

## Better understanding data requires tracking its history and context.

BY LUCIAN CARATA, SHERIF AKOUSH,  
NIKILESH BALAKRISHNAN, THOMAS BYTHEWAY,  
RIPDUMAN SOHAN, MARGO SELTZER, AND ANDY HOPPER

# A Primer on Provenance

ASSESSING THE QUALITY or validity of a piece of data is not usually done in isolation. You typically examine the context in which the data appears and try to determine its original sources or review the process through which it was created. This is not so straightforward when dealing with digital data, however: the result of a computation might have been derived from numerous sources and by applying complex successive transformations, possibly over long periods of time.

As the quantity of data that contributes to a particular result increases, keeping track of how different sources and transformations are related to each other becomes more difficult. This constrains the ability to answer questions regarding a result's *history*, such as: What were the underlying assumptions on which the result is based? Under what conditions does it remain valid? What other results were derived from the same data sources?

The metadata that needs to be systematically captured to answer those (or similar) questions is called *provenance* (or *lineage*) and refers to a graph describing the relationships among all the elements (sources, processing steps, contextual information and dependencies) that contributed to the existence of a piece of data.

This article presents current research in this field from a practical perspective, discussing not only existing systems and the fundamental concepts needed for using them in applications today, but also future challenges and opportunities. A number of use cases illustrate how provenance might be useful in practice.

**Where does data come from?** Consider the need to understand the conditions, parameters, or assumptions behind a given result—in other words, the ability to point at a piece of data, for example, research result or anomaly in a system trace, and ask: Where did it come from? This would be useful for experiments involving digital data (such as *in silico* experiments in biology, other types of numerical simulations, or system evaluations in computer science).

The provenance for each run of such experiments contains the links between results and corresponding starting conditions or configuration parameters. This becomes important especially when considering processing pipelines, where some early results serve as the basis of further experiments. Manually tracking all the parameters from a final result through intermediary data and to original sources is burdensome and error-prone.

Of course, researchers are not the only ones requiring this type of tracking. The same techniques could be used to help people in the business or financial sectors—for example, figuring out the set of assumptions behind the statistics reported to a board of directors, or determining which mortgages were part of a traded security.

**Who is using this data?** Instead of tracking a result back to its sources,







you can capture provenance to understand where that result has been subsequently used or to find out what data was further derived from it. For example, a company might want to identify all the internal uses of a certain piece of code in order to respect licensing agreements or to keep track of code still using deprecated or unsafe functions that need to be removed.


Using similar mechanisms, end users should be able to track what personal information is used by a mobile application and determine whether it is displayed locally or sent over the network to a third party. The same use case covers the general propagation of erroneous results, when we need to understand what pieces of data have been invalidated by the discovery of an error.

**How was it obtained?** Provenance can also be used to obtain a better understanding of the *actual process* through which different pieces of input data are transformed into outputs. This is important in situations where computer engineers or system administrators need to debug the problems that arise when running complex software stacks.


In cases where it is possible to differentiate between correct and erroneous system output, comparing their provenance will point to a list of potential root causes of the error. In more complex scenarios, the issue might not be directly linked to particular outputs but to an (undesired) change in behavior. Detecting system intrusions or explaining why the response tail latency has increased by 20% for a server are good examples. In those cases, grouping outputs with similar provenance could be used for identifying normal versus abnormal system behavior and explaining the differences between the two.

### Provenance Systems

Together, the three use cases provide an overview of the ideal provenance application space, but they do not describe the technical details involved in making those applications possible. To realize each scenario in practice, one or more *provenance systems* need to be integrated into the data-processing workflow, becoming responsible for capturing provenance, propagating it among related components, and making it accessible to user queries.



**As the quantity of data that contributes to a particular result increases, keeping track of how different sources and transformations are related to each other becomes more difficult.**



In many ways, you might already be running a specialized version of such a system: all auditing, tracing frameworks, or change-tracking solutions collect some form of provenance, even though they might not identify it as such. The advantage of thinking about provenance as a stand-alone concept is the ability to use this metadata in a principled way, allowing result verifiability and complex historic queries regardless of the underlying mechanisms used to collect it and across applications and software stacks.

Historically, provenance systems were the focus of research in the database field, with the aim of understanding how and when materialized views should be updated in response to changes in the underlying tables.<sup>9</sup> Because of the well-defined relational model, it has proven possible both to derive precise provenance information from queries<sup>7</sup> and to develop formalisms that allow its concise representation.<sup>13</sup> This has been further extended in systems such as Trio,<sup>28</sup> allowing records to incorporate an associated uncertainty, which can be propagated across multiple queries by using provenance.

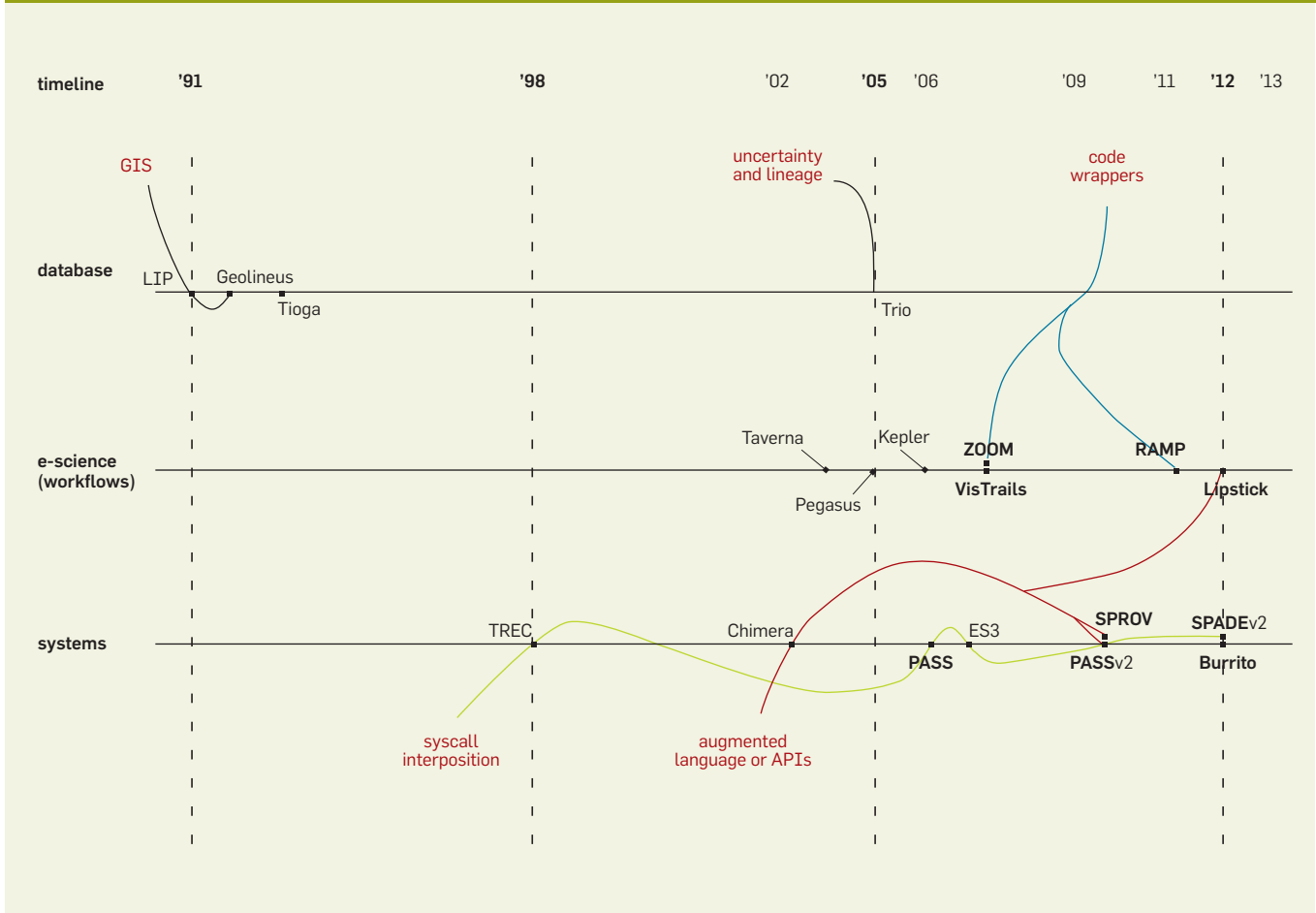
In contrast, capturing provenance for applications performing arbitrary computations (not restricted to a small set of valid transformations) has proven more challenging. Research efforts in this area have focused on the collection of provenance at particular points in the software stack (by modifying applications, the runtime environment, or the kernel).

The accompanying figure presents a general timeline of provenance systems. This article looks at the characteristics of eight of these (PASS, SPADE, VisTrails, ZOOM, Burrito, SPROV, Lipstick, and RAMP), each representative of a larger class of solutions:

**Operating-System Level.** PASS<sup>22,23</sup> and SPADE<sup>12</sup> investigate provenance by observing application events such as process creation or I/O. Those are then used for inferring dependencies among different pieces of data.

**Workflows.** VisTrails<sup>26</sup> and ZOOM<sup>2</sup> are workflow-management systems with the ability to track provenance for the execution of various workflows and (in the case of VisTrails) for the evolution of the workflows themselves.

## A timeline of provenance systems.



**Application Level.** Burrito<sup>14</sup> tracks user-space events, while also supporting additional user-provided annotations. SPROV<sup>16</sup> focuses on the security of provenance and provides a thin wrapper around the standard C I/O library. A newer version is capable of using provenance captured by other systems, such as PASS.

**Big Data.** Lipstick<sup>1</sup> and RAMP<sup>24</sup> both tackle the problem of tracking provenance in big-data scenarios (MapReduce jobs).

It is the properties of these systems that define what can be recorded and with what trade-offs, overhead, and security implications.

### Provenance System Properties

Effectively using provenance systems in practice requires understanding a number of aspects related to:

- The exact metadata being captured.
- The effort required for integrating provenance systems within existing workflows (running special kernels,

making runtime changes, or linking applications with provenance libraries).

- Understanding how provenance metadata can be later queried.
- Evaluating the overhead imposed by those systems.
- Security issues added by provenance, which might require different access controls from those of the data itself.

This article categorizes the properties of the systems selected as representative according to these features, referring to the motivating use cases as required.

**What can it capture?** The metadata captured by provenance systems typically relates the state of digital entities (files, tables, programs, network connections, and so on) at different stages in their lifetimes to historic dependencies on other entities or processes. In this context, two concepts are fundamental for determining what is captured and how: *granularity* and *layering*.

*Granularity.* The granularity of cap-

ture refers to the size of basic primitives that accumulate provenance within a system. Consider a scientist who uses a configuration file storing various experiment parameters as one of the inputs to a simulation program. Capturing provenance at file granularity will establish the dependency between the simulation program and the configuration file name. The scientist is interested in understanding the relationships between the simulation results and individual parameters in the file, however, and this requires capture at subfile granularity.

The exact meaning of varying granularity (from fine to coarse) also depends on the underlying data model of the application. For example, database-provenance systems could store provenance metadata for an entire table, a row within the table, or for each cell. Provenance capture at the table level is coarse grained and can answer questions such as: From which other tables has table X derived its data? Finer granularities would determine the relation-

ships between individual rows or cells. Of course, multiple granularities can be considered at the same time.

Systems such as PASS<sup>6</sup> capture provenance by intercepting system calls made by applications as they execute. At this level, provenance is fine grained and can provide a detailed image of an application's execution and dependencies.

The *noise levels* in the collected data, however, are also elevated, making it harder to extract useful information. Consider a Python script that copies one file to another. When running the script, the Python interpreter will first read and load any required modules from disk. Thus, beyond the dependency on the actual input, the final provenance graph will link the output file to all the Python modules used by the interpreter. This extra data can make it difficult to sift through the provenance graph as an end user, so, generally, heuristics are needed to determine which entities are important and which should be ignored.

Workflow systems such as VisTrails<sup>26</sup> avoid the noise problem and can capture provenance at any granularity, because the processing steps and their dependencies are explicitly declared by the end user. Such systems, however, are also inherently limited to recording only those data transformations that were part of the defined workflow.

*The  $n$ -by- $m$  problem.* Independent of the system that is chosen, accurately determining the dependencies between input and output data may not be possible. This is illustrated by the  *$n$ -by- $m$  problem*, where a program reads  $n$  input files and writes  $m$  output files. Even when tracing system calls for individual reads and writes, it is not possible to infer which reads affected a particular write, so the provenance graph has to link each output file to all of the inputs. A system that is unaware of the semantics of individual data transformations within a process will always present a number of such false-positive relationships. Both PASS and VisTrails have this problem, as they treat the process or each workflow step as a black box.

The  *$n$ -by- $m$  problem* can be solved by capturing provenance at an even finer granularity. This can be done using binary instrumentation tech-

niques<sup>25</sup> and computing the provenance of the output as a function of the executed code path and data dependencies. Even if this method requires no modification of the application, the trade-off is a significant increase in space and time overhead. A low-overhead alternative would be to modify the application to explicitly disclose relevant provenance using an API such as CPL,<sup>17</sup> but this requires additional effort from the developer, as we discuss later.

Granularity is not the only aspect that users need to think about when determining their requirements for a provenance system. It is just as important to know in which layer the provenance collection takes place.

*Layering.* Provenance metadata can be captured at multiple layers in the stack (that is, for the application, middleware (runtime/libraries), operating system, and/or in hardware). Capturing provenance across multiple layers provides users with the ability to reason about their data and processes at different *levels of abstraction*, with each layer providing a different view on the same set of events happening in the system.

For example, consider copying rows between two tables in a spreadsheet and saving the result. A system that collects provenance at the operating-system layer will observe a number of I/O operations to/from the file. The notions of tables and rows, however, are known only to the application, and dependencies among them cannot be inferred from the metadata collected by lower layers. If querying for such relationships is needed, provenance must be captured in the application layer as well.

*Cooperation between layers.* When requiring provenance capture at multiple layers, a practitioner could choose a different (specialized) provenance system for each layer in the stack or a single provenance system that was designed to span capture across multiple layers.

In both cases, multiple provenance-aware components must cooperate by communicating different metadata between layers. This can be achieved either by adhering to a common provenance data model, such as Open Provenance Model (OPM)<sup>21</sup> or Provenance

Data Model (PROV-DM<sup>20</sup>), or by providing a universal API and allowing each component to both accept and generate provenance using it. PASSv2 provides a disclosed-provenance API (DPA-PI) that can be used for this purpose.

A second issue exists, however. Merely collecting metadata at different layers will result in islands of provenance, unrelated to each other. For an actual *mapping* of provenance objects between layers, all entities describing the same event must be grouped—for example, by tagging them with a unique identifier.

SPADEv2, for example, uses a multisource fusion filter (with process ID as a tag) to combine provenance data from multiple sources describing the same event and working at the same level of abstraction. When provenance is reported at different levels of abstraction, SPADEV2 uses a cross-layer composition filter that has the same purpose.

*Data versioning.* Provenance collection in a given layer typically involves capturing the chain of events performed by the application on a given piece of data, though this does not necessarily require the system to capture multiple versions of data as it is being transformed. Assume a user edits a file using a text editor on a PASS-enabled system. The provenance metadata saved by PASS can provide information such as the program used to edit, number of bytes written to the file, and so on, but it is not possible to revert the file to a previous state or know what the actual data changes were. In cases where the current contents of the file depend on values in previous versions, provenance systems need to store versions of data besides events in order to assure full verifiability.

Because of this, provenance systems such as Burrito<sup>14</sup> not only track system call-level events, but also run on top of a versioning file system. Other systems such as Lipstick and RAMP do not require versioning as they run on top of append-only file systems (all versions are implicitly stored).

Versioning can prove expensive when done for certain layers in the stack (such as for hardware registers) but in other cases it might simplify the capture of provenance. This is the case in the application layer, where data


versions are implicitly stored as undo/redo actions. Most GUI applications provide this functionality by default, and intercepting the undo stack has been shown to be a viable method for automatically inferring provenance.<sup>8</sup>

**Integrating provenance into existing workflows.** The effort needed for integrating a new piece of technology within an existing workflow is an important practical criterion when choosing a provenance system. This measures how much the provenance system will intrude on the user's normal working practices, and a cost-benefit analysis should be made depending on the use case.


Some systems impose larger upfront expenses because of how they collect metadata. For example, they require developers to attest explicitly to provenance information through APIs that record annotations about the actions being executed. An example of this is DPAPI, which offers augmented read and write calls to which one can pass data indicating the meaning of the read or write call being made. The result is an increase in the development effort, as code must be updated to call the new API. All future code changes must also keep the provenance-related code in sync, and failing to do so will most likely cause invalid metadata to be captured.

Similarly, systems such as ZOOM or VisTrails require you to declare the entire workflow in advance and can track only those dependencies that run on top of their execution engines. Subsequent work must be done within the same system if dependency links need to be maintained. As a group, the literature refers to these as *disclosed provenance* systems, and they are recognized for their ability to offer improved semantic descriptions of provenance. The trustworthiness of the provenance captured in this way, however, is a concern when running in untrusted environments.

Other provenance systems aim to reduce the overhead imposed on the user. These tend to take a different approach by observing the users' applications, recording information about how these applications interact with each other and the rest of the operating system, and inferring provenance based on it. They are often referred to



**Using a provenance system is only as useful as the questions that one can answer based on the collected metadata.**



as *observed provenance* systems. Examples are systems such as PASS that intercept system calls made by a program, or others such as SPADE that can hook into the audit subsystem in the Linux kernel to observe the program's actions. They tend to have the lowest intrusiveness. Often, once the system is installed, a user can proceed as usual while provenance is captured for all the executed operations. Observed provenance systems have their own shortcomings, however, mostly because of the loss of semantic information when treating each process as a black box.

**How do you answer questions using provenance?** Using a provenance system is only as useful as the questions that one can answer based on the collected metadata. Querying, however, is recognized as a challenging problem: users often want to query over a broad range of information, or they ask questions that the designers of a provenance system did not anticipate; depending on the granularity of capture, the system either might have insufficient data to respond to a query, or it might produce so much data that it is difficult to explore and understand it. From the research performed to date in the field, two core paradigms of querying have emerged, and a smaller number of systems use a hybrid of both approaches.

*Exploratory.* The first major paradigm is the exploratory query, which takes advantage of the human ability to spot patterns. This is important when users do not have an exact idea of what metadata they might want to retrieve. Exploratory systems are usually characterized by presenting the user with a visual representation of the provenance graph and providing tools to explore it without succumbing to information overload. This is a notably hard problem, given that even small provenance graphs can easily contain thousands of nodes. A number of the approaches involve either exploring subgraphs based on contextual filtering (such as InProv<sup>4</sup>) or using intelligent clustering methods. An example of the latter is the PASS Map Orbiter<sup>18</sup> viewer, which implements an algorithm for dynamically summarizing nodes, allowing the user to expand and contract areas of detail while browsing.



*Directed.* The second major paradigm is the directed query, an approach more closely linked to the classic field of database query. It requires the user to express questions about the provenance of data as queries in a language that is often a specialized extension of SQL or a path query language.

This is effective if the user knows precisely what information is required, but unlike exploratory methods, the directed query approach does not facilitate discovery of new insights about the provenance graph.

One example of the directed approach is vtPQL,<sup>26</sup> used in the VisTrails system. The language is designed to enable the user to express provenance queries about three different aspects of the workflow: the execution log, the abstract workflow representation, and the evolution of the workflow in time.


The user can specify restrictions on all of these spaces simultaneously—for example, restricting the execution logs to a particular day, highlighting a single workflow module, and choosing a particular version of the workflow. This is helpful, as it allows the user to think in terms of orthogonal querying concerns.

*Hybrid.* Some systems use a hybrid of the two paradigms. For example, the ZOOM system<sup>2</sup> starts from a user-provided “declaration of interest” to derive a contextually appropriate minimal form of the provenance graph. The heart of the system is an algorithm that summarizes “irrelevant” parts of the graph in ways that maintain their semantics. The user needs only to provide the list of the modules in the workflow definition that are of interest and is then allowed to browse the provenance graph without being distracted by unimportant pieces of information.


### Understanding Overhead

As with any computational functionality, provenance capture has associated temporal and spatial costs. Given that provenance support is likely to be an *additional* consideration to the primary function of the system, leveraged only when the lineage properties of the data are required, it is imperative to minimize the overhead.

General-purpose provenance systems typically capture either (disclosed) evolutions of a given workflow



**While intuitively it may appear that provenance capture at the operation level is prohibitively expensive from a temporal perspective, reported results show this is not the case.**



or (observed) low-level operations carried out by executing processes. Broadly speaking, the time and space overhead for capturing the provenance of workflow evolution is proportional to both the number of changes in the workflow *and* the number of times a workflow is executed. In comparison, the provenance overhead of capturing an execution log is proportional to the number of recordable operations executed.

*Time overhead.* In practice, the provenance-capture cost of workflow systems (and, by extension, other disclosed provenance systems) is minuscule because of their limited approach to collecting running process information. Both ZOOM and VisTrails, for example, report an approximately 1% increase in execution time.<sup>2,10</sup>

For systems that record process execution, provenance capture costs are a function of the costs of intercepting and recording observable operations. While intuitively it may appear that provenance capture at the operation level is prohibitively expensive from a temporal perspective, reported results show this is not the case. Kernel-based system-call interception mechanisms such as in PASSv2 have a 1% to 23% overhead on workloads representative of real-world applications.<sup>22,23</sup> Similarly, SPADEv2, which uses kernel-auditing infrastructure for provenance capture, reports less than a 10% overhead on Windows, Linux, and OS X for production Apache runs.<sup>12</sup>

For I/O-heavy workloads, however, provenance capture may impose larger runtime overheads. PASS, for example, reports up to a 230% overhead on small file benchmarks,<sup>23</sup> even though the absolute increase in execution times remains small.

The interception mechanism can also significantly influence provenance-capture overhead in this regard. SPADEv2, for example, supports operation interception via the kernel-auditing mechanisms on OS X, while on Windows it requires a file-system filter driver that relays operations to the provenance collector. As a consequence, provenance-enabled Apache builds are 50% slower on Windows but only 5% slower on OS X.

The temporal cost of recording operations may also be of potential con-

cern where provenance is being recorded at an extremely fine-grained level. In such situations it is common for the cost of provenance capture to equal or exceed the cost of the recorded operation, leading to slowdowns exceeding 100%. For example, in the Lipstick system operator-level provenance is reported to lead to a slowdown of two to three times,<sup>1</sup> while in the RAMP system, where provenance is collected at the tuple level by propagating tags through a MapReduce workflow, it is common to observe a temporal overhead of up to 75%.<sup>24</sup>

**Spatial overhead.** Similar to temporal overhead, the spatial overhead of systems recording process execution is a function of the amount of data per operation and the number of recorded operations. In the set of studied systems only half (SPROV, Burrito, Lipstick, and RAMP) are capable of recording data changes.

While the actual overhead of any workload is sensitive to multiple factors, here are two reported data points for illustrative purposes:

- The general-purpose PASSv2 system requires, on average, approximately 20% additional space overhead (compared with the original output size) to log all the operations for a workload representative of real-world applications.<sup>22</sup>

- The Burrito system, running on a real user workload, required 800MB for provenance storage and 2GB for file versions over a two-month period.<sup>14</sup>

These results indicate that storage overhead should not be prohibitive for most cases.

**Overhead trade-offs.** Generally speaking, there is a direct trade-off between capture granularity and provenance overhead. SPADEV2, for example, allows users to capture information at the function call or an application-defined level at the cost of increased temporal and spatial capture overhead. Similarly, SPROV allows users to specify modifications in higher-level semantics (for example, “new section added to file”) at the cost of reduced per-operation observability.

For users to adopt the most suitable system for their needs, it may be useful for them to predetermine what provenance information will be required to answer queries and at what granular-

ity this information will be sufficient, mapping it to the appropriate system.

Most systems also delay provenance construction in order to minimize capture overhead. PASSv2, for example, captures raw operation records, converting them to their final representation via an asynchronous user-space daemon.<sup>22</sup> SPADEV2 uses separate provenance collection threads to extract, filter, and commit operations to the provenance log. Other systems delay provenance collection to query time to avoid wasting resources computing provenance that will never be accessed. For example, Lipstick carries out provenance construction only when a query is made.<sup>1</sup> This delayed provenance-construction property is present in some workflow systems as well. ZOOM, for example, will compute some of the provenance at query time, based on the current user view. Depending on the required cardinality, timeliness, and complexity of provenance queries, deciding on those trade-offs may considerably improve overhead.

**Security issues.** It is imperative for provenance data to be secured against unauthorized access and to not leak any information about the data against which it is collected.<sup>5</sup> Fundamentally, this requires provenance to be managed under different access policies than those of the data. Doing so allows the user flexibility over the disclosure of provenance information. For example, one might make provenance inaccessible to people outside an organization, as it would reveal proprietary workflow or processes. The final data result, however, might be freely available to anyone.

Formally, the security aspects of provenance are defined as its *confidentiality* (only authorized parties can read it) and its *integrity* (it cannot be forged or altered). Both properties are considered essential for performing integrity, validation, and consistency checks on data.

Two solutions to the problem of providing secure provenance have been put forth. The first leverages the concept of reference monitors: Patrick McDaniel et al. discuss a secure system for end-to-end provenance based on the principle of a host-based tamper-proof provenance monitor

that mirrors the well-known reference monitor concept for the enforcement of security policies.<sup>19</sup> The presence of the reference monitor means the security of provenance collection does not have to rely on the integrity of other system components such as the kernel. While this solution is feasible, there is no known practical implementation to date.

The second solution is based on provenance chains<sup>11,16</sup> where processes that generate provenance must attest to the information added in an encrypted, nonmodifiable, and nonrepudiable manner. Guaranteeing these three properties ensures all collected provenance can remain confidential and keep its integrity. Of the systems included in this article, SPROV<sup>16</sup> is a practical implementation of provenance chains. It primarily provides confidentiality and integrity guarantees for file modifications.

SPROV leverages a number of concepts in cryptography to fulfill the security requirements: confidentiality is maintained by encrypting the metadata describing each change; record integrity is maintained by checksumming records; and attestation is supported by signing records with the public key of the creating user.

In addition to the key concepts of confidentiality and integrity, SPROV provides a number of useful features that may be of interest to the practitioner (and a consideration for future secure provenance systems): through the use of cryptographic commitments,<sup>3</sup> SPROV enables selective exposure of records to third parties; by employing broadcast encryption,<sup>15</sup> it supports selective access control for multiple auditors without requiring a corresponding proportional increase in the number of keys; finally, threshold encryption<sup>27</sup> is supported, enabling separation-of-duty scenarios in which the decryption of records requires participation from at least one auditor in a number of distinct groups.

SPROV has no mechanism for preventing unauthorized reads, relying instead on the encryption of records to prevent unauthorized access. It is, however, the only system of those included here that provides any provenance confidentiality and integrity guarantees.



While all systems acknowledge the security of provenance is a fundamental concern, the rest rely on existing access-control mechanisms such as SQL grant privileges and file permissions to ensure security.

## Research Challenges And Opportunities

Contrasting the initial use cases and what can actually be achieved with current provenance systems makes it clear that research is needed in a number of areas.

**Querying and visualization.** Despite the research carried out so far toward querying and visualizing provenance, these are still challenging problems. It remains to be seen how existing knowledge about graph exploration and visualizations could be applied, or whether totally different representations are required.

**Computing with provenance.** Moving beyond human queries, provenance should be made available to applications, allowing automated validation of inputs, limiting error propagation, or self-diagnosing changes in output quality or system behavior.

**Distributed systems.** There have been attempts to extend provenance to networked systems, but problems related to heterogeneity (not all nodes being provenance aware), scalability, long-term collection, and storage remain to be solved.


**Security and privacy.** Collecting provenance has implications on data security and privacy, but most implementations have not considered untrusted environments or adversarial workloads.

## Conclusion

The computing power and storage capacities available today allow large quantities of data to be processed in complex ways. Sometimes the transformations applied are not directly controlled by or even known to developers (multiple layers of abstraction, learning algorithms). Therefore, a lot of information about a result is lost when no provenance is recorded, making it harder to assess quality or reproducibility. Computing is becoming pervasive, and the need for guarantees about it being dependable will only aggravate those problems; treating provenance as a first-class citizen

in data processing represents a possible solution.

## Acknowledgments

We would like to thank George Courlouris for his feedback and our reviewers for their constructive comments and suggestions. 

## Related articles on queue.acm.org

### Provenance in Sensor Data Management

Zachary Hensley, Jibonananda Sanyal, Joshua New

<http://queue.acm.org/detail.cfm?id=2574836>

### CTO Roundtable: Storage

Mache Creeger

<http://queue.acm.org/detail.cfm?id=1483110>

### Better Scripts, Better Games

Walker White, Christoph Koch, Johannes Gehrke, Alan Demers

<http://queue.acm.org/detail.cfm?id=1483106>

## References

1. Amsterdamer, Y. et al. Putting lipstick on pig: Enabling database-style workflow provenance. In *Proceedings of the VLDB Endowment* 5, 4 (2011), 346–357.
2. Biton, O., Cohen-Boulakia, S. and Davidson, S.B. ZOOM\*UserViews: Querying relevant provenance in workflow systems. In *Proceedings of the 33rd International Conference on Very Large Databases*, (2007), 366–369.
3. Blum, M. Coin flipping by telephone: a protocol for solving impossible problems. In *Advances in Cryptology—A Report on CRYPTO '81*, (1982).
4. Borkin, M.A. et al. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2476–2485.
5. Braun, U., Shinnar, A., Seltzer, M. 2008. Securing provenance. In *Proceedings of the 3rd Usenix Workshop on Hot Topics in Security*, (2008), 1–5.
6. Braun, U. et al. Issues in automatic provenance collection. In *Proceedings of the International Conference on Provenance and Annotation of Data*, (2006), 171–183.
7. Buneman, P., Khanna, S. and Tan, W.C. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*, (2002), 316–330.
8. Callahan, S.P. et al. Towards process provenance for existing applications. In *Proceedings of the 2nd International Provenance and Annotation Workshop*, (2008), 120–127.
9. Cui, Y., Widom, J. and Wiener, J.L. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems* 25, 2 (2000), 179–227.
10. Freire, J. et al. Managing rapidly evolving scientific workflows. In *Proceedings of the International Conference on Provenance and Annotation of Data*, (2006), 10–18.
11. Gates, C. and Bishop, M. One of these records is not like the others. In *Proceedings of the 3rd Usenix Workshop on the Theory and Practice of Provenance*, (2011).
12. Gehani, A. and Tariq, D. SPADE: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, (2012), 101–120.
13. Green, T. J., Karvounarakis, G., Tannen, V. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, (2007), 31–40.
14. Guo, P.J., and Seltzer, M. Burrito: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th Usenix Conference on Theory and Practice of Provenance*, (2012) 7–7.
15. Halevy, D. and Shamir, A. The LSD broadcast encryption scheme. In *Advances in Cryptology*, (2002), 47–60.
16. Hasan, R., Sion, R. and Winslett, M. The case of the fake Picasso: preventing history forgery with secure

- provenance. In *Proceedings of the 7th Conference on File and Storage Technologies*, (2009), 1–14.
17. Macko, P. and Seltzer, M. A general-purpose provenance library. In *Proceedings of the 4th Usenix Conference on Theory and Practice of Provenance*, (2012), 6–6.
18. Macko, P. and Seltzer, M. Provenance Map Orbiter: interactive exploration of large provenance graphs. In *Proceedings of the 3rd Conference on Theory and Practice of Provenance*, (2011)
19. McDaniel, P. et al. Towards a secure and efficient system for end-to-end provenance. In *Proceedings of the 2nd Conference on Theory and Practice of Provenance*, (2010), 2–2.
20. Moreau, L. and Missier, P. PROV-DM: The PROV Data Model. Technical Report. World Wide Web Consortium, 2013.
21. Moreau, L., et al. The Open Provenance Model Core Specification (V1.1). *Future Generations Computer Systems* 27, 6 (2011), 743–756.
22. Muniswamy-Reddy, K.-K., et al. Layering in provenance systems. In *Proceedings of the Usenix Annual Technical Conference*, 2009.
23. Muniswamy-Reddy, K.-K., et al. Provenance-aware storage systems. In *Proceedings of the Usenix Annual Technical Conference*, (2006), 43–56.
24. Park, H., Ikeda, R. and Widom, J. RAMP: A system for capturing and tracing provenance in MapReduce workflows. In *Proceedings of the 37th International Conference on Very Large Databases*, (2011).
25. Saxena, P., Sekar, R. and Puranik, V. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, (2008), 74–83.
26. Scheidegger, C., et al. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience* 20, 5 (2008), 473–483.
27. Shamir, A. 1979. How to share a secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613.
28. Widom, J. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, 2004.

**Lucian Carata** (lucian.carata@cl.cam.ac.uk) is a Ph.D. student in the Computer Laboratory, University of Cambridge. His research focuses on the next-generation disclosed provenance systems, with the aim of understanding and controlling the behavior of complex systems.

**Sherif Akoush** (sherif.akoush@cl.cam.ac.uk) is a Research Associate at University of Cambridge Computer Laboratory. He is exploring provenance in “Big Data” systems and its applications.

**Nikilesh Balakrishnan** (nikilesh.balakrishnan@cl.cam.ac.uk) is a Research Assistant in the Computer Laboratory, University of Cambridge. His research focuses on building general-purpose provenance systems with emphasis on usability and wide adoption among the user community.

**Thomas Bytheway** (thomas.bytheway@cl.cam.ac.uk) is a Research Assistant in the Computer Laboratory, University of Cambridge. His research interests are in building general-purpose provenance systems and exploring querying and visualization techniques.

**Ripduman Sohan** (ripduman.sohan@cl.cam.ac.uk) is a Senior Research Associate and Co-PI of the Fabric For Reproducible Computation (FRESCO) project in the Computer Laboratory, University of Cambridge. He previously worked on storage, virtualization, networking and energy-efficient computing.

**Margo Seltzer** (margo@eecs.harvard.edu) is the Herchel Smith Professor of Computer Science in Harvard's School of Engineering and Applied Sciences. She was co-founder and CTO of Sleepycat Software, the makers of Berkeley DB, until Oracle acquired Sleepycat in 2006. She is now an architect in Oracle Labs.

**Andy Hopper** (ah12@cam.ac.uk) is Professor of Computer Technology at the University of Cambridge, Head of Department of the Computer Laboratory, and elected member of the University Council. His research interests include computer networking, pervasive and sensor-driven computing, and using computers to ensure the sustainability of the planet.

Copyright held by Owner/Author(s). Publication rights licensed to ACM. \$15.00.