A primer on provenance

Lucian Carata
Sherif Akoush
Nikilesh Balakrishnan
Thomas Bytheway
Ripduman Sohan
Margo Seltzer
Andy Hopper

March 12, 2019

1 Introduction

Assessing the quality or validity of a piece of data is not usually done in isolation. Instead, we typically examine the context in which data appears, try to determine its original sources or review the process through which it was created. However, this is not straightforward when dealing with digital data: 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, it becomes harder to keep track of how different sources and transformations are related to the final result. This inevitably constrains our ability to answer questions regarding that result's *history*, like: 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 in order to answer those (or similar) questions is called *provenance*[1] and refers to a graph describing the relationships between all the elements (sources, processing steps, contextual information and dependencies) that contributed to the existence of a piece of data.

We present current research done in this field from a practical perspective, discussing existing systems and the fundamental concepts required to understand how to use them in applications today, but also looking at future challenges and opportunities.

A number of use cases are representative for understanding 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, being able to point at a piece of data (re-

[1] or lineage; we will consider the two terms equivalent in this article.

search result, 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 are used 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 in figuring out the set of assumptions behind the statistics reported to a board of directors, or in determining what mortgages were part of a traded security.

Who is using this data?

Instead of tracking a result back to its sources, we 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 for keeping track of code still using deprecated/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 only 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 get 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 arising when running complex software stacks.

In cases where it is possible to differentiate between a correct and an 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 for explaining the differences between the two.

2 Provenance systems

Together, the three use cases provide an overview of the ideal provenance application space, but do not describe the technical details involved in making

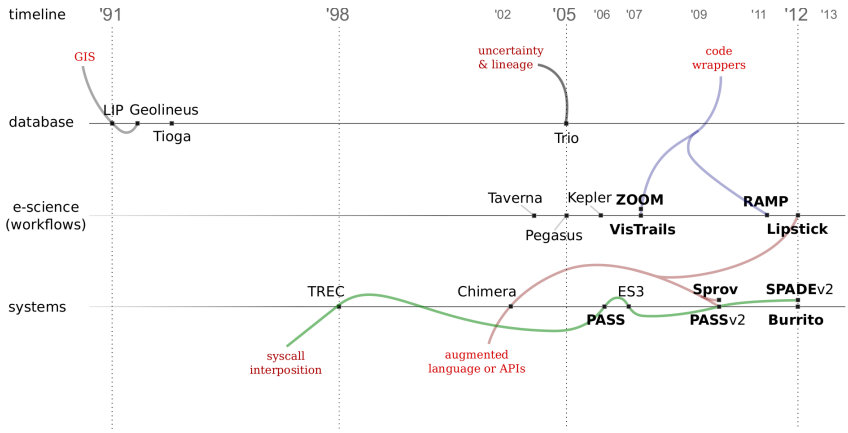

Figure 1: A timeline of provenance systems. The ones reviewed in this paper are in bold

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 between related components and making it accessible to user queries.

In many ways, you might already be running a very specialized version of such a system today: all auditing, tracing frameworks or change tracking solutions will collect some form of provenance, even though they might not identify it as such. The advantage of thinking about provenance as a standalone concept is the ability to use this metadata in a principled way, allowing result verifiability and complex historic queries irrespective of the underlying mechanisms used to collect it and across applications/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 [9]. Because of the well defined relational model, it has proven possible to both derive precise provenance information from queries [7] and to develop formalisms which allow its concise representation [13]. This has been further extended in systems such as Trio [28], allowing records to have an associated uncertainty and being able to propagate it across multiple queries by using provenance.

In contrast, capturing provenance for applications performing arbitrary computations (with possible side effects and not restricted to a particular 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).

Figure 1 presents a general timeline of systems and we discuss the characteristics of eight of them, each representative for a larger class of solutions in the design space:

OS level: PASS [23, 22] and SPADE [12] investigate capturing provenance by observing application events such as process creation or IO. Those are then used for inferring dependencies between different pieces of data. Subsequent versions of those systems have also added the ability to integrate with special user space

libraries in order to obtain more information about application behavior.

Workflows: Vistrails [26] and Zoom [2] are workflow management systems with the ability to track provenance for the execution of various workflows and (in case of Vistrails) for the evolution of the workflows themselves.

Application level: Burrito [14] tracks user space events, while also supporting additional user-provided annotations. Sprov [16] focuses on the security of provenance, and provides a thin wrapper around the standard C IO library. A newer version is capable of using provenance captured by other systems, such as pass.

Big data: Lipstick [1] and Ramp [24] both tackle the problem of tracking provenance in big data scenarios (Map-Reduce jobs).

It is the properties of those systems that are important for understanding what can be recorded and with what trade-offs, overheads and security implications.

3 Provenance system properties

From the perspective of someone who wants to start using provenance for his/her own applications and data, there are a number of aspects that are particularly relevant when looking at a provenance system:

- **What can it capture?** Understanding what metadata is relevant for a series of data transformations and how the captured information enables and limits the type of questions that could be asked later.
- **Integration effort:** Integrating the system within existing data processing scenarios might involve the need to run on a special OS kernel, make changes to the runtime environment or link applications with provenance libraries.
- **How to answer queries using provenance?** The way one might explore and ask questions based on the captured metadata is important for understanding how provenance can be used to satisfy the various use cases.
- **Understanding overheads**: Given one use case, it is often essential to grasp whether the overheads imposed by running the provenance system are acceptable, and to be able to predict how those overheads will scale as a function of the number of data dependencies and transformation steps executed.
- **Security issues:** provenance metadata will often require different access controls from those of the data itself, and it is important to understand the security concerns raised by the use of a particular provenance system.

We categorize the properties of the systems we have picked as representative according to the above features, referring back to the motivating use cases as required. For an orthogonal view, a per-system summary of properties can be consulted in Table 1.

3.1 What can it capture?

The metadata captured by provenance systems typically relates the state of digital entities (files, tables, programs, network connections etc.) at different stages in their lifetime 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*.

3.1.1 Granularity

The *granularity* of capture 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 discover the dependency between the simulation program and the configuration filename. However, the scientist is interested in understanding the relationships between the simulation results and individual parameters in the file, which requires capture at sub-file 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 relationships between individual rows or cells. Of course, multiple granularities can be considered at the same time.

Systems such as PASS [6], 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.

However, the *noise levels* in the collected data 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 [26] 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. However, such systems are also inherently limited to *only* recording data transformations that were part of the defined workflow.

The n-by-m problem Independent of the system that is chosen, it may not be possible to accurately determine the dependencies between input and output data. 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's 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 even finer granularities. This can be done using binary instrumentation techniques [25] 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 tradeoff is a significant increase in space and time overheads. A low-overhead alternative would be to modify the application to explicitly disclose relevant provenance using an API such as CPL [17], but this requires additional effort from the developer, as further discussed in section 3.2.

Granularity is not the only aspect a user needs to think about when determining his/her requirements for a provenance system. It is just as important to know in which layer the provenance collection takes place.

3.1.2 Layering

Provenance metadata can be captured at multiple layers in the stack i.e. 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 collecting provenance at the OS layer will observe a number of IO operations to/from the file. However, the notions of "tables" and "rows" are only known to the application, and dependencies amongst 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 pieces of metadata between layers. This can be achieved either by adhering to a common provenance data model (such as OPM [21] or PROV-DM [20]) or by providing an universal API and allowing each component to both accept and generate provenance using it. PASSv2 provides a a disclosed provenance API (DPAPI) that can be used for this purpose.

However, a second issue exists. 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 an unique identifier.

SPADEv2 for instance uses a multi-source 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. However, this does not necessarily require the system to capture multiple versions of data as it is being transformed. Assume that 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 etc. But it is not possible to revert the file back 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 processing events in order to assure verifiability. Because of this, provenance systems such as Burrito [14] not only track system call level events, but are also running 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. For example, deciding to version data in the hardware layer (versioning the values or a register) would create large amounts of data, and should be preceded by an evaluation of actual benefits. In other cases, versioning can actually improve the collection of provenance. This is the case in application layer, where a user can undo/redo actions. Most GUI applications provide this functionality by default and intercepting the undo stack has been shown to be viable method for automatically inferring provenance [8].

3.2 Integrating provenance into existing workflows

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

Some systems impose larger upfront expenses due to how they collect metadata. For example, they ask the application to explicitly attest to provenance information, as is the case with APIs that allow you to supply annotations about the actions being executed. An example of this is the PASSv2 DPAPI, which offers augmented read and write calls to which one can pass data indicating the meaning of the read or write call that is being made. The end 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 ask you to declare the entire workflow in advance and can only track 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 those as *disclosed provenance* systems, and they are recognized for their ability to offer improved semantic descriptions of provenance. However, the trustworthiness of the provenance captured in this way 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 OS and inferring provenance based on it. They are often referred to as *observed provenance* systems. Systems such as PASS that intercept system calls made by a program or others such as SPADE that can hook into the audit sub-system in the Linux kernel to observe the program's actions are examples of this type of system. They tend to have the lowest intrusiveness. Often once the system is installed a user can proceed as usual while having provenance captured for all of the operations they perform. However, observed provenance systems have their own shortcomings, mostly due to the loss of semantic information when treating each process as a black box.

3.3 How do I answer questions using provenance?

Using a provenance system is only as useful as the questions that one can answer based on the collected metadata. However, querying 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, there might be either insufficient data to respond to a query, or the system might produce so much data that it is difficult 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 some hybrid of both approaches.

Exploratory

The first major paradigm is exploratory query, which takes advantage of the human ability to spot patterns. This is important when users don't 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 giving them tools to better 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 taken involve either exploring subgraphs based on contextual filtering (such as InProv [4]) or intelligent clustering methods. An example of the latter is the PASS Map Orbiter [18] viewer, which implements an algorithm for dynamically summarizing nodes allowing you to expand and contract areas of detail while browsing.

Directed

The second major paradigm is 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 of a path query language.

The approach is effective if the users know precisely what information they require, but unlike exploratory methods it does not facilitate discovery of new insights about the provenance graph.

vtPQL [26] is an example of the directed approach 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 querying system allows a user to specify restrictions on all three 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 [2] starts from a user-provided 'declaration of interest' to derive a contextually appropriate minimal from of the provenance graph. The heart of the system is an algorithm that summarizes 'irrelevant' parts of the graph in ways that maintains their semantics. The user only needs 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.

3.4 Understanding overheads

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, only leveraged when the lineage properties of the data are required, it is imperative to minimize the overheads.

General purpose provenance systems typically capture either (disclosed) evolutions of a given workflow or (observed) low-level operations carried out by executing processes. Broadly speaking, the time and space overheads 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 overheads In practice the provenance capture cost of workflow systems (and by extension of other disclosed provenance systems) is minuscule due to their limited approach to collecting running process information. Both Zoom and Vistrails, for example, report an approximately 1% increase in execution time [2, 10].

For systems that record process execution, provenance capture costs are a function of the cost 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 that this is not the case. Kernel based system call interception mechanisms such as in PASSv2 have a 1–23% overhead on workloads representative of real-world applications [23, 22]. Similarly SPADEv2, which utilises kernel auditing infrastructure for provenance capture, reports $< 10\%$ overhead on Windows, Linux and OS X for production Apache runs [12].

For I/O heavy workloads, however, provenance capture may impose larger runtime overheads. PASSv2 for example, reports up to 230% overhead on small file benchmarks [22], but 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 concern 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 of over 100%. For example, in the Lipstick system it is reported that operator-level provenance leads to a slowdown of 2-3x [1] while in the RAMP system, where provenance is collected at the tuple level by propagating tags through a Map-Reduce workflow it is common to observe temporal overheads of up to 75% [24].

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

While the actual overheads of any workload are sensitive to multiple factors, we provide two reported data points for illustrative purposes: (i) the general purpose PASSv2 system requires, on average approximately 20% additional space overhead (as compared to the original output size) to log all the operations for workloads representative of real-world applications [22] and (ii) the BURRITO system, running on a real user workload, required 800MB for provenance storage and 2GB for file versions over a two month period [14]. These results lead us to believe storage overheads should not be prohibitive for most common cases.

Overhead trade-offs Generally speaking, there is a direct relationship between finer capture granularities and provenance overhead. Some systems leverage this relationship to trade-off granularity of capture for provenance information. 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 (e.g. "new section added to file") at the cost of reduced per-operation observability.

In order 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 provenance queries and at what granularity this information will be sufficient, mapping it to the appropriate system.

Most systems also delay provenance construction in order to minimize capture overheads. PASSv2 for example captures raw operation records, converting them to their final representation via an asynchronous user-space daemon [22]. SPADEv2 uses separate provenance collection threads to extract, filter and commit operations to the provenance log. Other systems delay provenance collection to query time in order to avoid wasting resources computing provenance that will never be accessed. For example, Lipstick only carries out provenance construction when a query is made [1]. 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 overheads.

3.5 Security issues

The security of provenance data is another fundamental issue. It is imperative for provenance data to be secured against unauthorized access and not leak any information about the data against which it is collected [5]. Fundamentally, this concern requires provenance data to be managed under separate access policies than the data it represents. 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 workflows or processes. However, the final data result might be freely available to anyone.

Formally, we define the security aspects of provenance as its *confidentiality*, i.e., that only authorized parties can read it and its *integrity*, i.e., that it cannot be forged or altered. We consider provenance that has both properties as essential for performing integrity, validation and consistency checks on data.

Two solutions address the problem of providing secure provenance. The first leverages the concept of reference monitors: McDaniel discusses 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 [19]. The presence of the reference monitor means that the security of provenance collection doesn't have to rely on the integrity of other system components such as the kernel. While this solution is feasible we have not encountered a practical implementation to-date.

The second is based on provenance chains [11, 16] where processes that generate provenance *must* attest to the information added in an encrypted, non-modifiable and non-repudiable manner. By guaranteeing these three properties we ensure that all collected provenance has the confidentiality and integrity properties. In our set of studied systems, SPROV [16] 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 of cryptographic commitments [3], SPROV enables selective exposure of records to third parties; by employing broadcast encryption [15] SPROV supports selective access control for multiple auditors without requiring a corresponding proportional increase in the number of keys. Finally, threshold encryption [27] 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 fact that records are encrypted to prevent unauthorized access. However it is the only system in our studied set that provides any provenance confidentiality and integrity guarantees. While all systems acknowledge that 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.

4 Research challenges and opportunities

Contrasting the initial use cases and what can actually be achieved with current provenance systems, it becomes clear there are a number areas in which future research is needed:

Querying and visualization: despite the research done so far in terms of querying, exploring and visualizing provenance, it still is a challenging problem and 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: going beyond human queries, provenance could be made available to applications, allowing automated processes of validating inputs, limiting error propagation or self-diagnosing changes in output quality or system behavior.

Distributed systems: there have been attempts of extending provenance to networked systems, but problems related to heterogeneity in distributed systems (where not all nodes are provenance-aware), scaling to a large number of nodes, long-term collection and storage remain to be solved.

Security and privacy: we know that collecting provenance has multiple implications regarding data security and privacy, but more research is needed to understand how applications might enable provenance questions like "who is using this data?" in untrusted environments.

5 Conclusion

The increasing computing power accessible to computer engineers and scientists alike has given us the ability to process large quantities of data, possibly using long chains of complex transformations.

The software design strategy of solving problems using multiple layers of abstraction limits some of this perceived complexity. However, this also implies that the computed results might depend on things we know nothing about (leaky abstractions).

Even ignoring abstraction, a lot of information about a result is lost when we fail to store provenance, which might mean it's impossible to asses its quality, reproduce or improve on it.

Integrating systems that collect provenance within normal processing workflows sets the stage for a better understanding of data dependencies, propagation of errors and trustworthiness. Those issues can only become more important as computing becomes pervasive, so it is vital to move towards a world where provenance is considered a first class citizen.

References

[1] Y. Amsterdamer et al. "Putting lipstick on pig: enabling database-style workflow provenance". In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 346–357.

[2] O. Biton, S. Cohen-Boulakia, and S. B. Davidson. "Zoom* userviews: Querying relevant provenance in workflow systems". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 1366–1369.

[3] M. Blum. "Coin flipping by telephone: A protocol for solving impossible problems". In: *Advances in Cryptology-A Report on CRYPTO'81* (1982).

[4] M. A. Borkin et al. "Evaluation of Filesystem Provenance Visualization Tools." In: *IEEE Trans. Vis. Comput. Graph.* 19.12 (2013), pp. 2476–2485.

[5] U. Braun, A. Shinnar, and M. Seltzer. "Securing Provenance". In: *The 3rd USENIX Workshop on Hot Topics in Security*. USENIX HotSec. San Jose, CA: USENIX Association, 2008, pp. 1–5.

[6] U. Braun et al. "Issues in automatic provenance collection". In: *In Proc. IPAW'06, volume 4145 of LNCS*. Springer, 2006, pp. 171–183.

[7] P. Buneman, S. Khanna, and W. C. Tan. "Why and Where: A Characterization of Data Provenance." In: *ICDT*. Vol. 1973. Lecture Notes in Computer Science. Springer, Jan. 3, 2002, pp. 316–330.

[8] S. P. Callahan et al. "Towards process provenance for existing applications". In: *Proceedings of 2nd International Provenance and Annotation Workshop (IPAW)*. 2008, pp. 120–127.

[9] Y. Cui, J. Widom, and J. L. Wiener. "Tracing the lineage of view data in a warehousing environment". In: *ACM Trans. Database Syst.* 25.2 (June 2000), pp. 179–227.

[10] J. Freire et al. "Managing rapidly-evolving scientific workflows". In: *Provenance and Annotation of Data*. Springer, 2006, pp. 10–18.

[11] C. Gates and M. Bishop. "One of These Records Is Not Like the Others". In: *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance*. Berkeley, CA, USA: USENIX Association, 2011.

[12] A. Gehani and D. Tariq. "SPADE: Support for provenance auditing in distributed environments". In: *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc. 2012, pp. 101–120.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. "Provenance semirings". In: *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. Beijing, China: ACM, 2007, pp. 31–40.

[14] P. J. Guo and M. Seltzer. "Burrito: wrapping your lab notebook in computational infrastructure". In: *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*. USENIX Association. 2012, pp. 7–7.

[15] D. Halevy and A. Shamir. "The LSD broadcast encryption scheme". In: *Advances in Cryptology—CRYPTO 2002*. Springer, 2002, pp. 47–60.

[16] R. Hasan, R. Sion, and M. Winslett. "The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance." In: *FAST*. Vol. 9. 2009, pp. 1–14.

[17] P. Macko and M. Seltzer. "A General-purpose Provenance Library". In: *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*. TaPP'12. Boston, MA: USENIX Association, 2012, pp. 6–6.

[18] P. Macko and M. Seltzer. "Provenance map orbiter: Interactive exploration of large provenance graphs". In: *Proceedings of the 3nd conference on Theory and practice of provenance*. TAPP'11. 2011.

[19] P. McDaniel 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*. TAPP'10. San Jose, California: USENIX Association, 2010, pp. 2–2.

[20] L. Moreau and P. Missier. *PROV-DM: The PROV Data Model*. Technical Report. World Wide Web Consortium, 2013.

[21] L. Moreau et al. "The Open Provenance Model Core Specification (V1.1)". In: *Future Gener. Comput. Syst.* 27.6 (June 2011), pp. 743–756.

[22] K.-K. Muniswamy-Reddy et al. "Layering in provenance systems". In: *Proceedings of the 2009 USENIX Annual Technical Conference*. 2009.

[23] K.-K. Muniswamy-Reddy et al. "Provenance-aware storage systems". In: *Proceedings of the 2006 USENIX Annual Technical Conference*. 2006, pp. 43–56.

[24] H. Park, R. Ikeda, and J. Widom. "RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows". In: *37th International Conference on Very Large Data Bases (VLDB)*. Stanford InfoLab, 2011.

[25] P. Saxena, R Sekar, and V. Puranik. "Efficient Fine-grained Binary Instrumentationwith Applications to Taint-tracking". In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '08. Boston, MA, USA: ACM, 2008, pp. 74–83.

[26] C. Scheidegger et al. "Tackling the Provenance Challenge one layer at a time". In: *Concurrency and Computation: Practice and Experience* 20.5 (2008), pp. 473–483.

[27] A. Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[28] J. Widom. *Trio: A System for Integrated Management of Data, Accuracy, and Lineage*. Technical Report 2004-40. Stanford InfoLab, 2004.