Supporting Secure Provenance Update by Keeping "Provenance" of the Provenance

Amril Syalim, Takashi Nishide, Kouichi Sakurai

▶ To cite this version:

Amril Syalim, Takashi Nishide, Kouichi Sakurai. Supporting Secure Provenance Update by Keeping "Provenance" of the Provenance. 1st International Conference on Information and Communication Technology (ICT-EurAsia), Mar 2013, Yogyakarta, Indonesia. pp.363-372, 10.1007/978-3-642-36818-9_40 . hal-01480195

HAL Id: hal-01480195
https://inria.hal.science/hal-01480195

Submitted on 1 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Supporting Secure Provenance Update by Keeping "Provenance" of the Provenance

Amril Syalim[1], Takashi Nishide[2], and Kouichi Sakurai[2]

[1] Fakultas Ilmu Komputer, Universitas Indonesia
amril.syalim@cs.ui.ac.id
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
{nishide,sakurai}@inf.kyushu-u.ac.jp

Abstract. Provenance of data is a documentation of the origin and processes that produce the data. Many researchers argue that the provenance should be immutable: once a provenance is submitted, it should not be changed or updated. A main reason is that the provenance represents the history of data, and the history should not be altered or changed because it represents the fact in the past. Provenance can be represented by a graph, where each node represents the process executed by a party and an edge represents the relationship between two nodes (i.e. a child node uses the outputs of the parent nodes). A method to ensure that the provenance has not been updated is by using signature chain, where the signatures of the parent nodes are recorded in the children nodes so that any changes to the parent nodes will raise inconsistencies between the parent and the children. However, sticking to the requirement that the provenance should be immutable requires unlimited data storage and also we have problems whenever we need to update the provenance for an accidental error. In this paper, we propose a method that allows updates in the signature chain-based secure provenance, while keeping the signature consistent. The main idea is by keeping the "provenance" of the provenance itself, that is the history of update of the provenance, in the form of the signatures of the previous versions of the nodes. We implement the idea by keeping the signatures of the previous version in a signature tree similar to the Merkle-tree, where the a parent node in tree is the aggregate signature of the children. Using this method, the storage requirement to store signatures is always smaller than the number of updates.

Key words: Provenance, Provenance Security, E-science, Data lineage

1 Introduction

Provenance is a documentation of data history. It records the processes that produce the data and relationship between the processes. A simple example of the provenance is the documentation about how to produce a patient record in a hospital [2, 6, 12, 13]. The patient record contains the data about the medical treatment or medical test of the patient which is produced by the physicians and laboratory staffs in the hospital. In this case the processes to produce data are executed by the physicians or the laboratory staffs. A process may have relationships with the other processes, for example: to produce a medical treatment, a physician uses the data produced by another physicians (i.e. medical diagnosis) or data used by a laboratory staffs (i.e. a result of blood or urine test of the patient).

Provenance can be represented by a graph [14, 7, 1, 14, 5], where the nodes represent the processes and the data while the edges represent relationships between processes. An example of the graph model for provenance is the open provenance model which is proposed as a standard for provenance [16, 15]. The open provenance model specification defines some types of the nodes (i.e. actor, process and artifacts), and some types of the relationships between the nodes.

Some of the the important research problems with the provenance are related to security [21, 10–12, 19, 20]. The security problems are caused by malicious update and deletion to the provenance. For example in a hospital, if we allow update to the provenance and the patient record, there can be inconsistencies between provenance and data created by a physician with another physician or laboratory staffs. A physician A may decide a medical treatment based on the diagnosis created by physician B, or medical test produced by a laboratory staff C. If B or C updates/deletes the provenance of diagnosis or medical test, the treatment by A may be incorrect because it is based on the previous versions of the diagnosis or medical tests produced by B or C. If the patient who takes the medical treatment complains about a misconduct by A, A cannot refer to B or C to explain why he/she decides the treatment. In an extreme case, it is possible B or C may maliciously/intentionally try to frame A for a misconduct charge.

1.1 Problem Definition

As a history record, many researchers argue that the provenance should be immutable (no change is allowed in the history records) [5, 18, 15, 16, 8, 7, 17]. However, without being able to update a submitted provenance, the provenance storage is always growing (indefinitely). In this paper, we try to address the problem on how to allow updates/deletes the provenance to save spaces on the storages, while keeping the provenance consistent. The methods are useful in the case the actors who produce the provenance honestly update the provenance for an accidental/unintentional errors and in the case the storage is limited. The main security requirement is: no actor is allowed to exploit the update/deletion features to do a malicious behaviors or avoid responsibilities for the data produced in the past.

1.2 Contributions

The idea of our solution is by keeping the "provenance" of the provenance, where using this method, we allow updates/deletions to the provenance but we keep the history of the previous versions of the provenance in the forms of the signatures of the previous versions. So that, even if an actor has updated the provenance and the data, the actor cannot reject the previous versions of the provenance and data. To save the spaces, we store the signatures in a tree similar to the Merkle-tree and combines the signatures using the aggregate signatures techniques. Using this method, for any number of updates the grow of signatures storage for the previous versions of the provenance and data is much smaller than the normal updates in the provenance (where we should keep all signatures and data of the previous versions).

2 Related Work

2.1 Hash/signature chain, stamping and countering

The integrity scheme to timestamp digital documents using hash/signature chain was first proposed by Haber el al. [9] . It uses a Trusted Timestamping Service (TTS) that issues signed timestamps and also links two timestamps requested consecutively. The TSS links two timestamps by storing the hash value of the first timestamp in the second timestamp. Any changes to the first timestamp can be detected by checking the hash value in the second timestamp. This method is applied by Hasan et al. [12] to a chain model of the provenance where the provenance is modeled as a chain. Aldeco-Perez et al. [1] and Syalim et al. [20, 19] extend the hash/signature chain to the provenance graph model.

To detect the problem of the deletion to a provenance node, Syalim et al. proposed countering provenance [20]. The basic idea is a Trusted Counter Server (TCS) assigns a unique consecutive counter number for each node in the provenance graph, so that any deletion to the nodes can be detected from the missing counter number in the nodes. To detect deletion of the newest node, the TCS should store the latest counter number in a trusted storage.

2.2 Aggregate Signatures

Aggregate signatures is a technique to combine signatures on many different messages into a short signature. Some aggregate signatures have restriction that they can only be verified if there is no duplicate messages or public keys. However, it is possible to develop a scheme that does not have any restriction [3].

Aggregate signatures can be implemented using bilinear maps [4], that is with the requirement of the existence of a mapping between groups for example the map $e : G_1 \times G_2 \to G_T$ where $|G_1| = |G_2| = |G_T|$ with bilinear (for all $u \in G_1, v \in G_2$ and $a, b \in \mathbb{Z}, e(u^a, v^b) = e(u, v)^{ab}$) and non-degenerate ($e(g_1, g_2) \neq 1$) properties. A particular aggregate signature scheme proposed by Boneh et al [4] is as follows:

Key Generation the user picks random secret key $x \overset{R}{\leftarrow} \mathbb{Z}_p$ and computes the public key $v \leftarrow g_2^x$
Signing to sign a message M, compute $h \leftarrow H(M)$, where $h \in G_1$, and the signature $\sigma \leftarrow h^x$.
Aggregation for a set of signatures $\{\sigma_1, \sigma_2, ..., \sigma_k\}$, compute the aggregate signature $\sigma \leftarrow \prod_{i=1}^{k} \sigma_i$.
Aggregate Verification for all users $u_i \in U$ with public keys $v_i \in G_1$ and the original messages M_i, computes $h_i \leftarrow H(M_i)$ and accept if $e(\sigma, g_2) = \prod_{i=1}^{k} e(h_i, v_i)$ holds.

3 Preliminaries

3.1 Definition

Definition 1 (Provenance definition). *Provenance related to the data set* $D = \{d_0, d_1, ..., d_{n-1}\}$ *stored in a database* DB *for* $n > 0$ *is a set of provenance nodes* $P = \{p_0, p_1, ..., p_{n-1}\}$ *and a binary relation* E *recorded in a Provenance Store* PS*. A provenance node* p_i *consists of an identification* PID*, a process description* A*, a process executor* CID*, the list of references to a set of inputs* $\{ref(I_i)\}$*, for* $I \subseteq D$ *and a reference to an output* $ref(d_i)$ *for* $d_i \in D$*.* E *is a binary relation in* $PIDSET$ *where* $PIDSET$ *is a set of* PID *of all provenance nodes.* E *represents the provenance edges such that for* $(x, y) \in PIDSET \times PIDSET$ *and for* $x \neq y$ *the process documented in provenance node with* PID y *takes the output of process documented in node with* PID x *as its input.*

In this definition, we store four kind of information about the process execution in the provenance: the description of the process, the information about the process executor, the list of the inputs and an output.

3.2 Participants in the provenance system

The provenance system consists of the following participants:

- A data storage DB, where the data is stored
- A provenance storage PS, where the provenance of data is stored for a long term storage
- Process executors C with identification CID is the actors who execute the process, produce the data, submit data to DB, and submit the provenance to PS
- An auditor ADT, is the actor who checks the integrity of the provenance and data

3.3 The basic provenance recording method

A simple model of the provenance recording system is as follows. The process executor C_i queries the inputs $I \subseteq D$ from DB and a set of $\{PID_{in}\}$, that is the provenance ID of I. C_i executes the process that produces the data output d_i, creates the provenance node p_i and a set of edges, stores d_i to DB and submits provenance node p_i and edges to PS. The provenance edges are a set of mapping from the PID of provenance p_i to the parents that produce I, so we write the edges as $\{PID, PID_{in}\}$. We define the simple protocol as follows [Note: we use the index $_{in}$ to refer to the parents that produce the inputs I, for example PID_{in} is a PID of the parent, C_{in} is a process executor at the parent]:

$$DB \rightarrow C_i : I$$
$$PS \rightarrow C_i : \{PID_{in}\}$$
$$C_i \rightarrow DB : d_i$$
$$C_i \rightarrow PS : node = p_i, edge = \{PID, PID_{in}\}$$

3.4 Aggregate Signature *Aggr* and *TreeAggr*

We define a function $Aggr$ which aggregates the signatures of different messages created by one or many different parties into one signature. A function $TreeAggr$ aggregates the signatures into a set of signatures where the number of element is less or equal to the total number of the original signatures. The method to aggregate the signatures using $TreeAggr$ is explained in Section 5.

4 Integrity scheme for provenance by keeping "the provenance" of the provenance

4.1 Provenance recording: Securing Provenance with signature-chain

For a secure provenance recording, when querying inputs I and the provenance of the inputs PID_{in}, the process executor also queries the collection of signature of inputs, that is the *signnode* in all parent nodes. In the scheme, we include a new participant, a Trusted Provenance Mediator (TPM), which is a trusted party who mediates the provenance recording. We assume that this party is trusted and will not cheat for any purposes. The complete provenance recording protocol is as follows:

$$\begin{aligned}
DB \rightarrow C_i &: I \\
PS \rightarrow C_i &: \{PID_{in}, Sign_{C_{in}}(p_{in}, d_{in})\} \\
C_i \rightarrow DB &: d_i \\
C_i \rightarrow TPM &: Insert(node = p_i, edge = \{PID, PID_{in}\}, \\
&\quad signnode = Sign_{C_i}(p_i, d_i), \\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\}) \\
TPM \rightarrow PS &: node = p_i, edge = \{PID, PID_{in}\}, \\
&\quad signnode = Sign_{C_i}(p_i, d_i), \\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\}, \\
&\quad signtpm = Sign_{TPM}(signnode, signedge)
\end{aligned}$$

The main difference of this scheme with the basic provenance recording method in Section 3.3 is we include signature chain to the provenance, by recording the signature of the node (*signnode*) and signatures of all edges that connect the node to the parents (*signedge* – represented by the parent signatures). The signatures are signed by the TPM before submitted to the provenance store PS (*signtpm*).

4.2 Provenance update: Allowing update in the signature chain

Update to a node by the same process executor is allowed, but we should keep the aggregate of the previous versions of the node. For example, C_i updates p_i to p'_i and d_i to d'_i and no update to the edges, so that no change to *signedge*.

This update will delete the previous version of p_i, and change it to p'_i. However it records the signatures of the previous versions in the form of an aggregate signature.

$$\begin{aligned}
C_i \rightarrow DB &: Update(d'_i)\\
C_i \rightarrow TPM &: Update(node = p'_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Sign_{C_i}(p'_i, d'_i),\\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\})\\
TPM \rightarrow PS &: node = p'_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Aggr\{Sign_{C_i}(p_i, d_i)\}, Sign_{C_i}(p'_i, d'_i)),\\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\},\\
&\quad signtpm = Sign_{TPM}(signnode, signedge)
\end{aligned}$$

The second update by C_i changes p'_i to p''_i and d'_i to d''_i and no change to *signedge*.

$$\begin{aligned}
C_i \rightarrow DB &: Update(d''_i)\\
C_i \rightarrow TPM &: Update(node = p''_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Sign_{C_i}(p''_i, d''_i),\\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\})\\
TPM \rightarrow PS &: node = p''_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Aggr\{Sign_{C_i}(p_i, d_i), Sign_{C_i}(p'_i, d'_i))\}, Sign_{C_i}(p''_i, d''_i))\\
&\quad signedge = \{Sign_{C_{in}}(p_{in}, d_{in})\},\\
&\quad signtpm = Sign_{TPM}(signnode, signedge)
\end{aligned}$$

We can accept any number of updates, however we only store the signatures of the previous versions of the nodes whose outputs are used by at least one child node.

4.3 The case of updates of the parent nodes

In Section 4.2, we assume no updates to *signedge* which means that no change to the input used by the node. However, a child may update the parent to the newest outputs of the parent. In this case, the child should keep the signature of the previous version of the parent using $TreeAggr$. For example, C_{in} updates p_{in} to p'_{in} and d_{in} to d'_{in}. C_{in} does exactly the same update as described in Section 4.2. After C_{in} updates the provenance and data, C_i also wants to update the parent to the newest version. So, C_i update *signedge* to $\{Sign_{C_{in}}(p'_{in}, d'_{in})\}$. The scheme to update the provenance in this case is as follows.

$$\begin{aligned}
DB \rightarrow C_i &: I\\
PS \rightarrow C_i &: \{PID_{in}, Sign_{C_{in}}(p'_{in}, d'_{in})\}\\
C_i \rightarrow DB &: Update(d'_i)\\
C_i \rightarrow TPM &: Update(node = p'_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Sign_{C_i}(p'_i, d'_i),\\
&\quad signedge = \{Sign_{C_{in}}(p'_{in}, d'_{in})\})\\
TPM \rightarrow PS &: node = p'_i, edge = \{PID, PID_{in}\},\\
&\quad signnode = Aggr\{Sign_{C_i}(p_i, d_i)\}, Sign_{C_i}(p'_i, d'_i)),\\
&\quad signedge = \{TreeAggr\{Sign_{C_{in}}(p_{in}, d_{in}), Sign_{C_{in}}(p'_{in}, d'_{in})\}\},\\
&\quad signtpm = Sign_{TPM}(signnode, signedge)
\end{aligned}$$

In this case, we aggregate all versions of *signedge* using $TreeAggr$.

4.4 Signature Verification

To verify consistency between a node and all of its children, the auditor ADT queries the *signnode* on the parent and all *signedge*(s) on the children. The ADT combines the *signedge*(s) to get an aggregate signatures of the parent and compares the result with *signnode*. A detailed example is described in Section 5.

5 Aggregating Signatures of Previous Versions of a provenance node using *TreeAggr*

In the provenance, we use the aggregate signature to save the spaces needed for storing the signatures of all previous versions of the provenance nodes. Rather than verifying the aggregate signatures using the messages and the public key of the signers, we verify the signatures by comparing the aggregate signatures stored at a node with the aggregate of all signatures stored at all children of the nodes (we cannot verify the signatures by checking the messages – the previous versions of the provenance – because they have been deleted to save the spaces in each update). The motivation to use $TreeAggr$ is because each child may not store all versions of the parent. So, if we use normal aggregate signatures $Aggr$ to save the spaces, and stores the aggregates in each child, we may not be able to combine the signatures to form a full aggregate of all versions of the parent.

Using $TreeAggr$, at first we represent all versions of the parent in all leave nodes in the signature tree. The child can aggregate the signatures of two consecutive versions (even and odd) in the leave nodes and stores the aggregate in the parent of the leaves. The same method is applied for each level of nodes in the tree to combine the signatures into a smaller number of signatures stored in the nodes at the higher level until we cannot get two consecutive nodes to aggregate.

We give an example of the aggregation as follows. The node Q created by a process executor C has four versions: Q, Q', Q'', Q''', it has 6 children R, S, T, U, V, W. R and S use the output of the first, second and third versions of Q, while T and U only uses the output of the second version. V and W use the output of the third version of Q. After sometimes, the children nodes S, U and W update the parent version to the latest version (Q''').

In the above case, in Q, we store the aggregate signatures of the previous version of Q, that is $Aggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q''\}$ and the latest signatures ($Sign_C(Q''')$). For each child, we store the $TreeAggr$ of all the parent signatures of the child. So, that for each child we store the signatures as follows:

$$\begin{aligned}
signedge_R &= TreeAggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q'')\}\\
&= Aggr\{Sign_C(Q), Sign_C(Q')\}, Sign_C(Q'')\\
signedge_S &= TreeAggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q''), Sign_C(Q''')\}\\
&= Aggr\{Sign_C(Q), Sign_C(Q')\}, Aggr\{Sign_C(Q''), Sign_C(Q''')\}\\
&= Aggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q''), Sign_C(Q''')\}\\
signedge_T &= TreeAggr\{Sign_C(Q')\}\\
&= Sign_C(Q')\\
signedge_U &= TreeAggr\{Sign_C(Q'), Sign_C(Q''')\}\\
&= Sign_C(Q'), Sign_C(Q''')\\
signedge_V &= TreeAggr\{Sign_C(Q'')\}\\
&= Sign_C(Q'')\\
signedge_W &= TreeAggr\{Sign_C(Q''), Sign_C(Q''')\}\\
&= Aggr\{Sign_C(Q''), Sign_C(Q''')\}
\end{aligned}$$

To check the integrity of the child nodes, we aggregate the signatures in some child nodes and compare the aggregate with the aggregate signatures on the parent. The parent Q stores an aggregate of the previous and the latest versions in $signnode = Aggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q''\}, Sign_C(Q''')$. Because the aggregation can be performed incrementally we may aggregate the signatures into

$$\begin{aligned}
signnode &= Aggr\{Aggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q'')\}, Sign_C(Q''')\}\\
&= Aggr\{Sign_C(Q), Sign_C(Q'), Sign_C(Q''), Sign_C(Q''')\}
\end{aligned}$$

We can compare the aggregate signatures on children with $signnode$ stored at the parent by finding correct combination of the tree-aggregate on the children to form the same aggregate signature as stored in $signode$. Some examples of the correct combinations are shown as follows (note: we get $Sign_C(Q''')$ from $signedge_U$):

$$\begin{aligned}
signnode &= Aggr\{signedge_R, Sign_C(Q''')\}\\
&= signedge_S
\end{aligned}$$

6 Security Analysis

To show that a child is a node that uses an output of a version of the parent, the auditor *ADT* should query all *signedge* of all children of the node, find a combination of signatures that can reproduce *signnode* and compare with *signnode* on the parent. Using the scheme explained in the previous versions, it is always possible to find a combination of signatures in the children of a node which produce exactly the same signature as *signnode*.

Theorem 1. *The Auditor ADT can always find a combination of the signatures* ***singedge*** *on the children which produce* ***signnode*** *in the parent.*

Proof. The parent node stores the signatures of the latest version and the signatures of the previous versions of the node, except for the versions that do not have any children. So, that a signature of a previous version of the parent node is stored in at least one child node. Each child only aggregates the signature of two consecutive versions of the parent, so that there are only two cases of the signatures: (1) the signature is not combined with another signature, in this case the signature is stored in its original form (2) the signature is combined with another signature to form an aggregation of two signatures.

If a signature σ_i is not combined, we should find the other signature of the consecutive version in other children. Because the other signature should be stored in at least one other child, if the other signature is stored as the first case (no combination with other signature), we can combine with σ_i to get a combination of two consecutive signatures. If the other signature has been combined, it should have been combined with σ_i (the same signatures stored in the other child), so in all cases we can combine elements in all *singedge* in the children to form *signnode* at the parent.□

7 Storage Requirements

The storage for *signnode* is always constant for any number of updates to the node, which is two signatures for each node (one signature for the latest version, and a signature for aggregate of signatures of all previous versions). As of the storage for *signedge*, in the best case, a child aggregates all the signatures of all versions of the parent, so it only needs to store one signature for each parent. The worst case is if the child does not uses any two consecutive versions of the parent, so we cannot reduce the number of signatures in $TreeAggr$. In this case, the number of signatures is $n/2$ where n is the number of updates to parent.

References

1. Rocío Aldeco-Pérez and Luc Moreau. Securing provenance-based audits. In *IPAW*, pages 148–164, 2010.
2. Sergio Álvarez-Napagao, Javier Vázquez-Salceda, Tamás Kifor, László Zsolt Varga, and Steven Willmott. Applying provenance in distributed organ transplant management. In *IPAW*, pages 28–36, 2006.

3. Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In *ICALP*, pages 411–422, 2007.
4. Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432, 2003.
5. Uri Braun, Avraham Shinnar, and Margo Seltzer. Securing provenance. In *The 3rd USENIX Workshop on Hot Topics in Security (HOTSEC 2008)*, USENIX HotSec, pages 1–5, Berkeley, CA, USA, July 2008. USENIX Association.
6. Vikas Deora, Arnaud Contes, Omer F. Rana, Shrija Rajbhandari, Ian Wootten, Tamás Kifor, and László Zsolt Varga. Navigating provenance information for distributed healthcare management. In *Web Intelligence*, pages 859–865, 2006.
7. Paul T Groth. *The Origin of Data: Enabling the Determination of Provenance in Multi-institutional Scientific Systems through the Documentation of Processes.* PhD thesis, University of Southampton, 2007.
8. Paul T. Groth and Luc Moreau. Recording process documentation for provenance. *IEEE Trans. Parallel Distrib. Syst.*, 20(9):1246–1259, 2009.
9. Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *J. Cryptology*, 3(2):99–111, 1991.
10. Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *ACM workshop on Storage security and survivability (StorageSS 2007)*, pages 13–18, 2007.
11. Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *7th Conference on File and Storage Technologies (FAST 2009)*, pages 1–14, 2009.
12. Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *ACM Transactions on Storage*, 5(4):12:1–12:43, December 2009.
13. Tamás Kifor, László Zsolt Varga, Javier Vázquez-Salceda, Sergio Álvarez-Napagao, Steven Willmott, Simon Miles, and Luc Moreau. Provenance in agent-mediated healthcare systems. *IEEE Intelligent Systems*, 21(6):38–46, 2006.
14. Simon Miles, Paul T. Groth, Steve Munroe, Sheng Jiang, Thibaut Assandri, and Luc Moreau. Extracting causal graphs from an open provenance data model. *Concurrency and Computation: Practice and Experience*, 20(5):577–586, 2008.
15. Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul T. Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric G. Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.
16. Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In *IPAW*, pages 323–326, 2008.
17. Kiran-Kumar Muniswamy-Reddy. *Foundations for Provenance-Aware Systems.* PhD thesis, Harvard University, 2010.
18. Yogesh Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. In *SIGMOD Record*, pages 31–36, 2005.
19. Amril Syalim, Takashi Nishide, and Kouichi Sakurai. Preserving integrity and confidentiality of a directed acyclic graph model of provenance. In *IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2010), LNCS 6166*, pages 311–318, 2010.
20. Amril Syalim, Takashi Nishide, and Kouichi Sakurai. Securing provenance of distributed processes in an untrusted environment. *IEICE Transactions*, 95-D(7):1894–1907, 2012.
21. Victor Tan, Paul T. Groth, Simon Miles, Sheng Jiang, Steve Munroe, Sofia Tsasakou, and Luc Moreau. Security issues in a soa-based provenance system. In *IPAW*, pages 203–211, 2006.