

# Issues in Automatic Provenance Collection

Uri Braun, Simson Garfinkel, David A. Holland,  
Kiran-Kumar Muniswamy-Reddy, Margo I. Seltzer

Harvard University, Cambridge, Massachusetts  
`pass@eecs.harvard.edu`

**Abstract.** Automatic provenance collection describes systems that observe processes and data transformations inferring, collecting, and maintaining provenance about them. Automatic collection is a powerful tool for analysis of objects and processes, providing a level of transparency and pervasiveness not found in more conventional provenance systems. Unfortunately, automatic collection is also difficult. We discuss the challenges we encountered and the issues we exposed as we developed an automatic provenance collector that runs at the operating system level.

## 1 Introduction

Today’s provenance management systems usually take one of two approaches to provenance collection: Either users enter it manually or applications explicitly collect provenance and enter it into a database. There is, however, a third model: *automatic provenance collection*. In automatic collection, the system observes the actions of users and programs and derives provenance, storing it without user or application involvement.

Automatic collection is a powerful approach, because it eliminates user error, consistently collects provenance across all applications, and captures more complete provenance than systems relying on a user’s or application developer’s assumptions about provenance. For example, automatic provenance collection in the operating system allows us to: identify system configuration changes (*e.g.*, new tools or libraries), identify environment variable modifications that alter program behavior, debug faulty builds that are missing dependencies, identify the source and creation of unusual files, and create scripts that produce objects.

In earlier work we described a prototype Provenance-Aware Storage System (PASS), built on Linux, that automatically collects provenance at the operating system level [18]. PASS is similar to systems such as ClearCase [5], GenePattern [9], and Vesta [10]. These systems observe users’ and applications’ activities, recording the provenance captured in these activities. PASS takes this one step farther, observing *all* processes that run on a PASS-enabled operating system, generating provenance for objects that do not have provenance (*i.e.*, are *unprovenanced*), and attaching complete system-level provenance to objects that are created on a provenance-aware file system. We capture low-level details like the operating system, kernel modules loaded, installed libraries, and process environment.

We found that automatic OS-level provenance collection is useful, complementing existing approaches. However, we exposed a number of challenging issues that arise from automatic collection. This paper introduces automatic provenance collection and discusses the more interesting challenges that arise in building these systems.

In Section 2 we define automatic provenance collection, placing it in the context of existing provenance solutions. In Section 3 we discuss the problems that arise when designing and building systems with automatic provenance collection. In Section 4, we present use cases where disclosed and observed provenance systems together provide more powerful solutions than either independently. In Section 5 we introduce *provenance pruning*, the deletion of provenance, and discuss strategies for implementing it. In Section 6 we discuss the privacy implications of automatic provenance collection. In Section 7 we discuss automatic provenance systems and technologies from which we can borrow in constructing automatic provenance systems, and in Section 8 we conclude.

## 2 What *is* Automatic Collection?

An automatic collecting system transparently records provenance for *all* activities it performs by observing the sequence of operations executed and translating relevant ones into provenance. For example, when a process begins running, the operating system identifies several process provenance attributes<sup>1</sup>, such as the executable, operating system, loaded kernel modules, libraries, environment, and command line.

The system continues to collect provenance about the running process, recording (for example) input sources. Whenever the process creates or modifies an object, the process’s provenance is assigned to the written object. We call this form of provenance collection *observed*, because the system derives provenance from the events that it observes. An observed provenance system does not necessarily understand the semantics of its observations, so it must record everything that is potentially part of an object’s provenance. This can lead to *false provenance* if the observing system does not perform detailed information flow analysis. Section 2.1 discusses existing systems that use observed provenance.

Most existing provenance systems use *disclosed* provenance. In disclosed provenance systems, users or applications present provenance to the system, using the provenance system merely as a storage and query engine. There are several kinds of disclosed provenance. *Manual* provenance, sometimes called *annotation*, is entered by users. For example, the provenance of data entered manually by a user must itself be manually entered. *Specified* provenance describes an object’s intended provenance in a structured way, typically by directing the system to produce the object via various transformations or workflows. For examples, workflow-based systems [23, 32] and makefiles capture intended provenance by describing how a target is created from its sources. The workflow systems

---

<sup>1</sup> We think of processes as having provenance so that we can transfer a process’s provenance to objects it creates.

frequently then generate true provenance from the execution of these workflows, while `make` does not. Instead, the canonical software development environment relies on a separate component, a source code control system, to store semantic provenance. In these systems, the difference between successive versions precisely identifies *what* changed, but the rationale is entered as manual provenance in the form of commit messages.

## 2.1 Observed-Provenance Systems

There are several domain-specific observed provenance systems. GenePattern [9] is a working environment for computational biology and biomedical research. It tracks provenance for the objects created in the environment. Clearcase [5] and Vesta [10] are environments designed for software development. Like our provenance-aware storage system (PASS), both Clearcase and Vesta use a customized file system to track provenance. Unlike PASS, both Clearcase and Vesta center their design on disclosed provenance, assuming that users execute commands specified in a control file. However, both allow some form of observation in which the system observes a user's actions creating a control file (script) to re-derive objects. Running in this observed mode is not the norm and requires explicit action by the user.

Observed provenance systems provide the benefit that provenance collection is automatic, requiring no user intervention. Observed systems collect and maintain provenance of objects by default, so *all* new data becomes provenanced. The major disadvantage of observed provenance systems is that they can only capture provenance to which they are exposed, and this frequently produces provenance with less semantic meaning than disclosed provenance systems. The next section illustrates how disclosed provenance systems complement observed provenance systems.

## 2.2 Disclosed-Provenance Systems

The vast majority of today's provenance systems use disclosed provenance. These systems require that the user describe a workflow, and then they provide an engine that executes the workflow. The combination of the workflow specification and the result created by running the engine on the workflow creates provenance. The line between observed and disclosed systems is not always crisp. In workflow-specification systems, the actual workflow engine is an observed provenance system; however, it is an observed system that relies on an underlying specification in order to function. Without the specification, the workflow engine cannot generate provenance. For expository purposes, we classify these systems as disclosed provenance systems, because they require the user to disclose explicitly the intended provenance of a result.

The myGrid [32] workflow enactment engine records provenance for each step in a workflow, including inputs and outputs, storing the provenance in a central provenance repository. The PASOA [23] project provides APIs that clients and services use to record provenance during workflow execution. Chimera [7] offers

a virtual data system that provides a virtual data language (VDL) and a virtual data catalog (VDC). The VDC implements a virtual data schema that defines the objects and relations that can be used to capture descriptions of program invocations and to record potential or actual invocations. The Earth System Science Workbench (ESSW) [8] is a data management infrastructure used for processing satellite imagery. The CMCS (Collaboratory for the Multi-scale Chemical Sciences) [21] uses a portal and metadata-aware content store as a base for building a system to support inter-domain knowledge exchange in chemical science. All these systems rely on some sort of specification or explicit API for provenance disclosure. Disclosed provenance systems also make use of user annotations. In fact, some systems [29] rely mostly on user-provided annotations.

Disclosed provenance systems usually provide richer semantic knowledge than observed systems. However, to obtain this benefit, users are restricted to using provenance-aware tools and must conduct their work within the confines of applications that understand how to collect provenance. Thus, the responsibility for provenance collection is dispersed throughout many different tools.

### 2.3 Observed Provenance Complements Disclosed Provenance

Observed and disclosed provenance provide complementary solutions. An ideal solution provides the full semantic knowledge of disclosed provenance using the automatic and transparent collection of observed provenance systems. While there has been significant research on disclosed provenance systems, there has been significantly less on observed provenance systems. To build systems with the benefits of both approaches, we must understand and overcome the challenges that automatic collection presents. We next discuss some of those challenges, and then in Section 4, we return to examples of how observed and disclosed systems complement one another.

## 3 Challenges

An observed provenance system faces the challenge of transforming observations into disclosed provenance. This transformation requires solving several problems. There may be several types of mismatches between the observed and desired provenance. We use the term *granularity* to refer to the mismatch between the operating system’s observation of a sequence of system calls and the scientific user’s desire to record provenance on an experiment.

Reconciling these mismatches leads to challenges in creating and maintaining provenance ancestry; automatic provenance collection can lead to cyclic ancestry, which is conceptually unacceptable. Versioning is one way to cope with cyclic ancestry, but it presents challenges similar to those found in automatically versioned file systems [25]. In the rest of this section, we explore each of these issues in more detail.

### 3.1 Granularity

Granularity refers to the types of objects for which a system maintains provenance. Coarse grain provenance might describe data and results produced by an entire research initiative (e.g., the Human Genome Project). At the other extreme, the programming languages and systems communities are sometimes interested in extraordinarily fine-grained provenance, such as byte- or bit-level provenance [17, 26].

Users are most frequently interested in coarse-grained provenance, such as an experiment or analysis. However, automatic collection systems most naturally operate at a finer grain.

Our prototype PASS collects system call events, recording provenance on a per-file basis. This is the most natural model for an operating-system-based provenance collector, but we can increase the utility of our system by creating coarse-grained views that are interesting to users. If the coarse-grain view is the only interesting view, it is best to perform this conversion (from fine-grain to coarse-grain) at collection time, reducing the provenance overhead.

Automatically identifying and constructing these coarser-grain views remains an open research problem for both data and the events that produce data. Our PASS prototype supports user annotations, which are a manual way to provide coarser views. We are exploring other approaches such as describing classes of files for which provenance is unnecessary (e.g., temporary files), in which case the output files of interest will have the provenance of the entire transformation. Identifying meaningful events (i.e., event granularity) poses a much larger problem, as it is tightly coupled to versioning.

### 3.2 Versioning

A system that both tracks provenance and allows data modification is inherently versioned. Each modification to a provenanced object changes the provenance of the object and creates a new version of that object, regardless of whether the system actually retains those different versions. On a system that does not explicitly track such versions, provenance provides the connections between versions of objects and distinguishes those objects at different points in their lifetimes.

Like an automatic versioning file system, a provenance-aware system must decide what constitutes a “modification” and thus when to declare new versions. The simplest approach, used by Wayback [4], creates a version on every write call. This simple approach is impractical for automatic provenance collection, because it yields too many meaningless versions. Another method, used in versioning file systems [15, 25], is copy-on-write, where a new version is created on the first write to a file between an open and close. A slightly different approach, copy-on-change [12], creates new versions only if a write actually modifies the object. A third alternative, used in a number of commercial and research systems [11, 13, 22, 24], is to take snapshots, or checkpoints. In this model, a snapshot contains the version of each file that existed when the snapshot was taken. These systems

typically take snapshots at regular intervals rather than being triggered by system activity. In PASS, we do copy-on-write: we consider a version complete and “frozen” on the last close (or on `fsync`) and on the next write a new version is created. A new version is also created if a file is truncated to zero length.

Copy-on-write and copy-on-change both involve grouping related sets of modifications into a single new version. This is a form of event granularity abstraction: combining multiple observed write operations into a single conceptual write operation. This merging process requires extreme caution in automatic provenance collection, because it can lead to an even more vexing problem: cycles.

### 3.3 Cycles

Any modification grouping mechanism introduces the possibility of cycles in the provenance. Cycles in provenance are nonsensical; an object cannot be descended from itself. Cycles may even violate causality: an object may not be created by another object it had somehow previously emitted. So automatic systems must avoid creating provenance cycles. Unfortunately, avoidance is difficult.

Consider the common behavior of a program that first reads and then writes a file. Either the write creates a new version of an object or it creates a provenance cycle. This simple case is easy to resolve, but suppose this process repeats in a loop. The provenance collector cannot observe the loop, only the read and write actions; to avoid creating multiple versions it must infer the existence of the loop and take appropriate steps.

Inferring loops in a single process is tractable, but systems of interacting processes can also produce cycles. Local knowledge is not necessarily sufficient to identify the situation, much less correct it. Consider two processes, P and Q:

P	Q
read a	
	read b
write b	
	write a

If no new versions are created, these processes produce a cycle, yet nothing about P or Q in isolation makes that evident. Simply creating additional versions does not adequately solve the problem; cyclic workloads produce too many versions. (For example, consider a parallel, iterative algorithm.) The solution is to abstract P and Q into a single higher-level conceptual entity, making the cyclic data flow internal to this higher-level entity. Internal data flow need not be provenanced.

Our prototype PASS maintains a global relationship graph and checks it for cycles every time an edge is added. This allows it to create higher-level abstractions. Our current algorithm creates a new version of every process involved in a cycle and then merges these versions together. The newly created merged object becomes the parent of all the files involved, without creating new versions. This handles many common cases with minimal overhead, but fails in some circumstances.

The cycle problem is inherent in any attempt to reduce the number of versions generated and thus inherent in event granularity abstraction. The same problem can and will appear in any automatic collection system. These problems do not appear in disclosed provenance systems, because statements of disclosed provenance are initiated by a human developer, who understands the granularity of the operations. In observed systems, we must deduce the granularity to identify the semantically correct grouping.

Cycle detection and elimination has been a major preoccupation of our recent research; nonetheless, we still do not have a satisfactory algorithm; it remains an open problem [2].

## 4 Integrating Observed and Disclosed Provenance

Systems that support both observed and disclosed provenance offer powerful features that cannot be obtained using either type alone.

Consider running the GenePattern [9] system on top of PASS. GenePattern possesses the semantic knowledge to record the exact analysis used to transform input data to output results. However, it does not know what version of the math library or what floating point processor was used. This information may be available to GenePattern, but not always, nor in a portable or reliable fashion. In contrast, PASS is integrated with the operating system, handling such issues natively. Together, GenePattern and PASS can answer the query, “Why did this identical transformation produce different results last week and this week?”

Alternatively, suppose that outside of the GenePattern environment a researcher “fixed” an input file. No provenance query in GenePattern can detect that the input file changed, explaining how identical analyses on “the same” input file yield different results.

As discussed previously, a significant challenge for observed provenance systems is identifying a semantically meaningful level of granularity. When a disclosed provenance system sits atop PASS, that disclosed provenance system provides precisely the information PASS needs to construct these coarser views.

## 5 Pruning

Provenance adds storage overhead. *Provenance Pruning* is the act of selectively removing provenance to save space. In general, pruning removes the provenance for entire objects; however, at the end of this section we consider approaches that erase only part of an object’s provenance.

Storage overhead can grow rapidly. Left unchecked there is nothing to prevent the provenance from dwarfing the data it describes. Such large space overhead can also harm query performance. For example, in recent work [18] we showed that small changes to a source file in the Linux kernel generated approximately two kilobytes of additional provenance when the kernel was rebuilt.

Some provenance can be pruned immediately at collection time. Users may not be interested in recording configuration files, such as `.bashrc` and `.profile`,

as ancestors of the bash shell. Similarly, users may not be interested in some output files, such as temporary files or `/dev/null`. In other cases, it would be useful to identify entire processes and the objects they modify as not requiring provenance. For example, `makewhatis`, which indexes man pages, examines all of them before writing out the index file; the provenance of that index file is thus quite voluminous and also completely uninteresting. Such specification of unprovenanced objects allows the system to prune provenance before it is ever recorded to disk.

There are also opportunities for provenance pruning after collection. Deleted files without descendants are good candidates for pruning at any time. It is useful to think of provenance as forming a tree where parent nodes are those nodes accessed as input during the creation of their children. In such a tree, the deletion of a leaf node can be accompanied by the deletion of that node's provenance, since, by construction, that node's provenance cannot be needed by any other node. We call this *bottom-up* pruning. It is interesting to ask if *top-down* pruning ever makes sense. On a long-running system, it might be useful to prune provenance to present the illusion that provenance history began at some time  $T$ , later than the actual beginning of that system's provenance. Alternately, it might be useful to declare some node as the new eldest ancestor and remove all its parental provenance.

Intermediate files provide another opportunity for pruning. Files that can be easily recreated by capturing the entire process that created them are good candidates for provenance pruning. For example, in a build environment, the object files can be recreated if necessary, so it may not be necessary to keep their provenance.

## 5.1 Policies

The pruning strategies described in the previous section implement policy decisions. Such policy decisions should be site-specific. Storage policies might place limits on the absolute amount of provenance retained or limit provenance as a proportion of the data it describes. Retention policies dictate when provenance can be erased, e.g., when no file in its ancestry remains.

## 5.2 Other provenance reduction strategies

**Supernodes** Short of erasing provenance, it is sometimes possible to compact or summarize existing provenance to save space. For example, it might be desirable to compress the provenance for a subtree whose internal nodes have been deleted. In this case, the system combines the provenance from internal nodes into a *supernode* for the child.

**Virtual Nodes** Some collections of attributes might recur frequently. These attributes can be combined and included in a *virtual node* and referenced where applicable. We already use an approach similar to this in our PASS prototype. We



store command lines and environments in their own database and refer to them by a unique ID number [18]. An alternate and more general representation would be to represent them as a provenanced object or virtual node. This approach provides a lossless storage reduction method.

**Removing attributes** Rather than combining attributes, we might choose to remove attributes if we can identify that they are irrelevant. As in pruning, attributes could be removed during collection or later on. Irrelevant attributes could be removed during collection or during later pruning. Identifying attributes that do not need to be recorded in provenance is another site-specific policy decision.

### 5.3 Pruning in PASS

In our PASS prototype, we do not yet provide a pruning policy specification mechanism. Instead, we provide a utility, *ptrunc*, that enables a user to explicitly truncate provenance, eliminating uninteresting ancestry [18].

## 6 Privacy and Security

Automatic provenance collection presents serious privacy challenges, because such a system collects significant information about its users. Replying to an email message by pasting a paragraph from a web page might cause the system to capture the sender of the original email and the time it was received; the time the message was read; the web addresses the user visited; and any intermediate drafts that were composed but not sent.

Information leakage is the primary privacy risk introduced by automatic collection. By its very nature, automatically collected provenance is *invisible* to users, and this invisibility provides an easy channel through which sensitive information can be released. For example, consider a manager composing an employee review that includes input solicited from the employee's colleagues. Presenting the review and its provenance to the employee reveals the identity of the colleagues who contributed to the review [14]. A manager who had to explicitly disclose the review's provenance to the system would be unlikely to make such a mistake.

Deciding what provenance should be captured, where it should be stored, how that store should be protected, and when (if ever) the stored provenance should be purged are current research areas. For example, provenance is sometimes stored in the document itself as a header [19] or in an alternative data stream [16], making it easy to keep document and provenance together. This approach enables privacy-friendly systems and applications that automatically detect and prevent violations of privacy or data sharing policies [30]. On the other hand, provenance that travels invisibly with a document might also compromise privacy if users are unaware of its existence.

While no provenance-aware storage systems are in widespread use, both the Microsoft Word and Adobe Acrobat file formats store hidden data that reveals elements of a document’s history, a kind of provenance. Experience with these formats illustrates that automatic collection introduces significant privacy risks:

- In June 2000, *The New York Times* posted an Acrobat file on its website containing names of Iranians who had assisted the CIA in the 1953 Iranian coup. Although the paper had tried to remove the names from the Acrobat file, the information was recovered and posted in an unredacted form on another website [31].
- In June 2002, the US Justice Department released a “Workplace Diversity” report that contained embarrassing information that the Department had attempted to delete [6].
- In March 2004, the SCO Group distributed Microsoft Word files to journalists containing hidden text that revealed the company’s legal strategy in its anti-Linux lawsuits[27].

On the surface, automated provenance collection also violates many traditional principles of Fair Information Practice [20]. For example, the *collection limitation* principle is violated, because no limits are placed on data collection. The *purpose specification* is violated, because no purpose for the data is specified at the time of collection. On the other hand, FIP principles such as *individual participation* and *security safeguards* could be used as requirements guidelines when designing systems for automatically collecting provenance. Embracing automatic provenance collection necessitates understanding and addressing the privacy implications.

We understood early that provenance security was both crucial and under-researched. We do not provide any access controls in our current prototype, and our early adopters are all users who freely share their data and analyses. In parallel, we undertook a small pilot project to identify the key features a provenance security model requires. That study revealed that we need two independent security models: one that provides conventional access control over provenance attributes and a second that provides access control over the branches of the ancestry tree [3]. We are currently implementing these models and studying the challenges in composing the two models.

## 7 Related Work

In Sections 2.1 and 2.2 we discussed alternate approaches to provenance. In this section, we discuss a few other approaches and related technologies that influence observed provenance systems or from which we can take advantage of prior knowledge.

### 7.1 Versioning File Systems

As mentioned earlier, the versioning challenges of observed provenance systems are similar to those in versioning file systems. The Elephant file system creates

a new version of a file on every write and does not immediately erase old versions [25]. As such, it highlights many of the issues we encounter with versions. Like PASS, Elephant must cope with the overhead associated with creating a new version on every write, and makes pruning essential. Elephant supports three pruning policies: keep all, one, or landmarks. Unlike PASS, Elephant versions file (and directory) data, and it does not gather provenance information.

## 7.2 Observed Provenance

PASS is not the only system to collect low level operations and attempt to infer semantic meaning from them. Both the Lineage File System and Transparent Result Caching take this approach.

Transparent result caching (TREC) captures system calls and tracks process lineage including parent processes, child processes, input files and output files [28]. Unlike PASS, all tracing occurs in user space. Read and write system calls are not intercepted. TREC relies instead on the open mode to infer whether files are inputs or outputs, a tradeoff between performance and accuracy. Both systems support makefile generation and detection of changed dependencies, but PASS provides additional query support unavailable in TREC.

Like PASS, the Lineage File System focuses on executables, command lines and input files as the source of provenance [14]. Unlike PASS, it ignores the hardware and software environment in which such processes run. A second, and perhaps more important, difference is that provenance collection is delayed in the Lineage File System and it is performed by a user-level thread that writes the lineage data to an external database. As a result, the tight coupling we require between data and provenance is lost, as is a significant part of the benefit. Since the Lineage File System stores its lineage records in a relational database, the query language is SQL. In our implementation, we use a simple key/value storage schema so that a variety of schema layers can be provided.

## 7.3 Exposing Semantic Knowledge

The observed systems considered thus far all try to infer semantic knowledge from a collection of events. Another is to isolate the semantic knowledge as a user specified component. Magpie combines observed provenance recorded in trace logs with a user specified event schema to construct a workflow model [1]. The user specified event schema specifies the rules for combining related events. For example, specifying which filesystem read and write operations correspond to a specific webserver request. Magpie differs from PASS in its use of a user specified event schema and the recording of provenance in separate trace files.

## 8 Conclusions

Automatic provenance collection is an important and powerful technique that complements existing provenance solutions. However, it carries challenges that

do not appear in these other systems. Our group has developed a provenance-aware storage system prototype that is now ready for limited experimental use. We encourage the community to try our prototype, develop an appreciation for the power of automatic collection, and tackle some of the fundamental research challenges that remain. This area holds great promise, but only through the construction of a variety of systems that automatically collect provenance at different levels, from the operating system to provenance aware applications, will we identify the right solutions to the challenges outlined here.

## References

1. P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, 2004.
2. U. Braun, D. A. Holland, K.-K. Muniswamy-Reddy, and M. Seltzer. Coping with cycles in provenance. <http://www.eecs.harvard.edu/~syrah/pass/pubs/cycles.pdf>.
3. U. Braun and A. Shinnar. A Security Model for Provenance. Technical Report TR-04-06, Harvard University, Jan. 2006.
4. Brian Cornell and Peter Dinda and Fabian Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the USENIX 2004 Annual Technical Conference, FREENIX Track*, 2004.
5. ClearCase. <http://www.ibm.org/software/awdtools/clearcase>.
6. R. Edmonds. Justice department hid parts of report criticizing diversity effort. *Associated Press*, Oct. 31 2003.
7. I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR*, Asilomar, CA, Jan. 2003.
8. J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 180–189. IEEE Computer Society, 2001.
9. GenePattern. <http://www.broad.mit.edu/cancer/software/genepattern>.
10. A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.
11. D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, January 1994.
12. K. Muniswamy-Reddy and C. P. Wright and A. Himmer and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, March/April 2004.
13. E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS-7)*, pages 84–92, Cambridge, MA, 1996.
14. Lineage File System. <http://crypto.stanford.edu/~cao/lineage.html>.
15. K. McCoy. *VMS File System Internals*. Digital Press, 1990.

16. Microsoft. How to use ntfs alternate data streams. July 13 2004.
17. S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 8. Morgan Kaufmann, 1997.
18. K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, June 2006.
19. Nost. Definition of the flexible image transport system (FITS), 1999.
20. Organisation for Economic Co-operation and Development. Guidelines on the protection of privacy and transborder flows of personal data, 1980.
21. C. Pancerella et al. Metadata in the Collaboratory for Multi-scale Chemical Science. In *Dublin Core Conference*, Seattle, WA, 2003.
22. Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. <http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf>.
23. Provenance aware service oriented architecture. <http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome>.
24. S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, January 2002.
25. D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.
26. J. Seward. Valgrind, an open-source memory debugger for GNU/Linux. <http://valgrind.org>, 2005.
27. S. Shankland and S. Ard. Document shows SCO prepped lawsuit against BofA. *News.Com*, Mar. 4 2004.
28. A. Vahdat and T. Anderson. Transparent result caching. Technical Report CSD-97-974, 8, 1997.
29. M. Wan, A. Rajasekar, and W. Schroeder. An Overview of the SRB 3.0: the Federated MCAT. <http://www.npaci.edu/DICE/SRB/FedMcat.html>, September 2003.
30. D. J. Weitzner, H. Abelson, T. Berners-Lee, C. Hanson, J. Hendler, L. Kagal, D. L. McGuinness, G. J. Sussman, and K. K. Waterman. Transparent accountable data mining: New strategies for privacy protection. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2006.
31. E. Wong. Web site lists Iran coup names. *The New York Times*, June 24 2000.
32. J. Zhao, M. Goble, C. and Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science.