altintas, O. Barney, and E. Jaeger-frank, "Provenance collection support in the Kepler scientific workflow system," in *In Proceedings of the International Provenance and Annotation Workshop (IPAW*, pp. 118–132, Springer-Verlag, 2006.

[4] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Visualization meets data management," in *In ACM SIGMOD*, pp. 745–747, ACM Press, 2006.

[5] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, (Berkeley, CA, USA), p. 10, USENIX Association, 2009.

[6] P. Macko and M. Seltzer, "A general-purpose provenance library," in *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*, TaPP'12, (Berkeley, CA, USA), p. 6, USENIX Association, 2012.