A Fine-Grained Workflow Model with Provenance-Aware Security Views

Zhuowei Bao
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania

Tova Milo
Tel Aviv University

Abstract

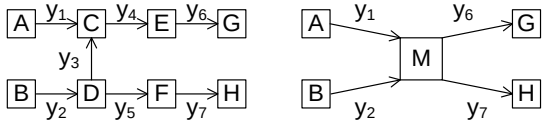
In this paper we propose a *fine-grained* workflow model, based on context-free graph grammars, in which the dependency relation between the inputs and outputs of a module is explicitly specified as a bipartite graph. Using this model, we develop an access control mechanism that supports *provenance-aware* security views. Our security model not only protects sensitive data and modules from unauthorized access, but also provides the flexibility to expose correct or partially correct data dependency relationships within the provenance information.

1 Introduction

Scientific workflows can be complex, and building them from scratch may take a large amount of human effort. It is therefore common to re-use workflows, or portions of workflows, by creating *composite modules* that encapsulate shareable sub-workflows. As observed in [3, 4], composite modules can also be used to create *views* of the provenance information associated with a workflow, showing users the subset of provenance information that is relevant to them and hiding the rest within unexpanded composite module executions.

Views can also be used to hide private information, which may include the intermediate data and modules within a composite module as well as the dependencies between the inputs and outputs of the composite module [4]. In this paper, we focus on the use of views to hide this type of information, and call them *security views*.

As an example, consider the workflow shown in Figure 1(a), where $\{A, B, \ldots, H\}$ denote atomic modules, and $\{y_1, y_2, \ldots, y_7\}$ denote the data flowing between the modules. A security view is constructed by contracting $\{C, D, E, F\}$ into a composite module M (see Figure 1(b)). By doing so, $\{y_3, y_4, y_5\}$ and $\{C, D, E, F\}$ are hidden from unauthorized users. Additionally, the exact dependencies between inputs and output of M are hidden, since without additional information the only assumption that one can make is that every output depends on every input. Thus, unauthorized users will assume that y_7 depends on y_1 which is false.



(a) Full workflow (b) Security view

Figure 1: A provenance-unaware security view

In previous work [6], composite modules that introduce false dependencies between inputs and outputs were considered bad (*unsound*), and were corrected by splitting them into smaller sound ones, so that the dependencies between accessible data (i.e., provenance information) were preserved. In contrast, we argue in this paper that unsound composite modules may be useful for security, so that sensitive data and provenance information are protected from unauthorized access. On the other hand, users who are authorized to access an unsound composite module should be told correct provenance information, e.g., a user who is authorized to access M should be told that y_7 does not depend on y_1.

If a user is shown the full provenance information that they are allowed to access (e.g., an authorized user is shown Figure 1(a)), then clearly they will be able to analyze the data and module dependencies (as given by their allowed view). However, since provenance is large we wish to support efficient techniques for determining such dependencies. (e.g., using labeling schemes [1, 2]). In particular such techniques should support the following features: 1) multiple input and output data for modules; 2) hiding exact dependencies between inputs and outputs of composite modules; and 3) multiple security views over the same workflow. The problem of efficient access

control with security views has been studied in the context of XML [7, 5]. However, our problem is more challenging since workflows have general DAG structure.

The rest of this paper is organized as follows. We present a *fine-grained* workflow model based on context-free graph grammars in Section 2. The graph grammar replaces a bipartite graph representing the dependency relation between inputs and outputs of a module with a graph with the same number of inputs and outputs, but a possibly different dependency relation. [1] On top of this model, we develop in Section 3 an access control mechanism that supports *provenance-aware* security views. Section 4 presents some initial thoughts on the problem of efficiently querying data provenance on security views. We conclude with Section 5.

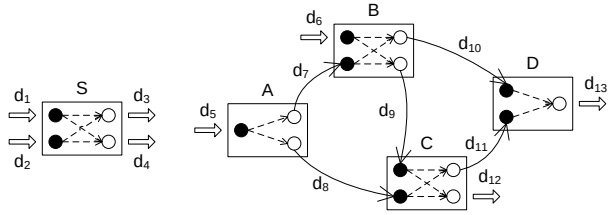
2 Fine-Grained Workflow Model

In this section, we present the fine-grained workflow model. The main idea is to define a workflow as a context-free graph grammar G using graph-based production rules, in such a way that the graph language $L(G)$ corresponds exactly to the set of all possible executions of this workflow (a.k.a. *workflow runs*). We start by describing basic workflow components and then introduce our context-free graph grammar.

2.1 Modules and Simple Workflows

The basic building blocks of a workflow are *modules*. Each module has a fixed set of *input ports* and a fixed set of *output ports*, and performs some data transformation from the inputs to the outputs. The dependency relation between them is specified by a set of *dependency edges*.

Definition 1. A module is a directed bipartite graph $M = (I \cup O, E)$, where I and O are disjoint sets of input ports and output ports, respectively, and $E \subseteq I \times O$ is a set of dependency edges.



(a) A module S (b) A simple workflow W_1

Figure 2: Basic Workflow Components

[1] This differs from the graph grammar in [2], which replaces a vertex with a two-terminal graph.

Example 1. The module S in Figure 2(a) has two input ports and two output ports, which are denoted by solid and hollow circles, respectively. S takes as input two data items, d_1 and d_2 (one per input port), and produces two data items, d_3 and d_4 (one per output port). Dependency edges are denoted by dashed edges.

In principle, a module can be defined as an arbitrary bipartite graph from the set of input ports to the set of output ports. However, certain constraints are natural in practice. For example, every input should be used (i.e. each input port has at least one outgoing dependency edge), and no output should be constant (i.e. every output port must have at least one incoming dependency edge).

A *simple workflow* is a set of modules which are connected by *data edges*. Each data edge connects an output port of one module to an input port of another module, and carries a unique data item that is produced by the former and then consumed by the latter. Note that two restrictions are imposed on the set of data edges in a simple workflow: (1) they are pairwise non-adjacent, that is, every input or output port is incident to at most one data edge; and (2) they do not introduce cycles among the modules. The restrictions are not necessary, but simplify the discussion, as shown in Appendix A.

Definition 2. Given a set $\mathcal{M}$ of modules and a set D of data edges connecting the modules in $\mathcal{M}$, the simple workflow formed by $\mathcal{M}$ and D is the directed acyclic graph $W = (I \cup O, E \cup D)$, where I, O and E are the sets of input ports, output ports and dependency edges, respectively, of all modules in $\mathcal{M}$.

Example 2. The simple workflow W_1 in Figure 2(b) consists of four modules and five data edges. A, B, C and D are module names (not necessarily unique); d_7, d_8, d_9, d_{10} and d_{11} are unique data items flowing on the data edges. To contrast with dependency edges, data edges are drawn as solid edges.

In the rest of this paper, we use the following notation. Given a finite set Σ of modules, Σ^* denotes the set of all simple workflows consisting only of modules chosen from Σ. Given a simple workflow W, I_W and O_W denote the set of input ports and output ports, respectively, of W which are not connected by any data edge. Since each data edge carries a unique data item, we will use the data item name to refer to both the input port and the output port for the data edge, e.g., $I_B = \{d_6, d_7\}$. In addition, we may write a simple workflow as a function over its component modules, e.g., $W_1(A, B, C, D)$.

2.2 Context-Free Graph Grammar

A simple workflow can be abstracted as a *composite module* that is used to form other (more complex) simple workflows. It is captured by a *graph-based production*.

Definition 3. A graph-based production is of form $M \xrightarrow{f} W$ where M is a composite module, W is a simple workflow, and $f : I_M \cup O_M \rightarrow I_W \cup O_W$ is a one-to-one mapping from I_M to I_W and from O_M to O_W.

Note that M and W must have the same interface (i.e., the same number of input ports and output ports), however, they may or may not have the same dependency relation between the input ports and output ports.

Definition 4. A graph-based production $M \xrightarrow{f} W$ is said to be *consistent* if for any input port $i \in I_M$ and any output port $o \in O_M$, o is reachable from i in M if and only if $f(o) \in O_W$ is reachable from $f(i) \in I_W$ in W; otherwise, it is said to be *inconsistent*.

Example 3. Consider a graph-based production

$$S \xrightarrow{f_1} W_1(A, B, C, D)$$

where S and W_1 are shown in Figure 2, and

$$f_1 = \{(d_1, d_6), (d_2, d_5), (d_3, d_{13}), (d_4, d_{12})\}$$

It is easy to check that this production is consistent.

A complex workflow may, in general, involve a nested hierarchy of simple workflows. We therefore model a *workflow specification* as a *context-free graph grammar* which contains a finite set of graph-based productions.

Definition 5. A workflow specification is defined as a context-free graph grammar $G = (\Sigma, \Delta, \mathcal{P}, S)$, where Σ is a finite set of modules, $\Delta \subseteq \Sigma$ is a set of composite modules, $\mathcal{P} = \{M \xrightarrow{f} W \mid M \in \Delta, W \in \Sigma^*\}$ is a finite set of productions, and $S \in \Sigma$ is a start module.

Example 4. Using Figures 2 and 3, we can build the workflow specification $G = (\Sigma, \Delta, \mathcal{P}, S)$, where $\Sigma = \{S, A, B, C, C', D, E, F\}$, $\Delta = \{S, B, C\}$ and $\mathcal{P} = \{r_1, r_2, r_3, r_4\}$ is defined as follows.

$$S \xrightarrow{f_1} W_1(A, B, C, D) \quad (r_1)$$
$$f_1 = \{(d_1, d_6), (d_2, d_5), (d_3, d_{13}), (d_4, d_{12})\}$$
$$B \xrightarrow{f_2} W_2(E, F) \quad (r_2)$$
$$f_2 = \{(d_6, d_{17}), (d_7, d_{14}), (d_{10}, d_{18}), (d_9, d_{15})\}$$
$$C \xrightarrow{f_3} W_1(A, B, C, D) \quad (r_3)$$
$$f_3 = \{(d_9, d_6), (d_8, d_5), (d_{11}, d_{13}), (d_{12}, d_{12})\}$$
$$C \xrightarrow{f_4} W_3(C') \quad (r_4)$$
$$f_4 = \{(d_9, d_{19}), (d_8, d_{20}), (d_{11}, d_{21}), (d_{12}, d_{22})\}$$

An execution of a workflow is generated from the specification $G = (\Sigma, \Delta, \mathcal{P}, S)$ as follows. It begins with the start module S, and applies a sequence of productions in $\mathcal{P}$ to replace all composite modules with the corresponding simple workflows. For recursive productions, multiple instances of one module may be created.

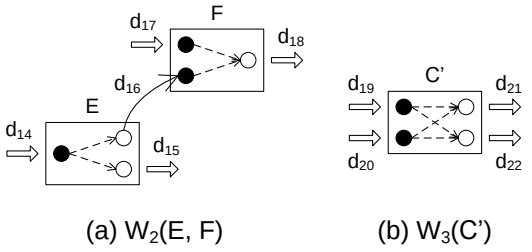


Figure 3: Other simple workflows

To formally define the execution, we first explain the graph derivation. Consider a context-free graph grammar $G = (\Sigma, \Delta, \mathcal{P}, S)$. Given two simple workflows $W_1, W_2 \in \Sigma^*$, W_2 is said to be *directly derived from* W_1 with respect to G, denoted by $W_1 \Rightarrow_G W_2$, if there is a production $M \xrightarrow{f} W$ in $\mathcal{P}$ such that W_2 can be obtained from W_1 by replacing one composite module M with a simple workflow W. Note that the replacement is unambiguous given the one-to-one mapping f. Let $\Rightarrow_G^*$ be the reflexive and transitive closure of $\Rightarrow_G$, then W_2 is said to be *derived from* W_1 with respect to G if $W_1 \Rightarrow_G^* W_2$.

The set of (all possible) *workflow runs* with respect to a specification is modeled as the graph language of the corresponding context-free graph grammar. More precisely, it consists of all simple workflows that can be derived from the start module and contain only *atomic* (non-composite) modules.

Definition 6. The set of workflow runs with respect to a workflow specification $G = (\Sigma, \Delta, \mathcal{P}, S)$ is defined as

$$L(G) = \{W \in (\Sigma \setminus \Delta)^* \mid S \Rightarrow_G^* W\}$$

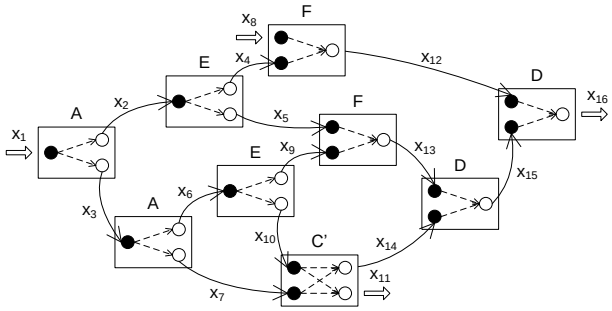


Figure 4: A workflow run $W \in L(G)$

Example 5. One possible workflow run $W \in L(G)$, with respect to the specification G in Example 4, is shown in Figure 4. It can be derived from the start graph S by the sequence of productions r_1, r_2, r_3, r_2, r_4. Since r_3 is recursive, it may be applied any times in a run.

3 Access Control with Security Views

This section presents the access control mechanism which can be used with our workflow model to support provenance-aware security views. First, we introduce the notions of *access control policies* and *security views*.

An access control policy specifies the level of granularity at which a group of users is authorized to see a workflow. In terms of the context-free graph grammar, it is defined as a subset of composite modules that the group of users is allowed to open.

Definition 7. An access control policy for a workflow specification $G = (\Sigma, \Delta, \mathcal{P}, S)$ is a subset $U \subseteq \Delta$ of composite modules.

An access control policy enforces a security view over the workflow specification (abbr. *secure specification*) by restricting the context-free graph grammar to a subset of composite modules so that only the productions for replacing these modules are included.

Definition 8. A security view, enforced by an access control policy U, over a workflow specification $G = (\Sigma, \Delta, \mathcal{P}, S)$ is defined as a context-free graph grammar

$$G_U = (\Sigma, U, \mathcal{P}_U = \{M \xrightarrow{f} W \in \mathcal{P} \mid M \in U\}, S)$$

Example 6. Consider an access control policy $U_1 = \{S, C\}$ for the workflow specification G given in Example 4. The secure specification is denoted by

$$G_{U_1} = (\Sigma, \{S, C\}, \{r_1, r_3, r_4\}, S)$$

Similarly, an access control policy enforces a security view over the workflow run (abbr. *secure run*) by restricting the graph derivation to a subset of productions.

Definition 9. Let $G_U = (\Sigma, U, \mathcal{P}_U, S)$ be a security view, enforced by an access control policy U, over a workflow specification $G = (\Sigma, \Delta, \mathcal{P}, S)$. Let $W \in L(G)$ be a workflow run which is derived from the start module S by a sequence ξ of productions. Then a security view, enforced by U, over W is defined as a workflow run $W_U \in L(G_U)$ which is derived from S by a subsequence of ξ restricted to $\mathcal{P}_U$ [2].

Example 7. The security view W_{U_1}, enforced by the access control policy U_1 (Example 6), over the workflow run $W \in L(G)$ (Example 5), is shown in Figure 5. It is derived from S by a sequence of productions r_1, r_3, r_4.

The overall security model is as follows. Given a workflow run $W \in L(G)$ and an access control policy U, only the module instances and data items which are visible in the secure run W_U are exposed to the users authorized by U. For example, in Figure 5, the instances of modules E and F and the data items x_4 and x_9 are hidden from the users authorized by U_1. Furthermore, the secure specification may also be provided to authorized users to describe the inherent structure of a workflow.

[2]More precisely, the subsequence of ξ contains only productions which are in $\mathcal{P}$ and depend only on productions in $\mathcal{P}$.

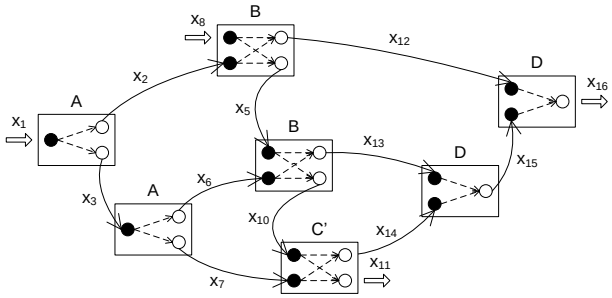


Figure 5: A secure run $W_{U_1} \in L(G_{U_1})$

4 Querying Provenance on Security Views

In future work, we will address the question of how to efficiently query data provenance using security views. The type of queries we consider are those which ask if one data item depends on another. The query must be answered using only the information (i.e., the security view) that is exposed to the user. As a result, users authorized by different policies may get different answers for the same query. It captures the information hiding that one intended when defining the security view.

Example 8. Returning to our example, consider two access control policies $U_1 = \{S, C\}$ and $U_2 = \{S, B, C\}$. The secure runs W_{U_1} and W_{U_2} are shown in Figures 5 and 4, respectively. The answer to whether x_{11} depends on x_8 is "yes" for U_1 because there is a sequence of data items x_8, x_5, x_{10}, x_{11} that are connected by dependency (dashed) edges in Figure 5, but the answer is "no" for U_2.

The fact that the answer to a query depends on the access control policy is caused by inconsistent productions (Def. 4). Given that the composite modules in an access control policy are transparent to authorized users, we can rewrite the dependency relations of these modules without changing the answers to any data dependency queries for this group of users. For some access control policies, it is possible (and beneficial) to do so to make all productions in the secure specification consistent. We believe that rewriting is key to optimize the dependency queries.

Definition 10. An access control policy U is said to be safe, if one can rewrite the dependency relations of composite modules in U so that all productions in $\mathcal{P}_U$ (defined in G_U) are consistent; otherwise, U is unsafe.

Example 9. In the running example, $U_1 = \{S, C\}$ is safe (no rewriting is needed), but $U_2 = \{S, B, C\}$ is unsafe. Clearly, an access control policy that enforces a non-recursive secure specification is always safe. As an example, $U_3 = \{S, B\}$ is safe (by rewriting B and S).

Our initial investigation shows that safety is a crucial property that enables efficient query processing.

5 Conclusions

This paper presents a *fine-grained* workflow model that supports *provenance-aware* security views. Using this model, a workflow can be exposed at different granularity levels to groups of users, depending on their authorized access control policies. More importantly, our provenance-aware security model not only protects sensitive data from unauthorized access, but also provides the flexibility to expose correct or partially correct data provenance. Our future work is to develop techniques for efficiently querying data dependency on security views.

A Loop and Fork

To show that both adjacent data edges and cycles (loops) can be effectively captured by our workflow model (as claimed in Section 2.1), we describe how to encode by graph-based productions two simple forms of *linear self-recursion*, called *loop* and *fork*. They are typical executions in scientific workflows, creating multiple instances of a module *in series* and *in parallel*, respectively.

Generally, a linear self-recursion can be expressed by the following two productions

$$M \xrightarrow{f_1} W_1(M) \quad \text{and} \quad M \xrightarrow{f_2} W_2$$

where $W_1(M)$ denotes a simple workflow in which the module M appears exactly once, and W_2 denotes a simple workflow that does not contain M.

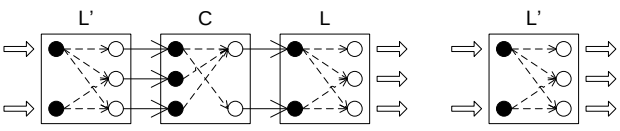


Figure 6: Simple workflows $W_1(L', C, L)$ and $W_2(L')$

Specifically, a loop is expressed by

$$L \xrightarrow{f_1} W_1(L', C, L) \quad \text{and} \quad A \xrightarrow{f_2} W_2(L')$$

where W_1 and W_2 are shown in Figure 6. In general, $L = L'$ is a loop module with m input ports and n output ports. Then C is a connection module with n input ports and m output ports, which ensures that two consecutive iterations are well-connected. Note that although a simple workflow defined in a specification is required to be acyclic (Definition 2), using the above rules, the loops can be effectively captured by our workflow model.

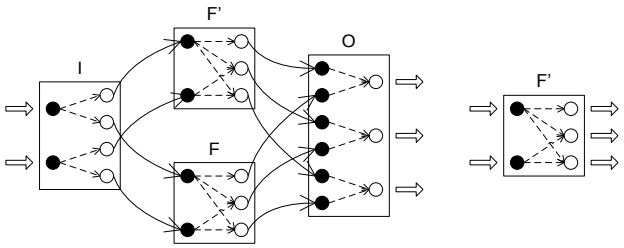


Figure 7: Simple workflows $W_1(I, F', F, O)$, $W_2(F')$

Similarly, a fork is expressed by

$$F \xrightarrow{f_1} W_1(I, F', F, O) \quad \text{and} \quad F \xrightarrow{f_2} W_2(F')$$

where W_1 and W_2 are shown in Figure 7. In general, $F = F'$ is a fork module with m input ports and n output ports. I and O are two auxiliary modules that distribute and collect the input and output data set. They also ensure that W_1 provides the same interface as F (rather than having $2m$ input ports and $2n$ input ports). Also note that such auxiliary modules can be used to capture adjacent data edges (Definition 2). For example, if k data edges are connected to the same input port, we can simply insert an auxiliary module with k input ports and one output port to perform certain aggregation on the k data items before sending them to the next module.

References

[1] BAO, Z., DAVIDSON, S. B., KHANNA, S., AND ROY, S. An optimal labeling scheme for workflow provenance using skeleton labels. In *SIGMOD Conference* (2010), pp. 711–722.

[2] BAO, Z., DAVIDSON, S. B., AND MILO, T. Labeling recursive workflows on-the-fly. In *SIGMOD Conference* (2011).

[3] BITON, O., BOULAKIA, S. C., DAVIDSON, S. B., AND HARA, C. S. Querying and managing provenance through user views in scientific workflows. In *ICDE* (2008), pp. 1072–1081.

[4] BITON, O., DAVIDSON, S. B., KHANNA, S., AND ROY, S. Optimizing user views for workflows. In *ICDT* (2009), pp. 310–323.

[5] FAN, W., CHAN, C. Y., AND GAROFALAKIS, M. N. Secure xml querying with security views. In *SIGMOD Conference* (2004), pp. 587–598.

[6] SUN, P., LIU, Z., DAVIDSON, S. B., AND CHEN, Y. Detecting and resolving unsound workflow views for correct provenance analysis. In *SIGMOD Conference* (2009), pp. 549–562.

[7] YU, T., SRIVASTAVA, D., LAKSHMANAN, L. V. S., AND JAGADISH, H. V. Compressed accessibility map: Efficient access control for xml. In *VLDB* (2002), pp. 478–489.