Repairing Provenance Policy Violations by Inventing Non-Functional Nodes

Saumen Dey[1], Daniel Zinn[2], and Bertram Ludäscher[1 2]

[1] Dept. of Computer Science, University of California, Davis
[2] Genome Center, University of California, Davis

Abstract. In scientific collaborations, provenance is increasingly used to explain, debug, reproduce, and determine the validity and quality of data products. In such environments, it can be infeasible or undesirable to publish the complete provenance of all the final output data products. We have developed ProPub, a system that allows users to publish a customized version of their data provenance, based on a set of publication and customization requests, while observing certain provenance publication policies, expressed as logic integrity constraints. The user's customization requests may violate one or more integrity constraints. In previous work, we removed additional parts of the provenance graph (i.e., not directly requested by the user) to repair policy violations. In this paper, we present an alternative approach which ensures that all relevant nodes are retained in the provenance graph. The key idea is to introduce new (non-functional) nodes that are used to represent lineage dependencies, without revealing information that the user wants to protect. With this new approach, a user may now explore different provenance publication strategies, and choose the most appropriate one before publishing sensitive provenance data.

1 Introduction

In the emerging paradigm of collaborative, data-intensive science, sharing data products even prior to publication is desirable [1,2]. Yet, without a proper scientific publication associated with openly published data, its validity and accuracy might be questionable. This is problematic in an open environment, where published data by one scientist is used by another scientist as input for further data analyses. In such an environment, data provenance (the lineage and processing history of data) can help to ensure data quality [3,4,5,6,7]. It is thus desirable to publish data products together with their provenance.

In many cases, however, provenance data can be sensitive and may contain private information or intellectual property that should not be revealed [7,8,5]. Consequently, a balancing act (Figure 1) is necessary between (i) the desire to publish provenance data so that collaborators can understand and rely on the shared data products, and (ii) the need to protect sensitive information, e.g., due to privacy concerns or intellectual property issues.

We view provenance as a bipartite, directed, acyclic graph, capturing which data nodes were consumed and produced, respectively, by invocation (i.e., computation) nodes. Our model thus corresponds to the Open Provenance Model (OPM) which captures the dependencies between data artifacts and invocations [9,10].

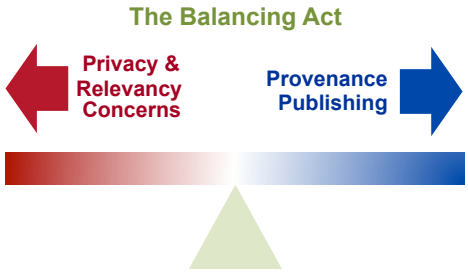


Fig. 1. In collaborative settings, scientists publish provenance for an improved understanding of the result data. With increasing privacy concerns, collaborators have to choose the right balance between providing sufficient provenance data and protecting sensitive information.

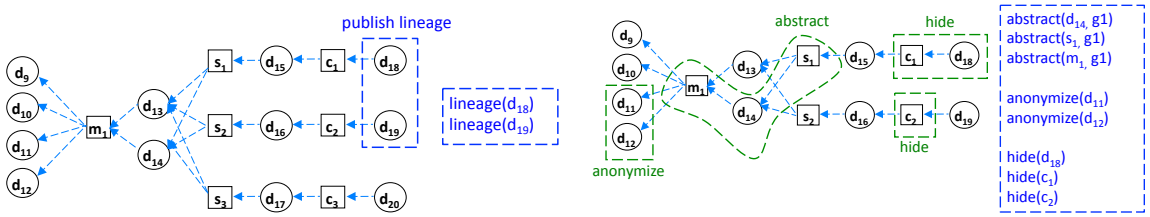
To sanitize provenance graphs, a scientist can remove sensitive data nodes or invocations nodes from the provenance graph. Alternatively, she can *abstract* a set of sensitive nodes by grouping them into a single, abstract node. This update may violate some of the integrity constraints of the provenance graph [11]. For example, grouping multiple nodes into one abstraction node may introduce new dependencies which were absent in the initial provenance graph. Hiding nodes may also make some nodes in the final graph appear independent of each other even though they are dependent in the initial graph. Thus, one can no longer trust that the published provenance data is "correct" (e.g., there are no false dependencies) or "complete" (e.g., there are no false independencies). Therefore, we propose a system that allows a publisher to provide a high-level specification what parts of the provenance graph are to be published and which parts are to be sanitized, *while guaranteeing* that at the same time certain provenance publication constraints are observed.

2 Motivating Example

Figure 2(a) shows the provenance graph (PG) taken from the First Provenance Challenge [12]. *Data nodes* are depicted as circles and *invocation nodes* (representing computations) as boxes; dependencies among them are shown as directed edges. These edges capture the lineage of data and thus are typically drawn from right (newer nodes) to left (older nodes). For example, $\mathtt{d}_{16}$ *was generated by* an invocation $\mathtt{s}_2$, and was in turn *used* by invocation $\mathtt{c}_2$, denoted by, respectively $\mathtt{s}_2 \overset{gen\_by}{\longleftarrow} \mathtt{d}_{16}$ and $\mathtt{d}_{16} \overset{used}{\longleftarrow} \mathtt{c}_2$.

Let us assume, the user wants to publish data products $\mathtt{d}_{18}$ and $\mathtt{d}_{19}$ along with their lineage data. Then, she will issue the publication requests as shown in Figure 2(a). A recursive query is used to retrieve all data and invocation nodes upstream from $\mathtt{d}_{18}$ and $\mathtt{d}_{19}$ and we get a modified provenance graph (PG′) as shown in Figure 2(b). Note that the lineage of $\mathtt{d}_{20}$ up to $\mathtt{s}_3$ is not relevant for $\mathtt{d}_{18}$ and $\mathtt{d}_{19}$ and hence not included in PG′. Further assume that before publishing PG′, the user also requests a set of customizations as shown in Figure 2(b).

Figure 3 shows the provenance graph we get after applying all the customization requests. We see that this provenance graph violates three provenance policies: There



(a) Provenance graph (PG) and publish request (b) User requests: `anonymize`, `abstract`, `hide`

Fig. 2. (a) User requests to publish the provenance of $\{d_{18}, d_{19}\}$; and (b) customization requests to *anonymize* data nodes $\{d_{11}, d_{12}\}$, to *abstract* nodes $\{m_1, d_{14}, s_1\}$, and to *hide* $\{c_1, d_{18}, c_2\}$

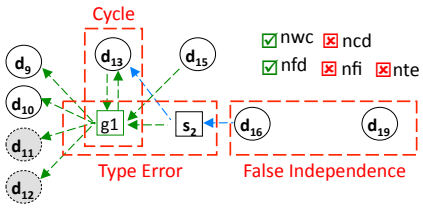


Fig. 3. Provenance graph after applying all user requests. Provenance policies No-Type Error (NTE), No-Cyclic Dependency (NCD) and No-False Independence (NFI) are violated, while No-Write Conflict (NWC) and No-False Dependence (NFD) are satisfied.

is a cycle between d_{13} and g_1, a type error for the edge from s_2 to g_1 (the graph should be bipartite), and there is no dependency between d_{19} and d_{16}, violating, respectively, the No-Cyclic Dependency (NCD), No-Type Error (NTE) and No-False Independence (NFI) policies. On the other hand, the provenance policies No-Write Conflict (NWC) and No-False Dependence (NFD) are not violated by these customization requests.

Outline and Contributions. In Section 3, we first describe the provenance model, user requests, provenance policies, and logical architecture of ProPub. This overall framework was proposed recently in [11]. In Section 4 we present our main contribution, i.e., a new way to repair policy violations, not by removing additional nodes (as in our prior work), but by introducing new (non-functional) nodes that represent the original lineage dependencies, without revealing information that the user wants to protect. We describe in detail how policy violations will be repaired such that all relevant nodes are retained in the final provenance graph. Related work is discussed in Section 5 and Section 6 presents some concluding remarks and suggestions for future work.

3 Provenance Publisher (ProPub)

In our recent work, we developed the system ProPub [11], which uses a declarative approach to publish customized policy-aware provenance. ProPub accepts the initial provenance graph and two types of input specifications. (i) User Requests: the publication and customization requests, and (ii) Provenance Policies: the integrity constraints

Relation Name	Description
`used`(I, D)	An edge specifying that the invocation I used the data artifact D.
`gen_by`(D, I)	An edge to indicate that the data artifact D was generated by invocation I.
`actor`(I, A)	An invocation node I, which was executed by actor A.
`data`(D, R)	A data artifact node, whose value can be retrieved using the reference R.
`dep`(X, Y)	An auxiliary relation and defined as `dep` = `used` ∪ `gen_by` and to specify that node X depends on node Y, irrespective of the node types.

Table 1. Provenance Model for PROPUB

User Request	Description
`ur:lineage`(D)	Selects the complete lineage for the data artifact D
`ur:anonymize`(N)	Erases the actor/process identify or the data reference from the node N
`ur:hide`(N)	Removes the invocation or data node N
`ur:abstract`(N, G)	Collapses all nodes N to the abstract group G
`ur:retain`(N)	Keeps the node N in the customized provenance

Table 2. User requests for lineage publication and customization

to be observed. PROPUB then applies all user requests on the initial provenance graph and checks for policy violations. In case there is a violation, it applies repairs and generates the customized provenance graph.

Provenance Model. The provenance model used in PROPUB is based on OPM, the Open Provenance Model [13] and our earlier work [14]: A *provenance* (or *lineage*) *graph* is an acyclic graph $PG = (V, E)$, where the nodes $V = \mathrm{D} \cup \mathrm{I}$ represent either *data* items D or actor *invocations* I. The graph G is bipartite, i.e., the edges $E = E_{\texttt{use}} \cup E_{\texttt{gby}}$ are either *used* edges $E_{\texttt{use}} \subseteq \mathrm{I} \times \mathrm{D}$ or *generated-by* edges $E_{\texttt{gby}} \subseteq \mathrm{D} \times \mathrm{I}$. Here, a *used* edge $(\texttt{i}, \texttt{d}) \in E$ means that invocation `i` has read `d` as part of its *input*, while a *generated-by* edge $(\texttt{d}, \texttt{i}) \in E$ means that `d` was *output* data, written by invocation `i`. We use the schema shown in Table 1.

User Requests. The user requests supported by the PROPUB framework are summarized in Table 2. The PROPUB system expects user requests to be asserted as relational facts that can then be used by a Datalog rule engine. An user request can be a publication request or a customization request. A customization user request can request to remove a node or an edge or to keep that in the final graph.

Provenance Policies. The provenance graph supported by PROPUB is a bipartite directed acyclic graph. Also, an invocation can read many data artifacts, but a data artifact is written by exactly one invocation. We developed three provenance policies to verify if these structural properties are satisfied in the provenance graph $\mathrm{PG}'^{\Delta u}$, which we get after applying all the customization requests on PG'. PROPUB has two more provenance policies to ensure the correctness and completeness of information. These provenance policies are briefly defined in Table 3.

Provenance Policy	Description
No-Write Conflict (NWC)	A data artifact can be written by only one invocation.
No-Cyclic Dependency (NCD)	There is no cycle between any two nodes X and Y.
No-Type Error (NTE)	Two nodes with a direct dependency are of different types.
No-False Dependence (NFD)	Two nodes are dependent in $\mathrm{PG}'^{\Delta u}$ only if they are dependent in PG'.
No-False Independence (NFI)	Two nodes are independent in $\mathrm{PG}'^{\Delta u}$ only if they are independent in PG'.

Table 3. Provenance Policies

Constraint	Description
ic:wc(X, Y)	Write conflict: invocations X and Y are creating the same data node.
ic:cd(X, Y)	Cyclic dependency between nodes X and Y.
ic:te(X, Y)	Type error: nodes X and Y are connected via *used* or gen_by edges, but don't have the corresponding node types.
ic:fd(X, Y)	False dependency: node Y depends on X in $\mathrm{PG}'^{\Delta u}$, but not in PG'.
ic:fi(X, Y)	False independence: node Y depends on X in PG', but not in $\mathrm{PG}'^{\Delta u}$.

Table 4. Integrity constraint relations used to detect policy violations

We use a set of integrity constraints (ICs) to check whether the provenance policies defined in Table 3 are satisfied. Table 4 lists the "witness relations" that are defined by rules (not shown) and which are used to detect particular IC violations.[3]

3.1 Logical Architecture

The logical architecture of the PROPUB system is shown in Figure 4. The user submits a set of publication and customization requests U_0. The module Direct-Conflict-Detection detects *direct conflicts* among the given user-requests. For example, a hide and a retain request on the same node is an obvious conflict. The user needs to update her original requests until all *direct conflicts* are resolved, resulting in a conflict-free user request U. The Lineage-Selection module computes the sub graph PG', which contains all to-be-published data items (specified using the 'lineage' predicate) together with their complete provenance.

The Request-Policy-Evaluation module calculates the updates (Δu: inform of *insert* and *delete*) needed to apply all the user requests from U on PG'. It applies Δu on PG' and get a customized provenance graph $\mathrm{PG}'^{\Delta u}$. Then it checks if all the selected provenance policies (PP) are observed by evaluating respective integrity constraints. In case some of the policies are violated, this module calculates updates (Δp: inform of *insert* and *delete*) needed to repair the violations. In a final conflict resolution step using the module Implied-Conflict-Detection-Resolution, the system detects all such implied

[3] For example., we can detect whether a data node is created by different invocations X and Y and record this as ic:wc(X, Y).

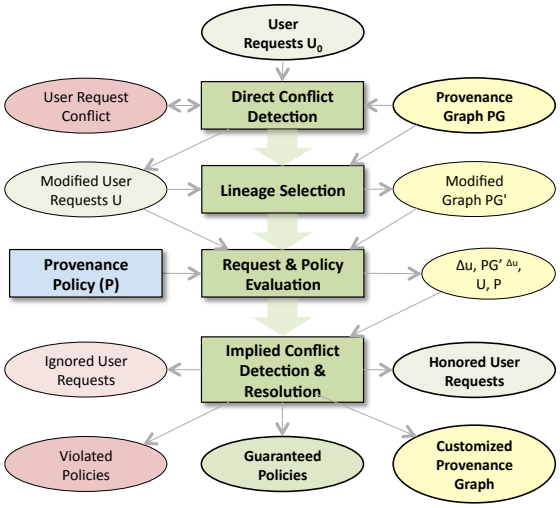


Fig. 4. PROPUB Architecture

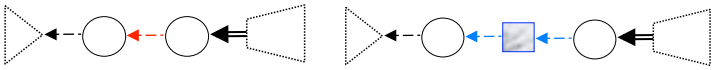
conflicts by comparing Δu and Δp. In case an *implicit conflict* is detected, it selects another subset from the given U and PP following the user preferences. These steps are repeated until there is no more policy violations. It then applies Δp on $\mathrm{PG}'^{\Delta u}$ to get the customized provenance graph (CG) ready to be published.

4 Repairing Policy Violations

If we apply the customization user requests on PG', we get an intermediate provenance graph $\mathrm{PG}'^{\Delta u}$ as we shown in Figure 2(a), 2(b) and 3. But, $\mathrm{PG}'^{\Delta u}$ may violate one or more provenance policies. In case the $\mathrm{PG}'^{\Delta u}$ violates a structural policy (NWC, NCD, and NTE), it will no more be a proper provenance graph. Also, in case it violates a non-structural policy (NFI and NFD), $\mathrm{PG}'^{\Delta u}$ may contain incorrect information or may become incomplete. Thus, user will not be able to publish $\mathrm{PG}'^{\Delta u}$. To resolve this issue, we apply the customization requests U on PG' in a strategic way such that it confirms to all the provenance policies.

Our strategy is primarily based on two ideas (i) inventing non-functional nodes, and (ii) converting user requests using other forms of user requests.

Inventing Non-functional Nodes. In case $\mathrm{PG}'^{\Delta u}$ has a structural violation, PROPUB resolves the violation by adding a new non-functional node. A non-functional node is added to maintain the structure of provenance graph. Presence of a non-functional node in the final customized graph may represent one data or invocation node or a set of data and invocation nodes. No mapping is maintained between the non-functional node and the nodes it replaced. Also, it will not carry any URL. Thus, no one will be able to reach to the value of a data artifact or the source code of an actor from a non-functional



(a) Policy Violation (b) Fix

Fig. 5. (a) direct dependency between data nodes causing a type error (NTE violation); (b) PROPUB resolves this by inventing a non-functional invocation node.

node. PROPUB invents minimum numbers of non-functional nodes to resolve a policy violation.

PROPUB uses the same strategy to resolve NFD policy violations. The fix to the violation of this policy is complex and may need more than one non-functional node to be added. In spite of this complexity, PROPUB resolves the violation using the minimum numbers of non-functional nodes.

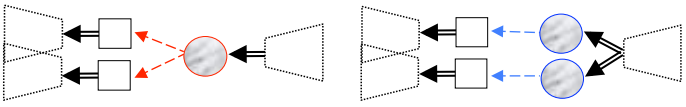
Converting User Requests. A publisher can use *ur:hide* requests to hide individual nodes or the partial structure of the provenance graph. When we apply these user requests all the selected nodes and the associated edges are removed from the provenance graph PG' and a set of independence may be created which violates the NFI policy. We can use the inventing *new non-functional node* strategy as discussed above and replace the selected node by a non-functional node to resolve this policy violation. But, this approach keeps the structure of the original provenance graph in the final provenance graph. Instead, PROPUB converts these *ur:hide* user requests into an equivalent set of *ur:abstract* user requests so that all the selected nodes are removed and no unintended dependencies are removed.

4.1 Repairing Structural Policy Violations

No-Type Error. This policy is violated in case there is a direct dependency between two nodes of same type (i.e. a dependency between two data artifacts or a dependency between two invocations). PROPUB invents a non-functional invocation node in case the policy violation is between two data artifacts as shown in Fig. 5. In the similar way, PROPUB invents a non-functional data node in case the policy violation is between two invocation nodes. We used the rules as shown below to create the non-functional nodes and fix the violations of this policy.

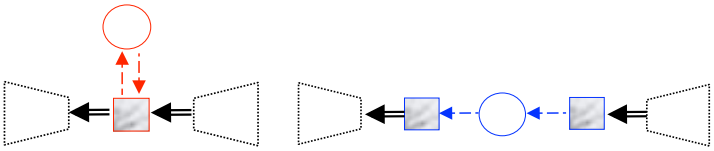
```
del_dep(X,Y) :- ic:te(X,Y).
add_data(f(X,Y),T) :- ic:te(X,Y), d_actor(X,_), T='ic:te'.
add_actor(f(X,Y),T) :- ic:te(X,Y), d_data(X,_), T='ic:te'.
add_dep(X,f(X,Y)) :- ic:te(N1,N2), X is N1.
add_dep(f(X,Y),Y) :- ic:te(N1,N2), Y is N2.
```

No-Write Conflict. This policy is violated in case there are $N(N \geq 2)$ *gen_by* edges for a data node. To resolve this violation PROPUB removes incorrect *gen_by* edges for the violated data node and keeps only one *gen_by* edge, which is there in PG. But, this



(a) Policy Violation (b) Fix

Fig. 6. In (a) there are two *gen_by* edges with the data node causing the No-Write conflict policy violation and PROPUB resolves this by inventing a non-functional data node as shown in (b).



(a) Policy Violation (b) Fix

Fig. 7. In (a) there is a cycle between the data node and the invocation node and PROPUB resolves this by inventing the non-functional invocation node as shown in (b).

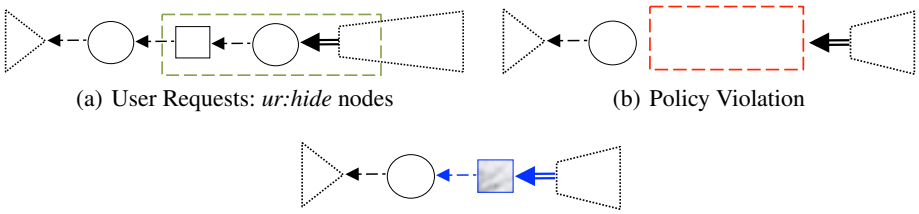
may violate the NFI policy as it removes dependencies for $N-1$ invocation nodes. To get around this side effect PROPUB invents $N-1$ non-functional data nodes and creates $N-1$ *gen_by* edges as shown in Fig. 6. Lastly, it copies all the *used* edges for the violated data node over to all $N-1$ non-functional data nodes. Following rules are used to create the non-functional nodes and fix the violations of this policy:

```
del_data(D) :- ic:wc(D).
del_dep(D,I) :- ic:wc(D), d_gen_by(D,I).
add_data(f(D,I),T) :- ic:wc(D), d_gen_by(D,I), T='ic:wc'.
add_dep(f(D,I),I) :- ic:wc(D), d_gen_by(D,I).
add_dep(I,f(D,I1)) :- ic:wc(D), d_gen_by(D,I1), d_used(I,D).
```

No-Cyclic Dependency. This policy is violated in case a node is reachable from itself. In Fig. 7, there is a cycle between a invocation node and a data node. To fix this violation PROPUB invents a non-functional invocation node and creates a *used* edge between the data node and the non-functional invocation node. Then it removes all the *gen_by* edges from the invocation node (except the one with the data node with which it has the cycle) and copies them over to the non-functional invocation node. In the similar way, PROPUB resolves this violation between two invocation nodes.

4.2 Repairing No-False Independence (NFI) Policy Violations

This policy is violated in case two nodes are not dependent in $\mathrm{PG}'^{\Delta u}$ even though they are in PG'. This may occur in case the *ur:hide* user requests are applied on PG' as shown in Fig. 8. One way to resolve this violation is to insert direct dependencies, which



(a) User Requests: *ur:hide* nodes (b) Policy Violation

(c) Fix

Fig. 8. PG' and user requests *ur:hide* are shown in (a). In (b) some dependencies are removed between nodes in $\mathrm{PG}'^{\Delta u}$. PROPUB then resolves this in two steps (i) transforms these *ur:hide* requests into equivalent *ur:abstract* requests and (ii) applies these *ur:abstract* requests on PG' and gets the customized graph is shown in (c).

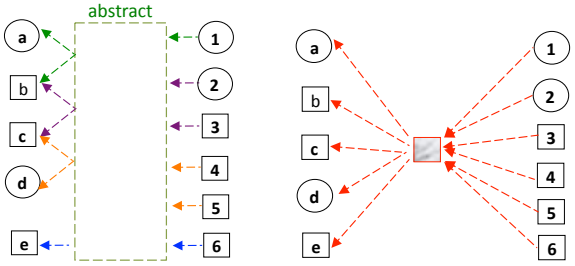
are there in PG' but missing in $\mathrm{PG}'^{\Delta u}$, between any two nodes in $\mathrm{PG}'^{\Delta u}$. But, this process may add too many edges and the graph may become unreadable. One optimization to this process is to develop transitive dependencies to reduce the total number of new edges needed. This may be computation intensive. PROPUB uses a different strategy to fix this violation. Following rules are used to transform the *ur:hide* requests into an equivalent set of *ur:abstract* requests:

```
hide_connected(X,Y) :- ur:hide(X), ur:hide(Y), dep(X,Y).
hide_connected(X,X) :- ur:hide(X).
hide_connected(X,Y) :- hide_connected(Y,X).
hide_connected(X,Y) :- hide_connected(X,Z), hide_connected(Z,Y).
smaller(X) :- hide_connected(X,Y), X < Y.
minimum(X) :- ur:hide(X), not(smaller(X)).
abstract_hide(X,G) :- hide_connected(X,G), minimum(G).
```

The customization user requests *ur:abstract* removes nodes from PG', but does not violate the NFI policy. To avoid the NFI polic violations PROPUB transforms the *ur:hide* user requests into an equivalent set of *ur:abstract* user requests. These will be applied to PG' in the same way the User issued *ur:abstract* requests are applied.

4.3 Repairing No-False Dependence (NFD) Policy Violations

This policy is violated in case two nodes are dependent in $\mathrm{PG}'^{\Delta u}$ even though they are not in PG'. This may occur in case the *ur:abstract* user requests are applied on PG' as shown in Fig. 9. In Fig. 9(a) we have a partial provenance graph showing the *ur:abstract* requests and the nodes with direct dependencies with one or more nodes selected to be abstracted. This figure shows that in PG' the data artifact '1' depends on data artifact 'a' and 'b'. In the similar way, the data node '2' depends on invocation nodes 'b' and 'c' and so on. Now, if we apply these *ur:abstract* requests by collapsing all the selected nodes into a abstracted node then in $\mathrm{PG}'^{\Delta u}$ the data artifact '1' become depended on nodes 'a', 'b', 'c', 'd', and 'e' and thus making $\mathrm{PG}'^{\Delta u}$ incorrect, as shown in Fig. 9.



(a) The *ur:abstract* user requests

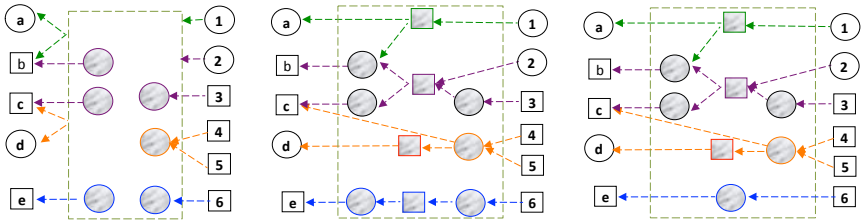
(b) The dependencies created by applying these *ur:abstract* user requests

Fig. 9. In (a) we show the boundary of one *ur:abstract* user requests set and the nodes with a direct dependency with one or more nodes selected to be abstracted. After these *ur:abstract* user requests are applied on PG' we get a new set of dependencies as shown in (b).

To avoid this policy violations ProPub takes a systematic three stages approach to apply the *ur:abstract* user requests. Instead of collapsing into one abstracted node, it invents a number of non-functional data and invocation nodes to maintain the dependencies between any two nodes in PG'^{Δ_u} as they are in PG'. This systematic approach ensures that the minimum number of non-functional nodes are invented. In the first stage, ProPub develops two sets in and out. The in is a set of data nodes which is used by some of the invocation nodes selected to be abstracted and invocation nodes which generated some of the data nodes selected to be abstracted. The out is a set of data nodes which is generated by one of the invocation node selected to abstracted and invocation nodes which used some of the data nodes selected to abstracted. It also calculates the dependencies for each of the node in set out on the nodes of the set in.

Now, ProPub creates non-functional data nodes for each of the invocation nodes from the sets in and out. One non-functional data node is created for exactly one invocation node from the set in through a *gen_by* edge. One non-functional data node is created for more then one invocation node from the set out through *used* edges in case these invocations have the same dependencies on the set in. At this stage, a non-functional data node is connected either to a node from the set in or to one or many nodes from the set out. For example, invocation node '4' '5' and depends on invocation nodes 'b' and 'c' and ProPub will create only one non-functional data node and two *used* edges. This is shown in Fig. 10(a).

In the second stage, it calculates the list of dependencies of all nodes from the set out to the nodes from in. ProPub creates one non-functional invocation node for each of these unique dependency lists and it creates *gen_by* edges for nodes from the set out which has the same dependency list. Then it creates *used* edges to connect to the nodes in in set from any of these non-functional invocation nodes. It will connect with respective non-functional data node created in the last stage in case an edge needs to be created with an invocation either from in or out. This outcome is shown in Fig. 10(b).



(a) Stage 1 (b) Stage 2 (c) Fix Stage 3

Fig. 10. Repairing No-False Dependence Policy Violations

Algorithm: CALCULATECUSTOMPG
INPUT: provenance graph PG, user requests U and provenance policies PP
OUTPUT: customized provenance graph CG

1. Test for *Direct Conflicts* // as explained in Section 3.1
2. IF there are *Direct Conflicts* THEN
3. RETURN *false* // User can resubmit after changing U
4. ELSE
5. Compute PG′// as explained in Section 3.1
6. Transform *ur:hide* user requests into *ur:abstract* user requests // as explained in Section 4.2
7. Apply all *ur:abstract* user requests on PG′ to get PG′$^{\Delta u}$// as explained in Section 4.3
8. Resolve NCC violations on PG′$^{\Delta u}$// as explained in Section 4.1
9. Resolve NWC violations on modified PG′$^{\Delta u}$// as explained in Section 4.1
10. Resolve NFT violations on modified PG′$^{\Delta u}$// as explained in Section 4.1
11. CG = PG′$^{\Delta u}$// Final customized provenance graph
12. RETURN CG

Fig. 11. Computing CG using the *Inventing Non-Functional Nodes* approach

In the final stage, PROPUB combines nodes if possible. For example, in Fig. 10(b) the path from node '6' to node 'e' has three consecutive non-functional nodes with no other dependencies. These three nodes can be replaced by only one non-functional data node. The result is shown in Fig. 10(c). Now, PROPUB removes all the nodes selected to be abstracted and associated edges from PG′.

4.4 Algorithm

The algorithm mentioned in Fig. 11 finds the customized provenance graph, if available. In this approach, we add non-functional nodes to repair policy violations. In Figure 3 we show that PG′$^{\Delta u}$ has a structural policy violation between nodes g_1 and s_2. Using this approach, we introduce a non-functional data node d such that d is dependent on g_1; and s_2 is dependent on d. Now, to fix the cycle between d_{13} and g_1 we introduce the non-functional invocation node g_2 and create a dependency (*gen_by*) edge from d_{15} to g_2. Then, we get the final CG as shown in Figure 12. Note that we are now able to keep all the relevant nodes in CG.

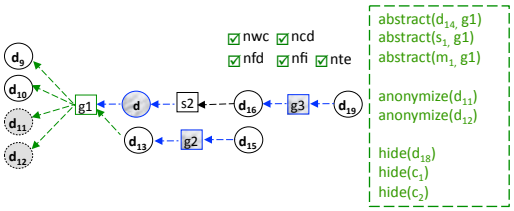


Fig. 12. Customized Provenance Graph after repairing all policy violations

5 Related Work

In [3,4,5,6,7], it has been observed that provenance can be used, e.g., to interpret results, diagnose errors, fix bugs, improve reproducibility, and generally to build trust on the final data products and the underlying processes. In addition, provenance can be used to enhance exploratory processes [15,16,17], and techniques have been developed to deal with provenance efficiently [18,19].

In many cases, provenance carries sensitive information, which can cause privacy concerns related to a data, actor, or workflow specification. Studying provenance, one can capture the functionality (being able to guess the output of the actor given a set of inputs) of an actor (module), or the execution flow of a workflow [8].

The security view approach [5] limits the available provenance to a user by providing a partial view of the workflow through a role-based access control mechanism, and by defining a set of access permissions on actors, channels, and input/output ports as specified by the workflow owner at design time. The ZOOM*UserViews approach [20] allows to define a partial, zoomed-out view of a workflow, based on a user-defined distinction between relevant and irrelevant actors. Provenance information is restricted by the definition of that partial view of the workflow.

In our recent work [11], we developed ProPub, which uses a declarative approach to publish customized policy-aware provenance. In this paper, we developed a new way to repair policy violations, not by removing additional nodes (as in [11]), but by introducing new (non-functional) nodes that represent the original lineage dependencies, without revealing information that the user wants to protect. We described in detail how policy violations will be repaired such that all relevant nodes are retained in the final provenance graph.

6 Conclusions

We discussed the need for provenance in scientific collaboration. Provenance data helps to build trust in the published results and data. However, provenance can also contain sensitive data and/or too much irrelevant detail. Thus, scientists should be able to "customize" provenance data before sharing it.

Our current ProPub system is based on the open provenance model (OPM). We plan to extend ProPub to include model extensions, e.g., to support structured data

structures, in particular nested collections [19]. Furthermore, ProPub currently suggests only one specific modified graph based on a given U and PP. In future work, we plan to investigate how to extend this approach to rank alternative solutions, thus supporting scientists even more in finding the desirable balance between revealing provenance information and preserving privacy when sharing data with collaborators.

References

1. Nature: Special Issue on Data Sharing. Volume 461. (September 2009)
2. Missier, P., Ludäscher, B., Bowers, S., Dey, S., Sarkar, A., Shrestha, B., Altintas, I., Anand, M., Goble, C.: Linking multiple workflow provenance traces for interoperable collaborative science. In: Workflows in Support of Large-Scale Science (WORKS), 2010 5th Workshop on, IEEE 1–8
3. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: a survey. ACM Computing Surveys (CSUR) **37**(1) (2005) 1–28
4. Simmhan, Y., Plale, B., Gannon, D.: A survey of data provenance in e-science. ACM SIGMOD Record **34**(3) (2005) 31–36
5. Chebotko, A., Chang, S., Lu, S., Fotouhi, F., Yang, P.: Scientific workflow provenance querying with security views. In: Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on, IEEE (2008) 349–356
6. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for Computational Tasks: A Survey. Computing in Science and Engineering **10**(3) (2008) 11–21
7. Davidson, S., Khanna, S., Roy, S., Boulakia, S.: Privacy issues in scientific workflow provenance. In: Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science, ACM (2010) 1–6
8. Davidson, S.B., Khanna, S., Panigrahi, D., Roy, S.: Preserving Module Privacy in Workflow Provenance. CoRR **abs/1005.5543** (2010)
9. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al.: The open provenance model core specification (v1. 1). Future Generation Computer Systems (2010)
10. Anand, M., Bowers, S., Ludascher, B.: Provenance browser: Displaying and querying scientific workflow provenance graphs. In: Data Engineering (ICDE), 2010 IEEE 26th International Conference on, IEEE (2010) 1201–1204
11. Dey, S., Zinn, D., Ludäscher, B.: PROPUB: Towards a Declarative Approach for Publishing Customized, Policy-Aware Provenance. In: Scientific and Statistical Database Management Conference (to appear). (2011)
12. Moreau, L., Ludäscher, B., Altintas, I., Barga, R., Bowers, S., Callahan, S., Chin, J., Clifford, B., Cohen, S., Cohen-Boulakia, S., et al.: Special issue: The first provenance challenge. Concurrency and Computation: Practice and Experience **20**(5) (2008) 409–418
13. Moreau, L., Clifford, B., Freire, J., Gil, Y., Groth, P., Futrelle, J., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Simmhan, Y., Stephan, E., den Bussche, J.V.: OPM: The Open Provenance Model Core Specification (v1.1). http://openprovenance.org/ (December 2009)
14. Anand, M., Bowers, S., McPhillips, T., Ludäscher, B.: Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In: Scientific and Statistical Database Management, Springer (2009) 237–254
15. Davidson, S., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD Conference, Citeseer (2008) 1345–1350

16. Freire, J., Silva, C., Callahan, S., Santos, E., Scheidegger, C., Vo, H.: Managing rapidly-evolving scientific workflows. Provenance and Annotation of Data (2006) 10–18
17. Silva, C., Freire, J., Callahan, S.: Provenance for visualizations: Reproducibility and beyond. Computing in Science & Engineering (2007) 82–89
18. Heinis, T., Alonso, G.: Efficient Lineage Tracking For Scientific Workflows. In: Proceedings of the 2008 ACM SIGMOD conference. (2008) 1007–1018
19. Anand, M., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: Proceedings of the 13th International Conference on Extending Database Technology, ACM (2010) 287–298
20. Biton, O., Cohen-Boulakia, S., Davidson, S.: Zoom* userviews: Querying relevant provenance in workflow systems. In: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 1366–1369