Layering in Provenance-Aware Storage Systems

Citation	Muniswamy-Reddy, Kiran-Kumar, Joseph Barillari, Uri Braun, David A. Holland, Diana Maclean, Margo Seltzer, and Stephen D. Holland. 2008. Layering in Provenance-Aware Storage Systems. Harvard Computer Science Group Technical Report TR-04-08.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:23597698
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Layering in Provenance-Aware Storage Systems

Kiran-Kumar Muniswamy-Reddy, Joseph Barillari, Uri Braun,
David A. Holland, Diana Maclean, Margo Seltzer, Stephen D. Holland

TR-04-08

Computer Science Group
Harvard University
Cambridge, Massachusetts

Layering in Provenance-Aware Storage Systems

Kiran-Kumar Muniswamy-Reddy, Joseph Barillari†, Uri Braun,
David A. Holland, Diana Maclean, Margo Seltzer
Harvard School of Engineering and Applied Sciences
†and the Harvard-MIT Division of Health Sciences and Technology

Stephen D. Holland
Iowa State University Department of Aerospace Engineering [*]

Abstract

Digital provenance describes the ancestry or history of a digital document. Provenance provides answers to questions such as: "How does the ancestry of these objects differ?" "Are there source code files tainted by proprietary software?" "How was this object created?"

Prior systems used to collect and maintain provenance operate within a single layer of abstraction: the system call boundary, a workflow specification language, or in a domain-specific application level. The provenance collected at each of these layers of abstraction is different, and all of it is important at one time or another.

All of these solutions fundamentally fail to account for the different layers of abstraction at which users need to reason about their data and processes. None of these systems support queries across different layers of abstraction to answer a question such as "The calculated values in my spreadsheet have changed. Is this due to a change in the spreadsheet, a difference in the spreadsheet application, the libraries being used, or the operating system being used?"

We present an architecture for provenance collection that facilitates the integration of provenance across multiple layers of abstraction and across network boundaries. We show how the need to support provenance collection at multiple layers drives the architecture. We present provenance-aware use cases from the field of thermography and quantify system overheads, showing that we can provide new functionality with acceptable overhead.

1 Introduction

In digital systems, *provenance* is the record of the creation and modification of an object. Provenance-aware database systems such as Trio [23] record provenance at tuple-granularity in a database. Service-oriented architecture (SOA) approaches [8, 9, 17] track provenance of user- or application-defined objects. System-call-based systems such as ES3 [3] and PASS [16] track provenance of files. In all of these cases, provenance increases the value of the data it describes.

The fundamental difference between these systems is the layer of abstraction at which each identifies and records provenance. Application level systems, such as Trio, record provenance at the semantic level of the application – tuples for a database system. Other application-level solutions record provenance at the level of business objects, lines of a source file, or other units that have semantic meaning to the application. SOA approaches are typically associated with workflow engines and record provenance at the level of workflow stages and data or message exchanges. System-call-based systems record provenance at the level communicated via system calls – processes and files.

While each level of abstraction is useful in its own right, integration across these layers is crucial and currently absent. For example, consider the following scenario: we obtain proprietary files from a third party, analyze those files with Python programs, and report results. The proprietary files have names that reveal confidential information, so one member of our lab copies the real files into generic data files with meaningless names. Any member of the team can then use the anonymized data files and perform analyses. When we report results we must map back the original file names. Historically, this is done manually via notes scribbled down during the initial anonymization. In a provenance-aware world, the system takes these notes automatically.

A solution that collects provenance only in Python (or in a Python application) does not address the confidentiality management problem. A solution at the system call layer cannot capture the details of the Python analyses. Solving the combined problem requires an ap-

[*] This material is based on work supported by the Federal Aviation Administration under Contract #DTFA03-98-D-00008, Delivery Order #0037 and performed at Iowa State University's Center for Non-Destructive Evaluation as part of the Engine Titanium Consortium Phase III Thermal Acoustic Studies program.

proach that integrates the information in a provenance-aware Python and a provenance-aware operating system, so that dependencies can be tracked across both layers.

We describe the PASSv2 system that enables this seamless integration. It is a system-call-level provenance system that natively tracks the provenance of files. PASSv2's layered architecture enables integration with an application-level Python-library we developed.

The contributions of this work are:

- An architecture for provenance-aware systems that integrates provenance across multiple layers of abstraction.
- A provenance-aware Python library and a general framework for provenance-aware applications.
- A provenance-aware NFS implementation.
- A query language for asking provenance questions.
- A prototype system encompassing provenance-aware applications, a provenance-aware kernel, and provenance-aware network attached storage.
- Use cases demonstrating the integration between provenance-aware applications and a provenance-aware system.

The rest of this paper is organized as follows. In Section 2 we make the case for provenance collection as a key infrastructure component and then discuss the unmet challenges it introduces in Section 3. In Section 4 we present our layered architecture, addressing issues such as functionality, recovery, and security. Section 5 introduces the challenges in querying provenance In Section 6 we discuss our experience making applications and NFS provenance aware. In Section 7 we describe several provenance use cases, demonstrating how they are addressed by our system. Section 8 presents the cost of providing these features in terms of time and space overheads, and we conclude in Section 9.

2 The Case for Provenance

Provenance systems are a crucial infrastructure component in areas like scientific computation, regulatory compliance, intellectual property management, system auditing, and data archival.

Provenance potentially eases the burden of regulatory compliance faced by many companies. Regulatory compliance has two main components: *supervision* and *discovery* [7], both of which are easily and naturally expressed in terms of provenance. For example, in order to comply with the United States Securities Exchange (SEC) rule 10b-5[20] (which prohibits insider trading), companies are required both to supervise all electronic communications to detect and prevent sensitive information disclosure and also to maintain records of all electronic communications to show compliance and respond to discovery in litigation [7]. In both cases, data ancestry plays a significant role, and provenance offers a natural solution for the problem domain. Supervision, for example, might stipulate that communications out of the finance office that derive from corporate earnings data should not be transmitted to anyone or any system that directly interfaces to the team that manages 401(K) investments. Discovery, on the other hand, might require finding all communications between employees E_1 and E_2, and client C, that contain the words/phrases "earnings", "trades" and "stock prices" – a combination of data query and a provenance query. Current solutions tend to be highly domain-specific (for example, particular to stock trading); the advantage of a provenance-based approach is that we can address larger, more abstract notions of regulatory compliance.

Another domain for which provenance is well suited is verifying intellectual property compliance. For example, companies that use and develop both proprietary software and open source software routinely require pre-release checks to make sure the proprietary software has not been tainted by open source software and vice-versa. In most cases, this is a tedious, manual process. A system that tracks provenance can look for paths in the ancestry tree between source files with different licensing models.

Provenance is also related to data security. The traditional concept of classified information is readily expressed in terms of provenance: no unclassified data shall descend from classified data except via a legitimate declassification. Furthermore, system auditing and provenance collection are closely related; in both cases one records actions taken and flows of information from file to file. One may be able to detect intrusions by looking for unusual patterns in provenance, similar to the technique based on unusual patterns of system calls [21]. Provenance is also useful for forensics after an intrusion has been detected [14].

3 Background

Previous work exists at each of the layers of abstraction discussed in the introduction, but there is no approach that integrates across the different levels providing a unified solution.

At the domain-specific level, systems like GenePattern [10] and our Python wrappers (described below) provide provenance for environments in which scientists perform routine analyses. Experiments done within the analysis environment retain provenance, but the chain of creation is often broken as data moves into or out of the environment. One of the most common provenance use cases in this domain is the ability to extract the sequence of processing steps that, after days or weeks of interactive experimentation, produced an important result.

Tracking provenance at the level provided by workflow engines – such as Pasoa [17], myGrid [24] and Chimera [9] – enables users to group related collections into single logical entities. For example, scientists frequently refer to logical data sets containing hundreds or thousands of individual files. These systems can answer queries such as "What were all the output files of a particular experiment?" or "What version of the software release are we using for this analysis?" These systems lose some of the semantic knowledge available at the domain-specific level, but do provide the ability to form larger, virtual objects such as collections.

System level solutions like ES3 [3] and PASS [16] capture information at the system level, losing both the semantic information of domain-specific solutions and also the relationships among data sets and processing units found in workflow engines. However, these systems provide a wealth of information about the environment in which objects are created, such as the specific binaries, libraries, and kernel modules in use.

All of these solutions fundamentally fail to account for the different layers of abstraction at which users need to reason about their data and processes. Simply taking the union of provenance at different layers of abstraction is insufficient, because the relationships between the objects appearing at different layers are vital.

There are two basic aspects to provenance: attributes and relationships. Attributes state facts about an object, such as its name or creation time. Relationships indicating things like data flow, process nesting, and versioning, represent the ancestry and are more interesting.

These relationships form a directed acyclic graph (DAG). (Why a DAG? Ancestry relationships must be acyclic; cyclic ancestry violates causality. They are directed; ancestor-of is not the same as descendent-of. Ancestry forms a graph, not a tree: if one process writes two files and another reads them, a "diamond" is formed.)

A naive approach to unifying provenance at differing layers of abstraction is to take the union of the DAGs. This overlays the DAGs without forming any connections between them. At a minimum, objects at each layer of abstraction must be mapped to objects at the surrounding layers so that the DAGs can be merged, allowing queries to cross the layers of abstraction. Managing these mappings of identities automatically requires the layers to share meaningful identity information. This observation is fundamental to handling provenance at multiple levels of abstraction and drives the layering architecture described in the next section.

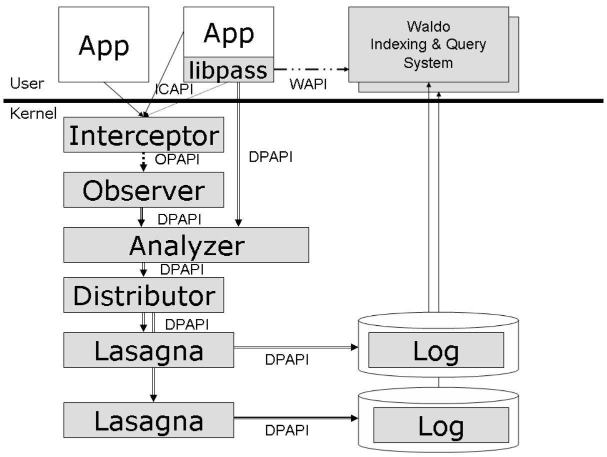

Figure 1: PASSv2 Architecture

4 Architecture

We present the PASSv2 architecture in six parts. We begin with a high level overview that introduces the main components of the PASS system. Following this, we detail five aspects of the PASS system that address the fundamental issues that we have encountered in provenance collection, namely: cycles (Section 4.2), layering (Section 4.3), persistence (Section 4.4), recovery (Section 4.5), and security (Section 4.6).

4.1 Overview

From a user perspective, PASSv2 is an operating system that collects provenance invisibly. Users interact with it as they would with any Linux system, except that after running unmodified applications, they can then run queries against the provenance via a GUI or command-line query tool. Figure 2, for example, shows the ancestry graph of a workload in brain imaging [18].

PASSv2 also incorporates provenance collected by provenance-aware applications. Application developers can develop PASS provenance-aware applications by augmenting their code to make it provenance aware and then linking their applications against `libpass`.

We present the PASSv2 architecture in four functional units: control, generation, analysis and recording, and index and query.

4.1.1 Control

The control component of PASSv2 interfaces between the application and the system. Provenance-aware applications disclose their own provenance to the system and will sometimes disable provenance collection for some objects. The control component allows this.

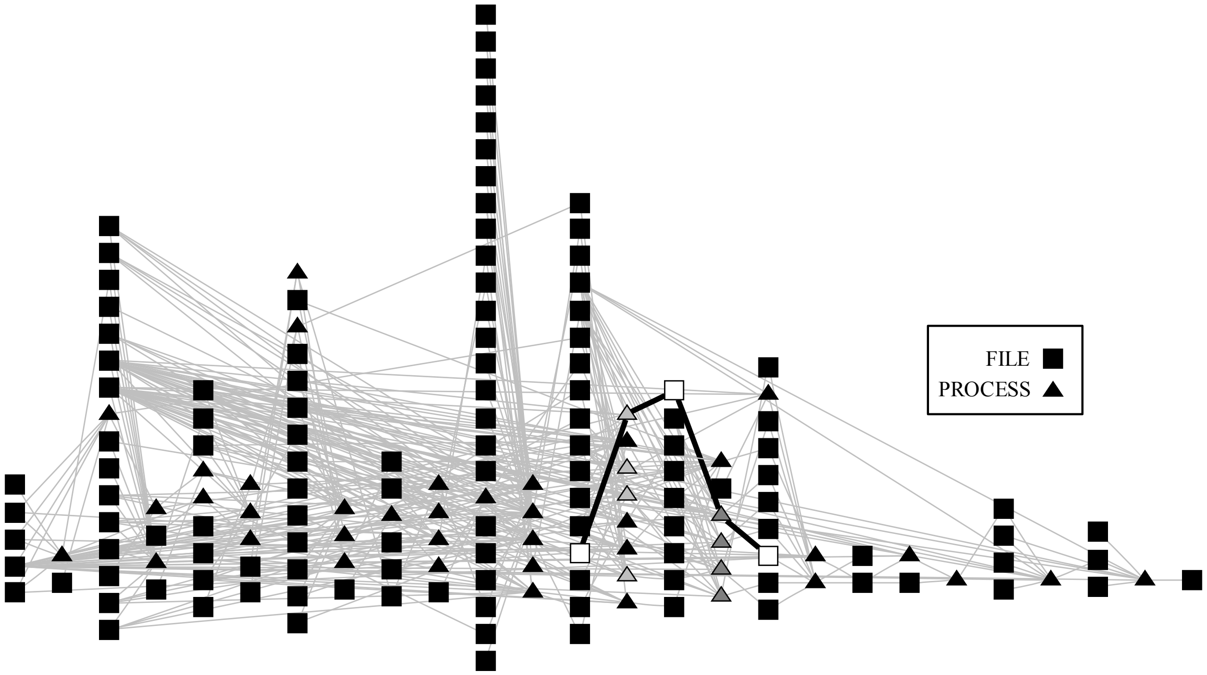

Figure 2: This figure shows the ancestors of the output file `atlas-x.jpg` from a brain imaging workload. The graph comprises 167 nodes and 379 edges. The light gray triangles are instances of the `alignuswarp` process; the dark gray triangles are instances of the `reslice` process. The heavy black line highlights one particular instance of a common processing pattern.

Control is implemented via a library, `libpass`, against which applications can link. This allows them to transmit provenance to PASS, disable provenance collection, and set security attributes on the collected provenance.

An application that does not link against libpass still has provenance collected for it; it simply is not provenance-aware as far as PASS is concerned. Thus, it provides no application provenance to the system and has no control over what observed provenance is or is not collected.

4.1.2 Generation

Provenance generation takes place as applications run, and consists of *identifying* relevant events and *translating* these events (system calls) into provenance information. For example, a `read` system call is a relevant event that must be translated into provenance stating that the process issuing the system call depends upon the file it read. Similarly, a `write` system call is a relevant event that translates to the converse relationship. When the same process does both of these, we implicitly establish that the second file depends transitively on the first.

The two components responsible for generation are the *interceptor*, which identifies relevant events, and the *observer*, which translates the identified events into provenance. These two components are distinguished primarily for implementation reasons: the observer is a self-contained piece of code, but the interceptor is a collection of small patches widely dispersed about the system call layer of the kernel. Enforcing a clear separation between them improves the robustness of the code. Also, we believe the distinction may be meaningful or helpful in the general case of provenance collection.

4.1.3 Analysis & Recording

PASSv2 gathers provenance explicitly from higher-level, provenance-aware applications and implicitly from the observer. These two streams come together at the *analyzer*, which eliminates cycles and redundant provenance. The analyzer was the most troublesome component in the original PASS prototype, and our experience with it led to several significant changes. First, instead of copying a process's provenance into each object it creates, we materialize processes in the storage system as first class objects with provenance. Second, we changed to a simpler cycle-breaking algorithm based on timestamp ordering, described in more detail in Section 4.2.

Third, we realized that analyzers had to stack to provide the layering we needed. Section 4.3 discusses layering in more detail.

Since processes are first-class objects, the system must track and store process provenance. However, processes are not by themselves persistent objects residing on a PASS-enabled volume, introducing the question of where their provenance should be stored. Similar issues arise with pipe objects, files from non-PASS volumes, and objects introduced by provenance-aware applications. In these cases, PASSv2 must select some PASS-enabled volume on which to record their provenance. We introduce the *distributor* to address this issue.

The distributor caches provenance records for all objects that are not PASS files. When those objects become part of the ancestry of a persistent object on a PASS-enabled volume, the distributor assigns these objects to that PASS volume and arranges for their provenance to be written.

The last piece of the analysis and recording component is making provenance persistent in the file system. The fundamental issue is consistency: what does it mean for on-disk provenance to be "consistent", and what guarantees does the file system provide?

Our definition of consistency is that all data on disk is described by provenance, and the provenance on disk accurately reflects the data on disk. Ensuring this requires that provenance and data flow together through the system and are identifiable as being inextricably coupled. For this reason we store provenance inside the file system; we can thus provide an integrated interface that maintains the coupling.

In PASSv2 we have a stacked file system called Lasagna[1]. Lasagna's stacking infrastructure is is derived from `ecryptfs` [11].

The first PASS prototype wrote provenance directly into databases that provided indexed access to provenance. This arrangement was neither efficient nor scalable, so PASSv2 writes all provenance records to a log that is later moved to a database and indexed by a user-level daemon process. (This log and its database are hidden within the stacked file system.)

The log is implemented as a collection of files, named `log.`*nnnnnnnnnn*, where the n's represent decimal digits. The kernel appends entries to the current log file, until it exceeds a parameterized maximum size or has been dormant for a parameterized length of time, at which point the the kernel closes that file and creates a new one with the next file number. The user-level daemon Waldo (described in the next section) uses the Linux `inotify` interface to monitor this activity, so when a log file is complete it can wake up, process it into the database, and remove it.

To ensure that the on-disk provenance accurately reflects the on-disk data, we use a write-ahead-provenance (WAP) protocol. WAP is analogous to write-ahead-logging from the database literature [13]. It means that all provenance records must be written to disk before the data they describe. This eliminates the possibility that unprovenanced data can get to disk. However, it is possible for the system to crash when provenance has been written but its corresponding data has not. This case is detected and addressed during file system recovery, discussed in Section 4.5.

4.1.4 Index and Query

Provenance queries come in numerous forms, and indices are essential for acceptable performance [4]. As we learned with the first PASS prototype, inline indexing degrades application performance by introducing long latencies. The PASSv2 log-based scheme decouples writing provenance records from entering those records into a database and indexing them. This allows indexing to go on in the background without forcing applications to wait, so we can maintain a rich set of indexes.

Our user level daemon, Waldo, reads provenance records from the log and stores the provenance into a Berkeley DB database as named attributes: name, inode, `argv`, environment, etc. On top of this base table, Waldo creates several indices used for query optimization. Chief among these is the descendent index: in the primary database we keep only ancestors, because this avoids having to update a widely scattered set of ancestor objects every time a descendent is created. The descendent index maintains reverse mappings. We also maintain indexes of object names and the contents of `argv` and environment strings.

In addition to writing the provenance database and indexes, Waldo is responsible for accessing the database on behalf of the query engine, `sage`. There is one Waldo for each PASS volume; there is one `sage` for each machine, which talks to the the various Waldo processes as necessary. It is also responsible for enforcing the security model discussed in Section 4.6.

4.2 Cycles

It seems contradictory that provenance could have cycles – after all, provenance represents ancestry and it is physically impossible to be one's own ancestor. So, how can cycles arise?

Cycles arise because the system is trying to avoid generating a large number of versions. Consider a process P that reads a file A and then writes A. Since A is an input

[1] The file system in our earlier prototype was called PASTA, Provenance And Stacking. Since this file system includes support for layering, Lasagna seemed an obvious name.

to P, P depends on A. However, since P writes A, A also depends on P. Thus, we have constructed a cycle.

The obvious solution is to assume the write to A creates a new version of A, eliminating the cycle. In fact, if every write creates a new version, there can never be cycles. Unfortunately, creating a new version on every write is unacceptably expensive. Not doing so, however, opens up the possibility of cycles. The fundamental question is when it is necessary to declare a new version; the goal is to do so, other than at user request, only when necessary to avoid creating a cycle.

PASSv2 uses a time-stamp based algorithm, leveraging work in the database literature. The main motivation for this approach is that it is simpler and adapts more naturally to an environment where analyzers stack in layers (see Section 4.3). The algorithm is as follows:

- Create a new version of an object *before* adding a cross-reference adding either a new ancestor or a new version of an existing ancestor.

The above rule is sufficient to avoid cycles.

4.3 Layering

Support for layering is critical for three reasons. First, a system that supports provenance-aware applications captures semantic data at multiple layers in the system. Second, our experience with the Second Provenance Challenge [19] revealed that object naming is fundamental for provenance interoperability, and layering is essential to relate semantically equivalent objects that have different names in different layers. Third, we wanted to support provenance-aware storage systems attached as network storage.

The fundamental issue in integrating application provenance with system provenance is resolving names between the layers. Applications must be able to map application objects to system objects or create handles to application objects to which the system can refer. This requires fairly simple handle creation and management support in our API between different provenance levels. Section 4.4 describes these APIs.

Supporting network-attached storage proved even more challenging. An analyzer must process all the provenance records at its abstraction layer in order to properly avoid cycles. Now, consider this in the context of a NAS setup where a single process on a client accesses data from two different storage servers. The analyzer must reside at the client, because that is the only place where all related provenance records are processed.

Next consider two programs running on different clients accessing the same server. By the same logic, the analyzer must reside on the server, because that is the only place where all related provenance records are processed.

Finally, combine these two scenarios: two client programs each accessing files from two different file servers. In this case, we need analyzers on both the clients and the servers.

This means that in general we must have an analyzer on every client and also an analyzer on every server; this in turn means that the client instance of the analyzer must be able to stack on top of the server instance, which means that the input and output data representations must be the same.

4.4 The Provenance File System, Lasagna

Lasagna extends the VFS interface with three new inode operations and one new superblock operation. The inode operations `pass_read` and `pass_write` map to standard `read` and `write` calls but are provenance-aware. The `pass_write` call takes both a data buffer and a collection ("bundle") of provenance records that describe that buffer; `pass_read` returns both the data requested and the unique identifier for the version of the file that was read. In a layered system, cycle-breaking at one layer might require creation of a new version in lower layers. Lasagna supports the `pass_freeze` inode operation to provide this versioning support. (Freeze "finishes" a version of an object; after being frozen, new activity "thaws" the object and increments the version number.)

A user-level provenance system may need to create new objects that do not map to a particular file system object, so Lasagna provides a superblock operation `make_object` that creates such objects. Such objects can also call `pass_write` to store provenance.

4.5 Recovery

As discussed in Section 4.1.3, our goal is that every piece of data on a PASS is accurately described by provenance records, and any provenance record reflects a current or past state of the data. WAP guarantees that all data is described by provenance, and recovery is the process guaranteeing that all provenance accurately describes data.

The goal of recovery is to handle the case where provenance records were written to disk, but the data they describe were not. There are two ways that this might happen. First, the disk write of the data can fail after the provenance records are on disk. Second, the system can crash.

In the first case, the system is still running and can take appropriate action. For each independent data write to disk, we generate a provenance WAP record that records the file to which the data belongs, the page to which the write is being applied, and an MD5 checksum of the page. All the WAP records corresponding to a single write system call are bracketed between BEGIN and END records to provide logical grouping. WAP records are forced to the log before the system issues the corresponding data writes. If the data write fails, we write a CANCEL record to the log. The CANCEL indicates to the recovery process that the write failed and that the provenance associated with the write should not be placed in the provenance database. If only a part of the data is written, i.e., short write, we re-issue WAP records that record the MD5 checksum of the page with just the successfully written data on it.

In the second case (system crash), recovery runs before the PASS volume is mounted. We first process all outstanding log files except the last one; the kernel handles each new log file as a checkpoint, so all data writes with provenance in the completed log files are present on disk. This processing is identical to normal Waldo processing except we check for duplicates and drop them.

For the final file, we break it up into transactions identified by BEGIN and END pairs. Any uncommitted transaction (no END record) is ignored. Next, we go through the transactions in reverse order. For each transaction, we iterate over the WAP records in that transaction. We then validate the checksum in each WAP record against the bytes in the file to which it refers. If the WAP record validates, we record that the designated page is now validated, and ignore any WAP records we later encounter for this page. If one write went to disk properly, then all previous writes to the same page must have also gone to disk, because we guarantee that writes are logged in the same order they are applied to the file system buffer cache.

If the data does not validate, we take all provenance records in the current transaction and write them to the provenance database, tagged with a MISMATCH flag. We do not generate regular indexes for these records, but instead enter them into a special mismatch index, so that we can generate reports about possible data corruption.

After recovery, we remove all log files and start the file system.

4.6 Security

We explicitly chose not to tackle the security problem in our first prototype, because security on provenance was not a well understood question, let alone a question with commonly accepted answers. The fundamental provenance security problem is that provenance and the data it describes do not necessarily share the same access control. There is no universally correct rule that dictates which of the two (data or provenance) requires stronger control. For example, consider a paper review – the data (contents of the review) must clearly be accessible to the authors, but the provenance (reviewers' identities) must not be accessible. In this case, the provenance must be more tightly controlled than the data. Now consider the documents produced by a government panel. The document produced (the data) may be marked as classified, but the membership of the committee and the identities of all participants in briefings (the provenance) might be entirely public. In this case, the data carries stronger access control than the provenance.

Securing provenance requires security for both provenance attributes (the names of applications, the people/uids that ran those programs) and the ancestry graph (the documents from which this document was derived and the ancestors of those documents). Securing attributes is straightforward and standard methods of specifying access controls are appropriate; we use the access control method provided by the host system. Securing ancestry is more complicated.

We use a graph-based access control model for securing ancestry data. Permissions for each ancestry relationship are set on a per-user basis. The possible permissions for an ancestry relationship are: *none* – the user is not permitted to know anything about the given relationship, including whether it exists; *exists* – the user is allowed to know the relationship exists but nothing about the ancestor/descendent on the other side of the relationship; *traversal* – the user has full knowledge of the ancestry relationship. Applications can use `libpass` to set access control on provenance attributes and/or ancestry relationships. During queries, the `sage` query engine enforces the access controls.

We are currently working on a formalization and accompanying proofs for this security model.

5 Querying

We stated earlier that provenance forms a directed acyclic graph (DAG). In this section, we discuss the implications and how they drove us to our current data model and our new query language.

We undertook a study where we shadowed computational science users to understand what types of queries they might ask a provenance system. We found that they were frequently interested in identifying files that were *friends* – files that had been through *similar* processing steps. Such a sequence of processing steps is a particular path through the ancestry graph, and users frequently wish to express shorthands for commonly used paths. Conversely, having previously identified a pair of objects believed to be friends, one might wish to compare the processing sequences that generated them. These sorts of queries are easily visualized. Figure 3 shows a subset of the workflow from Figure 2). The grey nodes repre-

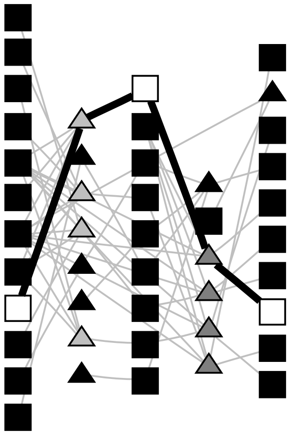

Figure 3: Cropped view of Figure 2, indicating a single path from the input file `anatomy4.img` (furthest-left white square), to the `align_warp` process, its output file `warp4.warp` (middle white square), the `reslice` process, and its output `resliced4.img` (rightmost right square). The grey nodes represent the processing steps on paths of friends of `resliced4.img`.

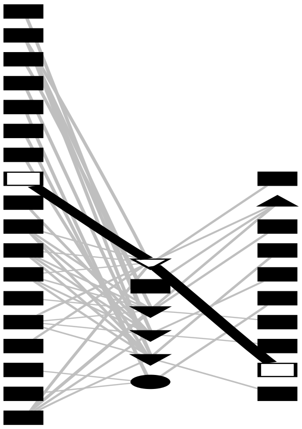

Figure 4: Collapsed version of Figure 3. All the friend paths, described as `align_warp`→ ∗ →`reslice` have been replaced by a single inverted triangle. (colored white) representing a higher-level processing step.

sent paths that lead to friends of `atlas-x.jpg`. Figure 4 shows the same subset with the friend workflows collapsed to a single processing step corresponding to the two-step path in Figure 3. After struggling through three generations of query languages for PASS and considering what we observed about our users, we composed the following list of requirements for a provenance query language:

- Paths through graphs are a central feature.
- Paths should be first class objects.
- The language should support regular expression matching on paths.
- The language should support aggregation operators (count, sum, etc.)
- The language should support sub-queries.

Most existing provenance systems use either an XML-based or relational representation. We found both lacking. XML has a notion of paths (XPath) but is inherently tree-structured and does not extend well to graphs. SQL has no native concept of paths; writing path-like queries in SQL requires mentally translating the paths into recursive queries, which are themselves expensive and unnatural to a relational environment. It seemed most appropriate to find a query language that was designed specifically for querying graphs.

The Lore semistructured database project at Stanford provided us with the Lorel [1] query language and its "OEM" data model. A semistructured database is one with no fixed schema; the data model in Lore is that of a collection of arbitrary objects, some holding values and some holding tables of named linkages to other objects. The data types of values and linkages are not fixed, and the query language is designed accordingly.

The OEM data model is appealing for provenance, because it naturally represents both graphs and object attributes, and Lorel provides the path-oriented query model for which we were looking. Unfortunately, we found that Lorel had several shortcomings. In particular, it did not support boolean values in the database, its formal grammar was ambiguous, and there were corner cases where the semantics were not well defined.

We also needed to extend Lorel to allow traversal of graph edges in both directions. While the edges in provenance graphs are directed (ancestor-of is not the same as descendent-of) it is clearly necessary to be able to traverse edges in either direction: "find all descendents" and "find all ancestors" are both valid and useful queries.

We present a more in-depth discussion of these issues in an upcoming publication [12].

We have developed a new query language based on

Lorel, which we call Path Query Language (PQL or "pickle"). It is specifically geared to handle our requirements for querying provenance.

Lorel is derived from OQL [5], the object query language from the O_2 database system [6], which is in turn derived from SQL. This heritage is evident in the general syntax and language organization. In all these languages the basic unit is the select-from-where construction, and PQL is no exception:

```
select atlas.input.name
from Provenance.obj as atlas
where atlas.name = "atlas-x.gif"
  and atlas.type = "file"
```

The from-clause names collections that are implicitly iterated; each combination of values is tested against the where-clause, and those that match are used in the select-clause to construct results. The collections to be iterated are sets of objects found by following paths.

PQL's conception of a path is a fully general regular expression over graph edges. This extends Lorel, where regular expressions only allowed repetition of single steps. Lorel's paths are in turn a generalization of the structure member references in OQL.

The example query above searches all the objects reachable from a single `obj` edge from the global object `Provenance` that have the given type and file name, and for each returns the names of all immediate inputs – processes that wrote to the file. This is a simple query that could be handled easily by almost any query system; the real power of PQL comes into play when more complex searches are desired. For example:

```
select result.name
from Provenance.obj{atlas}
   (.input{intermed})+ as result
where atlas.name = "atlas-x.gif"
  and atlas.type = "file"
  and intermed.name <> "softmean"
  and intermed.type <> "proc"
```

This is the second standard query from the Provenance Challenge [15]. It returns the names of all ancestors of the file `atlas-x.gif`, omitting everything reachable only through a process named `softmean`.

In PQL (as in Lorel) variables can be bound to the objects found at intermediate places in a path. The variable `atlas` is bound to the first object found starting from `Provenance` (as in the earlier example); then, under the regular expression operator +, the variable `intermed` is bound in turn to every object found after an `input` edge. Thus, in addition to fixing the starting point of the repetition, the where-clause excludes any intermediate step that is a `softmean` process.

Variables can also be bound to a path directly; the value of such a "path variable" is the precise sequence of nodes and edges found while traversing the path. Such values can be compared or returned as query results; this allows inspecting the structure of the database.

6 Provenance-Enabled Systems

We have been developing provenance-aware prototypes for a collection of tools including the Z-shell (zsh) the statistical environment R, Python, and NFS. In this section, we discuss how we made Python and NFS provenance-aware and the more-general approach we developed to make applications provenance-aware.

6.1 Provenance-Aware Python

The applications for which we wanted to collect provenance are all written in Python and manipulate large data files containing semantically meaningful *chunks* inside them.

We needed support in Python to track provenance at a useful level while retaining the ability to track provenance between our analysis programs and regular shell commands.

We developed a Python wrapper class that makes Python programs provenance aware. The wrapper exposes application objects, such as Python objects and methods, to the kernel. We associate objects and method invocations with *provenance descriptors* and automatically record their relationships to one another and to system-level objects. On each method invocation we record the parameters as inputs to the method invocation and record the relationship between the method invocation and its return value. We provide this with a Python provenance library and Python bindings for `libpass`.

Our Python wrapper connects to the provenance libraries using Python's `ctypes` foreign-function interface module. The python wrapper has two methods: `wrap()` and `wrapmod()`. `wrap()` takes a Python object and encloses it in a wrapper that creates provenance descriptor records for each method call, each call's arguments, and each call's results. Suppose that `obj` is an object and `obj.func` is a method on it that takes a single argument `x` and returns a single value `ret`. Calling `wrap(obj)` produces `wobj`, a provenance-aware wrapper around `obj` that intercepts method calls and produces provenance records.

Calling `wobj.func(x)` causes this wrapper to generate the following items:

- A provenance descriptor for the function call instance (`call_pd`).

- A cross-reference from call_pd to the container object a.
- A cross-reference from x to the call instance (call_pd).
- A cross-reference from call_pd to the returned object ret.

The wrapmod() function works similarly, except that it wraps Python module instances instead of Python object instances.

The user can control whether provenance information is recorded by selecting which objects to bind to file descriptors on provenance-tracked partitions. For instance, a user who wanted to record the provenance of ret through its creation by a.func and dependency on a.func's argument x would create a cross-reference from ret to a file descriptor on a PASS volume.

While tracking provenance through wrapper classes implemented directly in the target language has the advantage of simplicity, it also has some limitations. It is not possible for the wrapping process to be entirely transparent; since the wrapper is implemented in that language, it is inherently accessible to that language, whereas an interpreter that tracks provenance itself could completely hide that tracking from the target language. In Python, we found that some library routines used unnecessarily stringent type-checking that got caught up on our wrapper. Depending on the language, there may also be important operations that are difficult or impossible to wrap, such as constructors for builtin primitives. Nevertheless, our provenance system with its Python wrapper (and a few workarounds for issues such as the above) is able to evaluate the provenance of output files from Python scripts through method and function calls back to the origin of the data.

6.2 The General Case

Our experience with Python, zsh, and R led to a simple set of guidelines for making applications provenance-aware.

While PASS considers processes and files applications consider functions or methods and objects or variables. In the same way that our kernel PASS implementation maintains provenance records as files are accessed and processes execute, applications maintain provenance records on variables and the functions that operate upon them. Applications use the make_object call to create provenance descriptors (handles) for their variables and functions. They also obtain handles when reading data from the underlying PASS by replacing normal read calls with passusread calls. Finally, when they output persistent data, they replace write calls with passuswrite calls, passing the provenance they have collected to the kernel at this point.

The relative ease with which application needs matched our own internal system needs convinced us that our APIs are correct. The Python system was originally implemented by someone outside the PASS project. We obtained the library and integrated it with PASS in approximately 24 hours.

6.3 Provenance-Aware NFS

While provenance-aware applications stretch our design upwards, supporting network-attached storage stretched it downwards through the system stack. We selected NFSv4 as the basis for our provenance-aware NAS layer. We implemented provenance-aware NFS (PANFS[2]) on Linux 2.6.23.17 as it had a stable NFSv4 server implementation.

6.3.1 NFSv4 Background

We designed PANFS to fit in naturally with NFSv4. In this section, we review those features of NFSv4 most relevant to our implementation.

Client Side Caching NFSv4 supports *close-to-open* consistency caching. In this model, the NFS client caches data and its attributes on close. If an application at the client re-opens the file within a fixed period, the NFS client queries the server for the latest attributes. If the attributes returned by the server match those on the client, the cached data is valid and used as is; otherwise it is flushed. This mode provides sufficient consistency for most NFS applications. However, it does not provide complete consistency and can lead to problems, as we will see later in the section.

Delegations When a file is being accessed by a single client, the server delegates the file to the client. The client can perform all operations locally and the server prevents other clients from performing conflicting operations. If another client accesses the file, the server revokes the delegation and the client reverts to using close-to-open consistency. Delegations are different from leases/locks in that they are driven from the server side.

Mandatory Locking NFSv4 provides mandatory locking that allows clients to serialize access to a file. Previous versions used NLM, which was only advisory.

[2] One serves Lasagna in a PANFS.

6.3.2 Implementing PASS Operations in NFS

Section 4.1.3 discussed how we extended the inode and superblock operations to support local provenance-aware storage. Here, we discuss how these extensions affect an NFS implementation.

pass_write pass_write requires that we transmit provenance with data in a manner that enables the server to enforce WAP. We create an NFS operation analogous to the local pass_write, called OP_PASSWRITE, that transmits both data and provenance to the server. As long as the data is smaller than the NFSv4 client's block size (typically 64 KB in NFSv4), this approach is sufficient.

Unfortunately, not all data and provenance packets fit this limit. In these cases, we use NFS transactions to encapsulate a collection of operations that must be handled atomically by the server. First, we invoke an OP_BEGINTXN operation, that obtains a transaction ID from the pass volume that the server is exporting. Then, we send the provenance to the server in 64 KB chunks, using a series of OP_PASSPROV operations. Finally, we transmit the data along with a single ENDTXN provenance record using the OP_PASSWRITE operation. The ENDTXN indicates the end of the transaction to the server.

We considered an alternate implementation that obtains a mandatory lock on the file, writes the provenance, and then writes the data as a separate operation. This approach would have provided the coupling between provenance and data that we need for WAP; however, it does not allow us to recover from a client crash. If the client wrote the provenance, crashed before sending the data, and then came back up, there is no way for the server to determine that the provenance must be discarded. Our implementation solves this problem, because the transaction ID enables the server's Waldo to identify the orphaned provenance.

pass_read The NFS pass_read uses the OP_PASSREAD operation, which is quite similar to the normal NFS read. In addition to returning data, OP_PASSREAD also returns additional information necessary to construct a reference to the provenance of the object.

pass_freeze We implement the pass_freeze operation as a provenance record type in OPPASSWRITE. When the analyzer at the client issues a pass_freeze, the client increments the version and attaches a freeze record to the file. Later, when the client sends the file's provenance to the server with an OP_PASSWRITE, the server's Lasagna processes the freeze records, increasing the version number accordingly.

make_object Recall that the make_object is used by applications to create new objects that do not map to particular file system objects We added a new operation called OP_MKOBJECT that performs this function. We could have implemented this operation by creating a file handle at the server and returning it to the client. The client would then use the handle to write provenance. However, the problem with this approach is that it is hard to recover from either a server or client crash. Instead, the operation just returns a pnode number (object ID) to the client and the client constructs a surrogate inode and exports that to the userland as a file. To create the provenance descriptor, the user has to provide a handle to an object on the PASS volume. We bind the provenance descriptor to this PASS object and perform passuswrite calls to the provenance descriptor using that object's pass_write operation. The advantage of this implementation is that it leaves no state on the server. If the server crashes and comes back up, the client can continue to use the provenance descriptor as the PASS object can be rebuilt from the PASS object's file handle. Similarly, if the client crashes, the server does not have to clean up any state as it has only allocated a number to the client.

6.3.3 Network issues

Version Branching Due to the close-to-open consistency model, two different clients can open the same version of a file and concurrently make modifications to it. As we cannot fix the (in)consistency issues inherent in close-to-open systems, our goal is simply to provide the same guarantees that we do locally, that the on-disk provenance and data match. Our approach of versioning at the client and later versioning at the server can lead to *version branching*, where. two clients create independent copies of an object with the same version. We intend to explore allocating versions to clients in chunks to avoid this problem, but have not yet done so. Given the overall lack of precise consistency semantics in NFS, we do not expect our current solution to be problematic for existing applications.

Conflicts due to Partitions A client with a file delegation can make modifications to that file and then become partitioned before it flushes its provenance and data to the server. Another client could then modify the file and flush it to the server. If the original client returns, NFSv4 has mechanisms to detect that there is inconsistency and suggests that clients write conflicting data to a new file.

We leverage NFSv4's solution and attach provenance to the newly created file.

Operation ordering Operations issued by a client may arrive at the server in an order different than that issued by the client. `pass_read` and `make_object` are independent of the order in which they are executed, so they pose no problems. However, `pass_freeze` and `pass_write` are order-sensitive, particularly since `pass_freeze` is issued to break cycles in the records that are about to be written. An out of order arrival might result in a failure to break cycles. We avoid this problem by coupling `pass_freeze` with `pass_write` by making `pass_freeze` a record type for NFS `pass_write`.

Operation Idempotency NFS operations are supposed to be idempotent, but only `OP_PASSREAD` is idempotent. (That is, replaying it does not alter the state of the system.) The `OP_PASSWRITE` and `OP_MKOBJECT` are not. A replayed `OP_PASSWRITE` operation can result in provenance being attached to the wrong version of a file. We use a reply cache of the RPC transaction IDs of the `OP_PASSWRITE` operations to solve this problem. Although, `OP_MKOBJECT` is not idempotent, it is sufficiently cheap and stateless that we do nothing to make it idempotent.

Space Issues When an NFSv4 server delegates a file it specifies the amount of new data that the client can cache for the file, so that the client is guaranteed that there will be space on the server when it flushes the delegated file. When a client approaches this limit, it flushes the file's accumulated data to the server. We use the same approach, tracking both the provenance and data to determine when the client should flush its data.

7 Application Use Cases

Our use cases come from the Thermography Research Group at Iowa State University [22]. The group finds cracks in aerospace materials through frictional heating caused by rubbing of the crack surfaces due to vibration. This heating is imaged with an infrared camera.

The usual analysis procedure is to explore the measured data interactively using Python or Matlab. Without provenance tracking it is necessary to manually develop a script for analyzing the measured data through cutting and pasting commands between Python or Matlab and a text editor. Usually this script generates a graph to be used in a paper or presentation. Provenance provides the potential to use provenance to automate the script development. Provenance tracking also provides for implicitly tracking the relationship between calculated results such as graphs, the raw data from which those graphs originated, and the scripts and tools used to generate the graphics from the raw data. For the use case presented here, we used the Python wrapper class described in section 6.1.

7.1 Use Case: Determining Data Origin

Through approximately 400 experiments on 60 specimens over the course of a week, the research team at Iowa State developed a set of data quantitatively relating crack heating to the vibrational stresses on the crack. The experiment log for these data were stored in a series of XML files by the team's data acquisition system. A team member developed a plot relating crack heating as a function of crack length for two different classifications of vibrational stress σ: $20\text{MPa} < \sigma < 30\text{MPa}$ and $100\text{MPa} < \sigma < 120\text{MPa}$. In this case the interactive Python session used to create the plot was already reduced to a script. Our goal was to evaluate, based on provenance, which XML data files actually contributed to the plot.

This would have been a simple problem for a provenance aware filesystem, except that in order to figure out which data to use in the plot, the script read in *all* the XML data files. The provenance-aware filesystem alone is not capable of distinguishing the data used in the plot from the data read in by the script. In contrast, the layered provenance-aware Python/filesystem combination was able to both determine which XML data files were *actually used* in the plot, and in addition determine what Python scripts, modules, and libraries were used in generating that plot.

```
select inputfile.name
from Provenance.obj{plot}
  .%*{func1}.%+{func2}.%* as inputfile
where plot.name = "plot.png"
   and plot.type = "FILE"
   and inputfile.name glob "*.xml"
   and inputfile.type = "file"
   and func1.type = "func"
   and func2.type = "func"
```

This finds XML files in an ancestry chain that has passed through at least two Python functions. (We need two because of the details of the Python scripts involved.)

7.2 Use Case: Process Validation

The calculation routine we used to estimate crack heating temperatures during some of the experimental runs turned out to be flawed. While it calculated correct values most of the time, about 10% of the time it produced erroneous temperatures. We replaced the errant routine with a fixed routine that calculated correct values from the binary data files and inserted them into the XML experiment log alongside the (renamed) old values. We then generated plots illustrating the correction process, manually reviewing each case where the calculation routines calculated different values. The provenance allowed us to directly connect each generated plot to the binary data file from which it came and quickly identify and troubleshoot any cases in which the the new routine gave incorrect data.

7.3 Summary

While provenance-aware applications are generally useful and many researchers develop *ad hoc* solutions to the problems they solve, the ability to integrate such solutions with system level approaches increases the value of both the system-level provenance and the application-level provenance.

8 Overheads

While the main contribution of this work is in the new capabilities available from the system, we need to demonstrate that these capabilities do not impose excessive overheads. We are concerned with execution time overheads due to the additional work that we do to collect provenance, and space overheads due to the space used to store provenance.

We evaluate these overheads using a domain specific workload and the commonly-used Linux compile workload. Our domain-specific workload comes from computational biology. We used `blast` [2] to find the protein sequences in a species that are closely related to the protein sequences in another species. The workload, which is identical to the one used in the original PASS paper [16], consists of formatting two input data files with a tool called `formatdb`, processing the two files with `blast` and then massaging the output data with a series of perl scripts. This is mostly CPU bound, there is little provenance generated, and both the processing and typical queries are simple.

We use five different configurations, comparing unmodified ext2 to PASS for each, demonstrating the overheads imposed by different parts of the system.

- local: The file system is mounted locally.
- nfs: The file system is mounted via NFS.
- nfs-async: The file system is mounted in NFS asynchronous mode.
- nfs-2c: Two clients access a single NFS-mounted server.

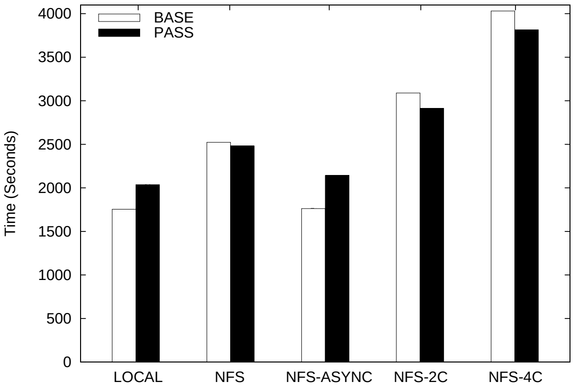

Figure 5: Figure shows the elapsed time for Linux compile for various configurations.

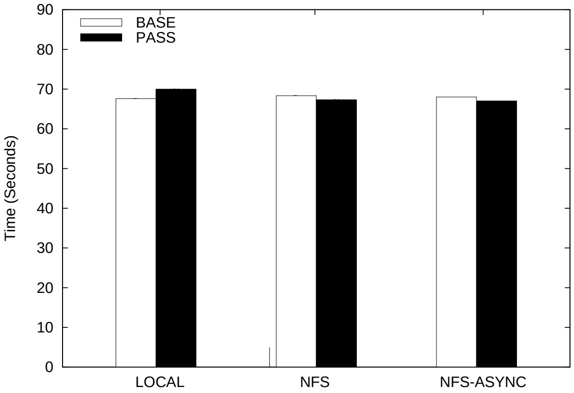

Figure 6: Figure shows the elapsed time for Blast workload for various configurations.

- nfs-4c: Four clients access a single NFS-mounted server.

For each configuration, we report elapsed times and space utilization.

Figures 5 and 6 compare PASS performance to that of the vanilla counterparts. In the local case, we impose a 16.1% overhead, but in the remote case, performance of the two systems is comparable, because the default NFS configuration runs in synchronous mode, and the disk write time dominates everything else. We also show the comparison when NFS is mounted in asynchronous mode, which produces a 21.5% overhead. When we increase the number of clients to two and four, our performance is again comparable to that of vanilla NFS.

The blast workload exhibits almost no overhead. We do not discuss other configurations for blast as they show equally small (and uninteresting) results.

	Data	Provenance	%overhead
Linux PA-local	1.25GB	190MB	14.79
Linux PA-nfs	1.25GB	165MB	12.8
Blast PA-local	5.6MB	820KB	14.28
Blast PA-nfs	5.6MB	152KB	2.64

Figure 7: Space overheads.

Table 7 shows the space overhead for both the Linux compile and blast workloads. We show only a single local and remote case, as adding more clients does not change the overhead percentage. It is interesting to note that the NFS cases generate less overhead, because client caching eliminates a significant number of records.

9 Conclusions

We have presented a second generation provenance-aware storage system that permits integration of multiple layers of provenance systems as well as network attached storage. These new features drive a layered architecture and dictate how provenance, data, and versions must flow through the system. The architecture has proved versatile enough to facilitate integration with application-level libraries and NFS. The ease with which we were able to integrate the application library with PASS convinces us that the architecture is substantially correct.

With our architecture, we make strong guarantees about the relationship between data and its provenance during both normal operation and after a crash. Finally, we demonstrate an end-to-end system encompassing provenance-aware applications and network-attached storage. We achieve all this with reasonable space and time overheads ranging between 0 and 16%.

References

[1] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. The Lorel query language for semistructured data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.

[2] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic local alignment search tool. *Molecular Biology 215* (1990), 403–410.

[3] Bose, R., and Frew, J. Composing lineage metadata with xml for custom satellite-derived data products. In *Proceedings of the Sixteenth International Conference on Scientific and Statistical Database Management* (2004), IEEE Computer Society, pp. 275–284.

[4] Braun, U., Muniswamy-Reddy, K.-K., Holland, D. A., and Seltzer, M. The Provenance Challenge: PASS. `http://twiki.ipaw.info/bin/view/Challenge/PASS`.

[5] Cattell, R. *The Object Database Standard: DBMG-93*. Morgan Kaufman, 1994.

[6] Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (May 1994), pp. 313–324.

[7] Edlund, S., Grandison, T., Hui, J., and Johnson, C. Supervision and Discovery of Electronic Communications in the Financial Services Industry. In *Proceedings of the 1st International Workshop on Governance, Risk and Compliance - Application Information Systems* (2008).

[8] Foster, I., and Kesselman, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing* (Summer 1997).

[9] Foster, I., Voeckler, J., Wilde, M., and Zhao, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).

[10] GenePattern. `http://www.broad.mit.edu/cancer/software/genepattern`.

[11] Halcrow, M. A. ecryptfs: An enterprise-class encrypted filesystem for linux. *Proceedings of the 2005 Ottawa Linux Symposium* (2005), 201–218.

[12] Holland, D. A., Braun, U., Maclean, D., Muniswamy-Reddy, K.-K., and Seltzer, M. I. A Data Model and Query Language Suitable for Provenance. In *Proceedings of the 2008 International Provenance and Annotation Workshop (IPAW)* (June 2008).

[13] Jim Gray, A. R. *Transaction Processing: Concepts and Technique*. Morgan-Kaufman, 1993.

[14] King, S. T., and Chen, P. M. Backtracking Intrusions. In *SOSP* (Bolton Landing, NY, October 2003).

[15] Moreau, L., et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*. Published online. DOI 10.1002/cpe.1233, April 2008.

[16] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (June 2006).

[17] Provenance aware service oriented architecture. `http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome`.

[18] The First Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge`.

[19] The Second Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge`.

[20] SEC Rule 10b-5. `http://www.sec.gov/rules/final.shtml`.

[21] Somayaji, A., and Forrest, S. Automated Response Using System-Call Delays. In *USENIX Security Symposium* (Berkeley, CA, 2000).

[22] Iowa State University Thermography Research Group. `http://thermal.cnde.iastate.edu`, 2007.

[23] Widom, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Conference on Innovative Data Systems Research* (Asilomar, CA, January 2005).

[24] Zhao, J., Goble, C.and Greenwood, M., Wroe, C., and Stevens, R. Annotating, linking and browsing provenance logs for e-science.