A Provenance-aware Access Control Framework with Typed Provenance

Lianshan Sun, Jaehong Park, Dang Nguyen, and Ravi Sandhu

Abstract—Provenance is a directed graph that captures historical information about data items in Provenance-Aware Systems (PAS). A variety of access control models and policy languages specific to PAS have been recently discussed in literature. However, it is still not clear how to efficiently specify provenance-aware access control policies and how to effectively enforce these policies with respect to complex provenance graph that can only be captured at run-time. To this end, we design and implement a provenance-aware access control framework with a layered architecture that features an abstract layer, including a Typed Provenance Model (TPM) and a set of TPM interpreters. TPM includes a set of abstract provenance types enabling efficient specification of provenance-aware policies. New provenance types can be composed of extant ones for specifying new policies. TPM interpreters can be integrated to enable the policy enforcement with respect to provenance graphs in different physical representations. By treating provenance types as special attributes, the proposed framework enables an adoption of provenance-aware access control in existing attribute-based access control frameworks, such as XACML-compliant ones. We implement the proposed framework by extending SUN's XACML implementation and show that it facilitates the specification of provenance-aware policies in XACML with minor extensions. We also analyze the performance of the proposed framework.

Index Terms—Typed Provenance Model, Provenance-aware Systems, Access control Framework, OPM, PROV-DM, XACML.

1 Introduction

Access control systems are common components in modern multi-user software systems. An access control system mediates a request to resources and determines whether the request should be granted or denied according to given policies [1], [2]. These policies are usually specified in policy languages [3] under the guidance of access control models [2].

Provenance is information about entities, activities, and people involved in producing a piece of data or thing. In the last decade we have seen the emergence of provenance-aware systems (PAS), which generate, store, process, and disseminate provenance of data items in domains such as scientific workflow, intelligence, and healthcare systems [4], [5], [6]. Provenance can be used to verify trustworthiness of data items or to reproduce the experiment results [7], [8].

Multi-user PAS needs access control facilities to protect not only normal data items but their provenance [9]. Provenance impacts access control in at least two ways. First, provenance access control (PAC) is required to protect sensitive provenance [10]. Second, provenance-based access control (PBAC) can be used to adjudicate access requests to sensitive resources [11], [12]. Note that we call both PAC policies and PBAC policies as provenance-aware policies.

Provenance captured inside a PAS differs from traditional data items and meta-data in that it is an immutable directed graph incrementally captured at run-time. Nodes of a provenance graph denote entities, activities, and people that are involved in producing a piece of data. Edges of a provenance graph denote causality dependencies between two nodes [7]. Provenance subgraphs with certain path patterns may reveal some meaningful information that should be protected or can be used for access control decisions [11], [13]. Extant access control models, policy languages, enforcement infrastructures, as well as policy authoring methodologies and tools cannot be straightforwardly adopted to accommodate provenance-aware access control in PAS [9], [14], [15].

To this end, researchers have presented several access control models [10], [11] as well as several policy languages [13], [16] for either PAC or PBAC. However, these models, languages and corresponding enforcement infrastructures do not provide a flexible and efficient way to specify and enforce various application-specific provenance-aware policies.

In order to specify provenance-aware policies in existing policy languages, policy specifiers have to understand security requirements at conceptual level as well as the complex provenance graph at implementation level. It is difficult to efficiently specify complex provenance queries to identify provenance subgraphs with application-specific semantics. These semantics are necessary for defining application-specific provenance-aware policies [13], [17].

Note that some researchers have identified the difficulty of specifying policies using complex provenance

- *L. Sun is with Shaanxi Univ. of Science and Technology, China; E-mail: sunlianshan@gmail.com;*
- *J. Park (**Corresponding author, jae.park@utsa.edu**), D. Nguren and R. Sandhu are with Univ. of Texas at San Antonio, USA; E-mail: dnguyen@cs.utsa.edu, ravi.sandhu@utsa.edu.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2410793, IEEE Transactions on Dependable and Secure Computing

graph. For example, Cadenhead et al. adopted several fixed (application-independent) provenance query templates, such as *why-query* and *where-query*, to reduce the efforts of specifying provenance-aware policies. However, their solution suffers from the inflexibility or inefficiency of specifying various application-specific provenance semantics [13].

Furthermore, extant enforcement infrastructures that come along with access control models and policy languages specific to PAS [11], [13], [16] are either conceptual architectures or just prototypes that are developed without considering engineering issues, such as facilitating efficient and flexible specification of access control policies, ensuring compatibility with current practices, and shielding the heterogeneity of underlying provenance stores.

To address these issues, we design and implement a provenance-aware access control framework with a layered architecture, which supports both PBAC and PAC [11]. The proposed architecture features an abstract layer, including a Typed Provenance Model (TPM) and a set of TPM interpreters.

TPM includes a set of abstract provenance types that can be used to efficiently specify provenance-aware policies during PAS development. Extant provenance types can be used to flexibly compose new ones to capture new provenance semantics that are necessary for enforcing security requirements. TPM also shields the complexity and heterogeneity of the underlying provenance graphs. So the proposed framework is extensible to work with provenance repositories in different physical representations by introducing appropriate TPM interpreters.

The proposed framework can enforce the policies specified using provenance types by introducing appropriate TPM interpreters. It treats *provenance-type* as a special *attribute* whose values are provenance types that are further processed by TPM interpreters. So it enables easy adoption of provenance-aware access control in PAS that deployed attribute-based access control frameworks, such as XACML-compliant ones. As a proof of concept, we implement the proposed framework by extending SUN's XACML implementation. We show that it accommodates the specification of provenance-aware policies in XACML with minor extensions and we also analyze its performance.

The rest of this paper is organized as follows. Section 2 and section 3 introduce PAS and provenance-aware access control respectively as preliminaries. Section 4 elaborates our goals and corresponding tactics. Section 5 presents the typed provenance model as the basis of designing the target framework. Section 6 presents a layered architecture of the target framework. Section 7 implements the framework and evaluates its performance and computability with current practices. Section 8 discusses related work and section 9 concludes this paper and envisions our future work.

2 Provenance-aware Systems

This section introduces basic concepts of PAS which influence the decision making during the construction of the provenance-aware access control framework.

2.1 Basics of PAS

A PAS mainly includes three components for provenance management: a capture mechanism, a representation model, and an infrastructure for storage, access and queries [17]. The capture mechanism collects provenance which complies with a given representation model. The underlying infrastructure stores the collected provenance and executes queries on provenance stores.

The capture mechanisms. Generally there are two provenance collection strategies, the observed strategy and the disclosed strategy [18]. The observed strategy requires operating systems to continuously collect provenance about running processes, their inputs and outputs [19]. The disclosed strategy requires adapted applications to collect provenance as designed by software architects [20], [21]. Users sometimes need to manually declare provenance when it cannot be captured by the system or application [11], [18]. Most PAS adopt the disclosed strategy because observed provenance is application independent and difficult to be used to answer application-specific questions [18].

The representation model. The collected provenance is often documented according to a given provenance data model, such as Open Provenance Model (OPM) [7], PROV-DM [8], or some proprietary provenance model [17]. OPM captures provenance as causality dependencies among different entities and enables provenance interoperability across systems [7]. PROV-DM [8] is the newest variant of OPM and is a member of the provenance specification family from W3C for provenance inter-operability across web-based applications. Although neither OPM nor PROV-DM has been universally accepted, their intersection does capture a common consensus as reviewed in section 2.2.

The infrastructure. The collected provenance is stored in a provenance repository in certain physical representations (storage models), ranging from specialized Semantic Web Languages and XML dialects stored as files to tuples stored in relational database tables [17]. A query engine is usually specific to a storage model. Hence, users have to write queries in languages specific to the storage model, such as SQL [22], Prolog [21], or SPARQL [23]. However, these general languages were not designed specifically for provenance. It is awkward and complex for users to write queries on provenance in these languages [17]. Even queries that are specified in a language designed specifically for provenance, such as OPQL [24] are likely to be too complicated for many users because provenance contains structural information represented as a graph [17]. For example, a policy language

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2410793, IEEE Transactions on Dependable and Secure Computing

that is an XML dialect requires specification of regular expressions in provenance-aware policies to dynamically identify sensitive provenance subgraphs [13].

Note that a provenance-aware access control framework is a subsystem of a PAS and is supposed to use the provenance graph that has been collected by existing collection mechanisms in PAS according to a given provenance representation model. However, the observed provenance tends to include too many details that policy specifiers can neither understand nor use in defining provenance-aware policies according to security requirements. It may also be recorded in inappropriate granularity for the purpose of defining policies. Policy specifiers need a flexible and efficient way to refer to provenance to enable efficient specification of provenance-aware access control policies.

2.2 Basic Provenance Model

This section introduces the basic provenance model that is used as the basis of our target framework. It is mainly the core structure of PROV-DM.

The basic provenance model shown in Figure 1-a includes three elements and seven relationships (or dependencies) among elements. Elements are entities (artifacts in OPM), activities (processes in OPM), and agents. In PAS, entities are snapshots of data objects at run-time, activities are processes that may take as inputs some artifacts and may produce other artifacts as outputs, and agents are special entities representing users or organizations that influence a process. Dependencies are causality relationships between any two elements (except from an agent to an entity or an activity because these have no practical semantics). Note that the core structure can be extended to include subtypes of core elements and dependencies to capture application-specific casuality semantics [8].

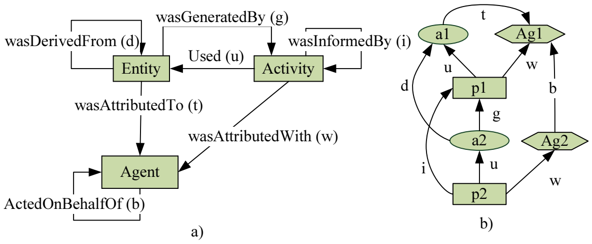

Fig. 1. a) The Core Structure of PROV-DM; b) An Example of Provenance Graph.

Most dependency types in Figure 1-a are literally comprehensible. Note that *wasAttributedTo* indicates that *Entity* was owned, processed, influenced by *Agent* while *wasAttributedWith* indicates that *Activity* was controlled or influenced by *Agent*. The name of each dependency in Figure 1-a has an abbreviation in brackets, such as *used* with its abbreviation '*u*'. Each dependency can be denoted as $R(n, m)$, where R denotes its short name such as '*u*' or '*g*', n the effect element, and m the cause element.

Figure 1-b shows a provenance graph with nodes and edges instantiated from elements and relationships in Figure 1-a. For example, the edge $u(p2, a2)$ denotes that an activity $p2$ used an entity $a2$.

Besides causality semantics denoted by individual edges, some application specific semantics could also be inferred from some paths in a provenance graph. For example, a path $u(p2, a2) \cdot g(a2, p1) \cdot u(p1, a1)$ indicates that the behavior of the activity $p2$ might be influenced by the entity $a1$. Note that neither OPM nor PROV-DM guarantee that every path in a provenance graph is semantically meaningful [7], [25]. However, some paths do reveal provenance semantics that could be used in specifying provenance-aware access control policies. We will discuss how to capture the meaningful paths in section 5.

3 Provenance-aware Access Control

This section introduces basic notions of PAC, P-BAC, and their common requirements on grouping provenance for efficiently specifying provenance-aware policies, and then discusses why they should be and how they can be aligned with the generic Attribute-Based Access Control (ABAC).[1]

3.1 PAC, PBAC, and their Common Foundation

There are at least two categories of provenance-aware access control, provenance access control (PAC) [10] and provenance-based access control (PBAC) [11]. PAC aims at protecting sensitive provenance from unauthorized access, while PBAC aims at adjudicating access requests to sensitive resources (including provenance) by using provenance as a decision factor.

Currently, there are no full-fledged access control models for PAC [10] even though their necessity has been well discussed in literature [9], [10], [14]. Instead, researchers have presented policy languages and corresponding enforcement architectures for PAC [13], [16]. These languages are usually results of extending XACML to incorporate provenance queries inside access control policies. The query results are sensitive provenance subgraphs to be protected. Various grouping mechanisms have been used to identify sensitive provenance subgraphs for defining PAC policies, for example statically pre-defined groups [27] and dynamically computed groups defined by regular expressions of edges in provenance graph [13].

Park et al. proposed a family of PBAC models [11]. A PBAC policy has some provenance related predicates as its decision part. Park et al. also adopted regular expressions to dynamically group provenance used in specifying PBAC policies. Furthermore, they introduced the concept of *Dependency Name or (named*

[1] Here we assume that ABAC has been adopted by PAS as a generic model for provenance-unaware policies, such as traditional DAC policies, MAC policies, and RBAC policies [26].

dependency) to refer to a regular expression that identifies one or more paths of a provenance graph. Park et al. defined a PBAC policy language in a context-free grammar but did not discuss how to specify PBAC policies in XACML-compatible languages.

In both PAC and PBAC, some provenance subgraphs should be grouped as a single unit that carries application-specific semantics in order to be protected as sensitive resources or to be used to adjudicate access requests. Dependency Name is a dynamic grouping mechanism that can be used to enable simple specification of complex queries on a provenance graph. Each dependency name is a query template that can be applied to a starting node to query its antecedents and even descendants in a provenance graph. Here, the notion of dependency name is common foundation for efficient specification of both PAC and PBAC policies because dependency names could be defined at first and then used later in multiple places by different policy specifiers [28]. The dependency type introduced in section 5 is an extension and formalization of *Dependency Name*.

Note that PAS may include user-declared provenance that could be modifiable and somewhat less trustworthy being user asserted. User-declared provenance would require PAC to consider how to protect it from being illegally modified and PBAC to determine how much to trust it for purpose of adjudicating access requests. For simplicity, we assume there is no user-declared provenance, so this paper only considers query operations on immutable system-captured provenance which is common to both PAC and PBAC.

3.2 Alignment of Both PAC and PBAC in ABAC

In real settings, provenance-aware policies are likely to be applied in conjunction with some forms of attribute-based policies since provenance alone usually are not enough to be used for access control decision. For example, consider a policy that a user can grade a homework and see its owner if and only if the user's position is a professor and the user did not previously review the homework. Here, we can view *position* as a general attribute that could have a string of *'professor'* as its value. The facts that both *its owners* and *the user did not previously review the homework* are provenance items involved in the policy. So it is desirable to specify and enforce both provenance-aware policies and generic attribute-based policies in a unified manner.

Generally, each attribute-based policy can be defined as a set of predicates on attributes of either subjects, objects, or environments [3], [29], [30]. Figure 2 shows the XACML architecture for ABAC. The request from an access requester is intercepted by Policy Enforcement Point (PEP) and forwarded to Policy Decision Point (PDP). PDP then evaluates the request according to given policies from Policy Administration Point (PAP) by querying values of attributes of subjects, objects, and environments from Policy Information Point (PIP). PDP then returns the evaluation result to PEP. PEP grants or denies the request and triggers the obligation services when necessary. Note that the context handler is responsible for forwarding and translating requests and responses in different representations among components.

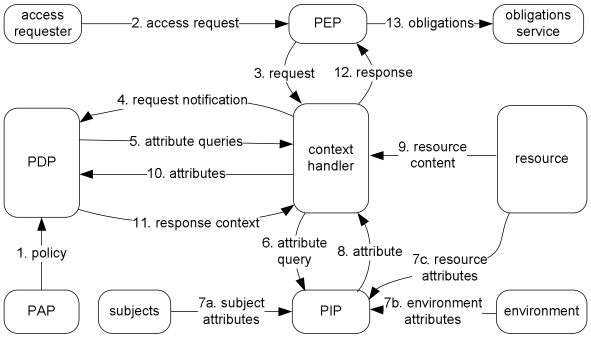

Fig. 2. The XACML Architecture [3]

From ABAC point of view, a PAC policy is a policy that has provenance as the requested resources and a PBAC policy is a policy that comprises predicates on provenance about either the requesting subject or the requested objects. Ideally, if we have special attributes whose values are meaningful units of provenance, then we would be able to specify and enforce provenance-aware policies and other policies in a unified manner.

However, most traditional attributes used in the attribute-based policies are of simple data types, such as *String* and *Integer*. Their values are usually easy to be stored and retrieved to/from databases, and to be compared to literal values. Things get much more complex if we try to introduce provenance attributes. First, it is difficult to specify literal provenance values.[2] Second, it is difficult for policy specifiers to flexibly define appropriate provenance attributes of a subject or an object to denote different provenance semantics. Third, it is difficult for the enforcement framework to efficiently validate predicates on provenance attributes and literal provenance values.

Based on the idea of *Dependency Names* [11], we introduce the concept of provenance types in section 5. If we view each provenance type as a possible value of a special attribute *provenance-type* of a subject or an object, and develop mechanisms to extract a provenance subgraph from a provenance graph by taking as input the provenance type which is a query template and the subject or the object which is the starting node that is fed into the query template, then it is feasible to specify provenance-aware policies as generic attribute-based policies. Consequently, tradi-

[2] Besides individual nodes and edges, some of them could be provenance subgraphs with meaningful semantics.

tional attribute-based access control frameworks, such as XACML-compliant ones, can be adapted to interpret provenance types for enforcing the provenance-aware policies specified using provenance types.

4 Goals and Tactics

In this section, we first identify high-level goals of our provenance-aware access control framework. To address these goals, we further identify four specific tactics.

G1: *Efficiency of specifying provenance-aware policies.* Policy specifiers are not likely to have enough knowledge and skills to use complex provenance graphs. So it is imperative to introduce appropriate provenance abstractions for efficiently specifying provenance-aware policies, especially, during PAS development.

G2: *Adaptability to changes of security requirements.* New security requirements keep emerging in software development and even in software operation. So it is imperative to enable policy specifiers to flexibly define new provenance abstractions that will be used to implement new security requirements as provenance-aware policies.

G3: *Extensibility to work with provenance repositories in different physical representations.* The proposed framework should be extensible to enforce provenance-aware policies with respect to provenance in different physical representations.

G4: *Performance of enforcing provenance-aware policies at run-time.* Enforcement of provenance-aware policies involves queries on a provenance graph, which could be much more time-consuming than queries on traditional attributes organized in linear structure. It is important to ensure that the proposed framework will not introduce significant performance overhead.

G5: *Compatibility with generic attribute-based solutions.*

G51: The policy language for provenance-aware policies should be compatible with that for generic attribute-based policies.

G52: The enforcement framework for provenance-aware policies should be compatible with the generic ABAC framework.

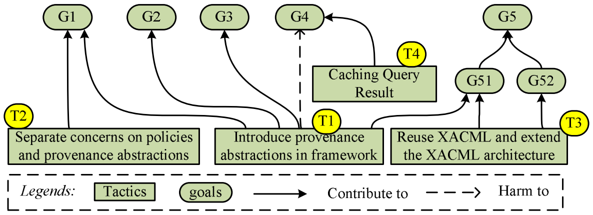
G1
G2
G3
G4
G5
T4
Caching Query Result
G51
G52
T2
T1
T3
Separate concerns on policies and provenance abstractions
Introduce provenance abstractions in framework
Reuse XACML and extend the XACML architecture
Legends:
Tactics
goals
Contribute to
Harm to

Fig. 3. Goals and tactics contributing to goals

To address these goals, below we identified four tactics that can be applied to the framework. Figure 3 shows the identified tactics that can help achieving the goals.

T1: Introducing an abstract layer of provenance can free policy specifiers from directly handling complex provenance graphs, shield the heterogeneity of different provenance storage models, enable the specification of provenance-aware policies at development time, improve the efficiency of policy specification and enable the enforcement of policies defined in provenance abstractions. However interpreting provenance abstractions into provenance subgraphs will inevitably introduce additional performance overhead.

T2: Separating concerns on policies authoring and those on provenance modeling can facilitate efficient policy specification. Policy specifiers can define policies using available provenance abstractions that were defined by software architects during PAS development. Policy specifiers can also define and evolve provenance abstractions for specifying new access control policies together with software architects or by themselves.

T3: Reusing XACML and extending the XACML architecture is helpful for achieving the goal of high compatibility. It allows us to specify and enforce both provenance-aware policies and provenance-unaware policies in a unified manner.

T4: Caching results of frequent queries on a specific starting node could be useful in mitigating performance overhead introduced by querying complex provenance graph.

In this paper, we applied the first three tactics in our framework. Tactic 4 is discussed briefly but not implemented in the prototype. This means our framework partially achieves the identified goals except G4.

5 Typed Provenance Model

We have discussed possible benefits of introducing an abstract layer of provenance between the consumers of provenance (such as PDP and policy specifiers) and the complex provenance graph. This section first introduces a running example, then introduces the overall idea of the typed provenance model (TPM) and its main elements and relationships among multiple elements. Note that we interchangeably use the term TPM either to denote the application-independent meta-model for modeling provenance abstractions or to denote a specific model instance of a PAS application in the rest of this paper. Note that the way it is used is similar to the way, in which the term *class model* is used in the object-oriented methodology.

5.1 A Running Example

A homework grading subsystem (HGS) is a running example in the rest of this paper. We assume that HGS has a simple role-based subsystem deployed and can authenticate a user as either a *Student* or a *Professor*.

- A student can upload, replace, submit, and view grade of her own homework.
- A homework can be submitted only once but can be replaced many times before submission.
- A professor or a student on behalf of a professor can review a submitted and ungraded homework if she did not own or review it before, and if it has been reviewed less than three times.
- A user can revise a review if she created the review and the reviewed homework is not graded.
- A professor can grade a homework if it has been reviewed at least two times.
- A student can see how many times her homework has been reviewed.
- A professor who grades a homework can see who its owners and reviewers are, and if any of its reviews was involved in conflict of interests.

5.2 Typed Provenance Model Overview

Provenance is usually viewed as retrospective information[3] about a system and can only be captured after the system starts running [7]. So it is very difficult for developers to define provenance-aware policies during PAS development. As we have argued, an intuitive idea is to introduce an abstract layer of provenance.

In Figure 4, TPM is introduced to bridge the gap between provenance questions in problem space and complex provenance graph in solution space. TPM abstracts underlying provenance graphs of a specific application as element types, dependency types, as well as relationships and constraints among them. On one hand, it enables users to specify resolvable provenance questions by flexibly composing new provenance types using existing ones. On the other hand, it guides provenance collection at run-time.

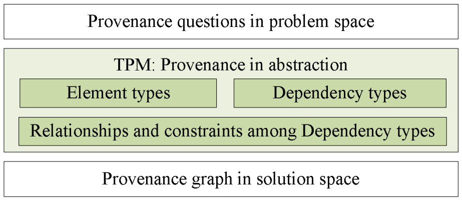
Provenance questions in problem space
TPM: Provenance in abstraction
Element types
Dependency types
Relationships and constraints among Dependency types
Provenance graph in solution space

Fig. 4. Provenance abstraction.

5.2.1 Element Types and Dependency Types

Each provenance type in a TPM identifies a set of nodes, edges, or subgraphs with commonalities in a provenance graph. TPM captures two kinds of provenance types: element types and dependency types.

Each element type in a TPM is an abstract data type that can be instantiated into elements in a provenance graph, including entities, activities, and agents [7]. For example in the HGS, an entity could be instantiated from a class in design model, such as $Homework$. An activity could be instantiated from a business operation in requirements model or a method of a class in design model, such as $upload$ or $submit$. An agent could be instantiated from an acting user of the target system, who plays organizational roles such as *Student* or *Professor*. A class (role) may inherit or include other classes (roles) to form a hierarchy. Provenance could be captured with respect to objects at different levels in a hierarchy and can be efficiently stored if the hierarchical information is suitably utilized [31].

Dependency type is the core concept of TPM. It has roots in the notion of *Dependency Name* [11], [28]. It models a provenance semantic that can be verified by one or more dependency paths (subgraphs) connecting elements of specific types in a provenance graph. A dependency type T defined below is a composition of its name (N), an element type E as effect, and an element type C as cause.

$$T := N(E, C). \tag{1}$$

Here N is a unique name of T and literally indicates semantics of T. Note that T and N are interchangeably used to refer to a dependency type in the rest of this paper. A dependency type can be instantiated into a provenance dependency instance by instantiating both its effect node type and cause node type. For example, a dependency type $ReviewedBy(Homework, \mathit{User})$ can be instantiated into $ReviewedBy(hw_1, u_1)$ to denote that homework hw_1 was reviewed by user u_1.

Note that we can view each dependency type as a provenance question getting either the effect or the cause nodes of a starting node. For example, $ReviewedBy(hw_1)$ returns the set of users who reviewed the homework hw_1 and we have $ReviewedBy(hw_1, u_1) \equiv u_1 \in ReviewedBy(hw_1)$. While N (such as $ReviewedBy$) of a dependency type is similar to a dependency name [11], [28], E and C clarify element types that will be involved in a possible provenance dependency.

TPM includes two kinds of dependency types, the primitive ones that can be instantiated into individual edges of a provenance graph, and the composite ones that can be instantiated into provenance subgraphs rather than individual edges. Each composite type is defined as a composition of primitive types to make it resolvable against an underlying provenance graph.

5.2.2 Primitive Dependency Types

Each primitive dependency type abstracts the semantic of a set of individual edges in a provenance graph. It is a subtype of one of the application-independent dependency types in PROV-DM core structure. It is introduced mainly to accommodate application-specific semantics with its literal name and types of the involved elements.

[3] Some researchers argued that prospective provenance, typically the workflow specification, should also be captured [17].

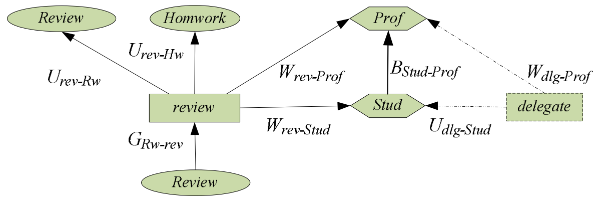

Fig. 5. Primitive provenance types

Figure 5 presents six primitive dependency types specific to HGS. The primitive types that share the same activity type form an activity-centered directed graph. Here, we use multiple copies of the entity type *Review (Rw)* to make the figure easier to read. Figure 5 shows that the activity type *review (rev)* was attributed with *Professor (Prof)* or *Student (Stud)* on behalf of *Prof* and used *Homework (Hw)* and existing *Reviews* of *Hw* as inputs to produce a new *Review* on *Hw*. Note that we consistently capitalize the initial letter of entity type names and agent type names, but leave activity type names as lowercase for readability. In the list below, the inside of [] shown at the end of each dependency type describes its meaning in natural language. If an activity review (p_1) takes as inputs a homework (h_1) and optionally a previous Review (r_1) of h_1, we can instantiate T_1 and T_2 as $U_{rev-Rw}(p_1, r_1)$ and $U_{rev-Hw}(p_1, h_1)$ respectively.

$T_1 := U_{rev-Rw}(review, Review)$[*review* used a *Review*];
$T_2 := U_{rev-Hw}(review, Homework)$[*review* used a *Homework*];
$T_3 := G_{Rw-rev}(Review, review)$[a *Review* was generated by *review*];
$T_4 := W_{rev-Prof}(review, Prof)$[*review* was attributed with *Prof*].
$T_5 := W_{rev-Stud}(review, Stud)$[*review* was attributed with *Stud*].
$T_6 := B_{Stud-Prof}(Stud, Prof)$[*Stud* acted on behalf of *Prof*].

Note that we name a dependency type in a pattern which comprises two parts. The first part is a captial letter such as 'U', 'G', 'W', and 'B', which denotes that the corresponding primitive types are subtypes of '***U**sed*', '*was**G**eneratedBy*', '*wasAttributed**W**ith*', or '*ActedOn**B**ehalfOf*' in PROV-DM respectively. The second part comprises the abbreviated type names of both effect and cause elements when the activity uses or produces only one instance of a specific type. Otherwise, the abbreviated type name of either effect or cause element should be replaced by unique string that denotes the role of the effect or cause element in the context of an activity. For example, a *divide (div)* process used two numbers, one as *dividend* and the other as *divisor*. So two dependency types could be $U_{div-dividend}$ and $U_{div-divisor}$.

Each activity type is generally a method signature in software design model or a business operation in requirements model that can be eventually refined into a set of methods. Each activity type can be instantiated into one or more processes at run-time. A PAS can easily capture primitive dependency types that are directly related to an activity, such as *wasGeneratedBy*, *Used*, and *wasAttributedWith*. These are essential dependency types. Other depencency types such as *wasAttribuedTo*, *wasDerivedFrom*, *wasInformedBy*, and *ActedOnBehalfOf* are less essential in that they need to be deliberately defined by software architects.

For example, suppose HGS has an activity *delegate (dlg)* that was attributed with a professor and used a set of students as its inputs to produce a delegation relation among students and a professor. Here, the software architect has two options to determine the specific primitive provenance types to be documented. One option is to capture two essential primitive types $U_{dlg-Stud}$ and $W_{dlg-Prof}$, and then to semantically construct $B_{Stud-Prof}$ from them. In this case, $B_{Stud-Prof}$ becomes a composite dependency type. The other option is to deliberately define $B_{Stud-Prof}$ as a primitive dependency type while omitting $U_{dlg-Stud}$ and $W_{dlg-Prof}$. Figure 5 visualizes the latter option by showing the provenance type $B_{Stud-Prof}$ in solid line and $U_{dlg-Stud}$ as well as $W_{dlg-Prof}$ in dashed line.

5.2.3 Composite Dependency Types

Each composite dependency type can be defined as a composition of primitive types. It is actually a path pattern that can be instantiated into paths composed of multiple edges instantiated from corresponding primitive dependency types. This section introduces various operators used to form a composite dependency type.

First, the simplest composite pattern is the concatenation of two dependency types. We introduce a binary operator "$\cdot$" to formally indicate the concatenation of two dependency types. A composite dependency type $T := T_i \cdot T_j$ indicates that the effect and the cause node types of T are the effect node type of T_i and the cause node type of T_j respectively, and that the cause node type of T_i is the effect node type of T_j. T means that some instances (t_i, t_j) of T_i and T_j can be concatenated to indicate that the effect node of t_i is to some extent caused by the cause node of t_j. For example, T_7 below indicates that a homework (*Hw*) was uploaded by a student (*Stud*). Note that G_{Hw-up} and $W_{up-Stud}$ are primitive type names and $upload(up)$ is an operation of HGS.

$$\begin{aligned} T_7 &:= UploadedBy(Hw, Stud) \\ &:= G_{Hw-up}(Hw, up) \cdot W_{up-Stud}(up, Stud), \end{aligned}$$

Second, provenance is usually captured in a directed graph. Usually, queries are made to trace provenance graph backward in time. However, users could ask about the effect nodes caused by a given cause node. For example, a reviewer may want to know all existing *Reviews* of a given homework as references of his/her reviewing the homework. To this end, we introduce an unary operator called "inversion" ("$^{-1}$") on dependency types. Letting Rw denote a class of *Reviews* and Hw denote a class of *Homeworks*, we define T_8 and T_9 as follows.

$$T_8 := Reviewof(Rw, Hw) := T_3 \cdot T_2$$
$$T_9 := Reviewof^{-1}(Hw, Rw) := T_8^{-1} := T_2^{-1} \cdot T_3^{-1},$$

Formally, we denote the inversion of a dependency type T as $T^{-1} = N^{-1}(C, E)$. It means that a cause node of type C caused one or more effect nodes of type E in the sense of N. It allows us to traverse a provenance graph from cause to effect.

Third, some provenance semantics could be constructed from paths with unspecified lengths because some activities in PAS might be repeatedly activated many times or optionally activated. For example, an uploaded homework can be optionally replaced many times before final submission. To concisely specify possible path patterns, we can define composite dependency types in regular expressions having the available dependency types as alphabet table. Specifically, we introduce the operator "*", "+", and "?" to define a regular expression over available dependency types. Note that $T*$ means zero or more T concatenated with each other by the operator "$\cdot$"; $T+$ means $T \cdot T*$ and $T?$ means zero or one T. For example, we define a dependency type T_{14} to model dependencies between the submitted homework and its historical versions, where $replace(rep)$ and $submit(sub)$ are two activities in HGS.

$$T_{10} := G_{Hw-rep}(Hw, rep), \quad T_{11} := U_{rep-Hw}(rep, Hw)$$
$$T_{12} := G_{Hw-sub}(Hw, sub), \quad T_{13} := U_{sub-Hw}(sub, Hw)$$
$$T_{14} := SubmisionOf(Hw, Hw) := T_{12} \cdot T_{13} \cdot (T_{10} \cdot T_{11})*,$$

Fourth, some provenance semantics can be validated by multiple paths in a provenance graph either disjunctively or conjunctively. For example, only the owner of a homework can upload, replace, and submit it and a homework cannot be reviewed by its owner due to conflict of interest. To this end, we introduce both the conjunctive and disjunctive operator "$\wedge$" and "$\vee$" to enable the conjunctive or disjunctive composition of multiple fine-grained dependency types into a composite dependency type. We define T_{15} - T_{20} for tracing owners of a homework in HGS as follows.

$$T_{15} := W_{rep-Stud}(rep, Stud)$$
$$T_{16} := ReplacedBy(Hw, Stud) := ((T_{10} \cdot T_{11})*) \cdot T_{10} \cdot \; T_{15}$$
$$T_{17} := SubmittedBy(Hw, Stud) := T_{12} \cdot W_{sub-Stud}(sub, Stud)$$
$$T_{18} := OwnedBy(Hw, Stud) := T_{17} \vee (T_{14}) * \cdot T_{16} \vee (T_{14}) * \cdot T_7$$
$$T_{19} := ReviewedBy(Hw, User) := (T_2^{-1} \cdot T_5) \vee (T_2^{-1} \cdot T_4)$$
$$T_{20} := ReviewedBySelf(Hw, Stud) := T_{18} \wedge T_{19},$$

Here T_{18} is defined as three disjunctive sub-types to denote that the semantic of "*OwnedBy*" can be validated in three ways. T_{17} says that the user who submitted a homework is its owner. $(T_{14}) * \cdot T_{16}$ says that the users who replaced a homework are its owner. $(T_{14}) * \cdot$ says that the user who uploaded a homework is its owner. Note that *Stud* is a subclass of *User*. So T_{20} denotes that a homework was reviewed by its owner, i.e. involving conflict of interests.

The proposed TPM provides high expressiveness in a sense that necessary application-specific provenance semantics are captured as regular-expression based path patterns. It also enables efficient provenance-aware policy specification and enforcement by utilizing named abstraction of the path patterns. However, the discussed TPM specifications and expressions are by no means complete or optimal. it is not our goal to show such a model. In fact, each PAS application may utilize a different set of provenance semantics. Some PAS may need much simplified models while others may need to extend the proposed model as necessary. For example, we can define a subtraction operation "\" among two dependency types. Then we can define a new ownership between students and their homework as $T_{21} := T_{18} \backslash ((T_{14}) * \cdot T_7)$ to indicate that only students who submitted or replaced a homework are the owners of the homework and that the student who uploaded the homework is not. Note that the subtraction operator can be utilized as part of TPM model or it can be also captured as part of policy expression. Having it in the TPM model can provide more expressive graph abstraction and simpler policy specification while requires more complex TPM interpreter. If captured in a policy instead, TPM interpreter could be simpler but more complex policy specification is needed.

6 Architecture Overview

As shown in Figure 6, this section presents a layered architecture of a provenance-aware access control framework as an extension of the XACML architecture in Figure 2. It includes three layers. The top layer and bottom layer include modules reused or extended from either the existing XACML architecture or provenance-aware systems. The middle layer comprises TPM and a set of TPM interpreters and is essential for efficient specification and enforcement of provenance-aware policies.

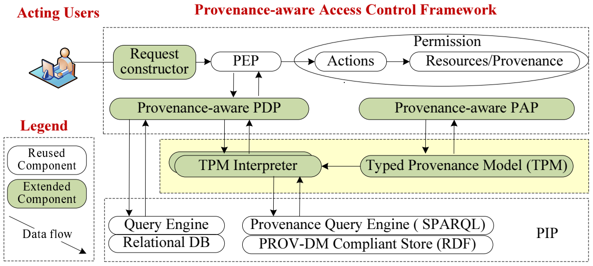

Fig. 6. A Layered Architecture of Provenance-aware Access Control Framework

A typical access control process enforced by the proposed architecture shown in Figure 6 is as follows. An acting user initiates a request in a PAS for permissions (actions against resources). Note that the requested resource might be provenance and the acting user has been authenticated as a subject at run-time by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2410793, IEEE Transactions on Dependable and Secure Computing

the PAS. The request will be intercepted by PEP and then be forwarded to PDP. PDP then evaluates the request against provenance-aware policies from PAP and may query necessary attributes (including provenance) from PIP during evaluation. Note that a provenance-aware policy is defined using dependency types in TPM and all dependency types in a policy will be parsed by a TPM interpreter into provenance queries specific to the underlying provenance store (the provenance-aware PIP). Provenance query engine executes the queries and returns the results to PDP. PDP returns the decision of *'permit'* or *'deny'* to PEP and PEP will then allow or block the request.

Note that the responsibility of PEP in a provenance-aware system is same as that of PEP in general systems. So we do not discuss the implementation of PEP in our framework and assume that we can reuse existing ones. In addition, PEPs in various forms that function at different points in a PAS may send/receive request/response messages in various formats. In X-ACML architecture, a context handler is responsible for communicating and translating these messages in different formats. We assume that all requests and responses are specified in XACML-compliant format and Figure 6 omits the context handler for simplicity. Except for PEP and context handlers, we need to make *Request Constructor, PDP, PAP,* and *PIP* in Figure 2 be provenance-aware to build a provenance-aware access control framework.

Request Constructor: A PAS needs to provide proper functionalities for users to access provenance of data items. So in Figure 6, the *Request Constructor* module needs to be extended to properly construct access requests to provenance. Specifically, it should construct not only the general access request (s, a, o) including a subject (s), an action (a), and an object (o) but an access request including provenance questions. As discussed before, each dependency type applied to a specific starting node is a provenance question with respect to that node. We assume that a PAS allows users to define a new dependency type as a composition of existing ones in a TPM at run-time to query arbitrary meaningful provenance. So an access request to provenance can be formulated as a tuple $(s, a, \langle P, o\rangle)$, where P is a set of dependency types that can be applied to the object o. Note that $\langle P, o\rangle$ that computes all cause or effect nodes of a starting node o in the sense of a dependency type $p \in P$ serves as the requested object in the request. For example, in the HGS, a request $(s, read, \langle\{OwnedBy\}, h_1\rangle)$ denotes that a subject s wants to $read$ the owner of a homework h_1, that is, to get $OwnedBy(h_1)$.

Provenance-aware PDP and PAP: A provenance-aware PDP needs to evaluate an access request against applicable provenance-aware policies retrieved from PAP. The provenance-aware PDP may not only query generic attributes from traditional relational databases but also query provenance from a PROV-DM compliant provenance repository. A provenance-aware PAP is responsible for specifying and storing provenance-aware policies in PAS and responsible for retrieving applicable policies according to the access request. Note that the provenance-aware PDP and PAP can function correctly if and only if the TPM and TPM interpreter in the lower layer functions correctly.

TPM and TPM Interpreter: Both provenance-aware PDP and provenance-aware PAP have one TPM as their common foundation. PAP enables the policy specifiers to specify provenance-aware policies using dependency types in TPM and enables the efficient retrieval of policies that are applicable to a given request. PDP should be able to correctly interpret dependency types in TPM by employing appropriate TPM interpreters. Both TPM and a series of TPM interpreters comprise an abstract layer in the proposed layered architecture to facilitate both PDP and PAP.

In Figure 6, TPM captures a set of application-specific provenance types that can be used to construct access control policies and user provenance questions. A TPM interpreter converts dependency types defined in TPM into provenance queries. These queries can then be executed by a provenance query engine to get required provenance information from the underlying provenance store. TPM interpreters are application-independent and specific to provenance query engines. Note that a provenance query engine (such as SPARQL [32]) is usually specific to a provenance repository in a particular physical representation (such as RDF). Multiple TPM interpreters and provenance query engines should be deployed when multiple provenance stores in different physical representations are used in a PAS.

Provenance-aware PIP: Policies in PAP may refer to various attributes of subjects, objects, actions, and the environment. A PIP provides an interface for querying necessary attributes for evaluating access control policies. Note that a provenance-aware PIP should be able to query provenance from some repositories. Because the infrastructure for storing and querying provenance is an essential building block of a provenance-aware system [17], we argue that the provenance-aware PIP could be reused from the overall database management infrastructure of PAS as shown in the bottom layer in Figure 6. It includes not only relational database management systems and corresponding query engines for attributes used in defining generic attribute-based policies, but also the provenance repositories and corresponding provence query engine for provenance used in defining provenance-aware policies. Note that the key to reuse a specific provenance query engine and corresponding provenance store in our framework is to develop and deploy an appropriate TPM interpreter.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2410793, IEEE Transactions on Dependable and Secure Computing

7 Implementation and Evaluation

In this section, we implement a prototype of our framework and then design experiments to evaluate the compatibility and performance of the prototype.

7.1 Implementation

We extended the *PDP* class in SUN's XACML framework to get a provenance-aware PDP. The extended PDP takes as inputs a provenance-aware access request and a set of available provenance-aware policies. Both of them should be written in XACML format and stored in XML files. For simplicity, we assume that both of them are directly specified in XACML format in the following experiments. So our prototype did not implement request constructor and context handler and simply implemented PAP as a set of policy files specified in XACML.

The TPM in our framework is pre-defined in a two column table including pairs of provenance type names and matching path patterns (composition rule). A TPM interpreter retrieves the matching path pattern of a given dependency type name from TPM table and translates the path pattern into queries specific to a particular query engine, such as SPARQL.

We implemented a SPARQL-specific TPM [32] interpreter as an extension to the *FunctionBase* class in SUN's XACML framework. The TPM interpreter is first registered into the provenance-aware PDP and then invoked by the PDP at the time of each access request evaluation. For examle, in the XACML Policy 2 given in the next subsection, the PDP invokes our SPARQL-specific TPM interpreter class named *"provenance-query-SPARQL"*.

We employed the Apache Jena framework [33] to provide both the RDF-enabled [34] data store for provenance graph and the ARQ query engine for enabling SPARQL queries [32]. In this work, we are using Jena-2.7.4 and the corresponding ARQ package.

7.2 Experiments and Evaluation

We deploy the implemented prototype onto a virtual machine instance which resides in our local Joyent SmartData center. The instance is an Ubuntu 12.10 image with 4GB Memory and a 2.5 GHz quad-core CPU. We design experiments on top of HGS application outlined in previous section to evaluate our framework in terms of its compatibility with XACML and performance overhead under extreme situations. To enable the experiments, we have to implement HGS-specific components, including TPM of HGS, provenance graph of HGS, access requests, and policies of HGS, which are necessary for our experiments besides the deployed application-independent framework. Note that TPM of HGS mainly includes entity types and dependency types introduced in section 5. Provenance graph of HGS is generated as a set of RDF tuples and stored in memory as a Jena model. Both access requests and policies are specified in XACML.

7.2.1 Compatibility Evaluation

This experiment feeds a provenance-aware access request and a provenance-aware policy that are specified in XACML into our framework to show its compatibility with XACML.

The following is an example of a provenance-aware request. A subject with the identity *au3* is requesting who is the owner of a homework *h1* (the provenance). The requested provenance is identified by two attributes, *provenance-type* with value *OwnedBy* and *provenance-startingnode* with value $h1$. *OwnedBy* is a dependency type that queries all owners of a homework and is defined in TPM as a composition of primitive dependency types (see T_{18} in section 5). Note that both Policy 1 and Policy 2 are specified in XACML with an minor extension. We introduce the a function ID *"provenance-query-SPARQL"*, and two special attributes *"provenance-type"* and *"provenace-startingnode"* into standard XACML.

Example 1. A provenance-aware request in XACML

```
<Request>
  <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0
      :subject-category:access-subject">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
        :subject:subject-id" DataType="http://www.w3.org
        /2001/XMLSchema#string">
    <AttributeValue>au3</AttributeValue></Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
        :resource:provenance-type" DataType="http://www.
        w3.org/2001/XMLSchema#string">
    <AttributeValue>OwnedBy</AttributeValue></Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
        :resource:provenance-startingnode" DataType="
        http://www.w3.org/2001/XMLSchema#string">
    <AttributeValue>h1</AttributeValue></Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
        :action:action-id" DataType="http://www.w3.org
        /2001/XMLSchema#string">
    <AttributeValue>readprov</AttributeValue></Attribute>
  </Action>
</Request>
```

The following is an example of a provenance-aware policy that corresponds to the request above. The target section of this policy shows that it is designed for guarding the action *readprov* on a provenance-type *OwnedBy* that could be applied to any instances of *Homework* class. The access rule *HwOwnerRule* is that a user will be allowed to see the owners of a homework only if the user is one of those who have been grading the homework. Note that the following policy explicitly refers to a specific TPM interpreter named *provenance-query-SPARQL*, which takes as inputs a starting node (designated as *h1* specified in the request) and a dependency type (*GradedBy*).

Example 2. A provenance-aware policy in XACML

```
<Policy PolicyId="PACPolicy"     RuleCombiningAlgId="
       urn:oasis:names:tc:xacml:1.0:rule-combining-
       algorithm:ordered-permit-overrides">
<Description>...</Description>
<Target>
<Subjects>     <AnySubject />    </Subjects>
<Resources> <Resource>
```

```
    <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0
        :function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/
        XMLSchema#string">OwnedBy</AttributeValue>
    <ResourceAttributeDesignator AttributeId="
        urn:oasis:names:tc:xacml:1.0:resource:
        provenance-type" DataType="http://www.w3.org
        /2001/XMLSchema#string" />
    </ResourceMatch>
</Resource> </Resources>
<Actions> <Action>
    <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0
        :function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/
        XMLSchema#string">readprov</AttributeValue>
    <ActionAttributeDesignator AttributeId="
        urn:oasis:names:tc:xacml:1.0:action:action-id"
        DataType="http://www.w3.org/2001/XMLSchema#
        string" />
    </ActionMatch>
</Action> </Actions>
</Target>

<Rule RuleId="HwOwnerRule" Effect="Permit">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0
     :function:string-is-in">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0
     :function:string-one-and-only">
<SubjectAttributeDesignator AttributeId="
     urn:oasis:names:tc:xacml:1.0:subject:subject-id"
DataType="http://www.w3.org/2001/XMLSchema#string" />
</Apply>
<Apply FunctionId="provenance-query-SPARQL">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0
     :function:string-one-and-only">
<ResourceAttributeDesignator
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:
     provenance-startingnode"
DataType="http://www.w3.org/2001/XMLSchema#string" />
</Apply>
<AttributeValue DataType="http://www.w3.org/2001/
     XMLSchema#string">GradedBy</AttributeValue>
</Apply>
</Apply>
</Rule>
<Rule RuleId="FinalRule" Effect="Deny" />
</Policy>
```

The experiment shows not only how to specify access request to provenance but also how to refer to a specific TPM interpreter in a provenance-aware policy. It also shows that both provenance-aware policies and generic attribute-based policies can be specified and enforced in a unified manner.

7.2.2 Performance Evaluation

A provenance-aware PDP could receive multiple concurrent requests in real settings. So its performance is critical. We perform experiments to evaluate the time it takes for the PDP instance to completely handle 500 simultaneous access requests. These requests are actually replica of one request, which is almost simultaneously issued 500 times by an agent simulating the user of HGS application. Note that the PDP instance will create an independent thread to serve each request it received. So there would exist multiple threads running simultaneously to serve multiple requests.

Querying complex provenance graph could be a time-consuming task. Different PAS may have provenance graphs with different quantity of edges, width, and depth, which would have influence on the provenance overhead of our framework. On one hand, some provenance information necessary for decision making could be spread out widely in a provenance graph. The outdegree of some nodes in a wide provenance graph could be very high. For example, one homework object can be reviewed by a multitude of reviewers. In a provenance graph, each *review* process is captured as a branch coming out of the homework object. To obtain all reviewers of the object, a query needs to trace through all these branches. It is interesting to evaluate the performance of our framework against a provenance graph with high width.

On the other hand, some provenance information could be spread in depth in a provenance graph. Some paths connecting a cause and an effect in a deep provenance graph could be very long. For example, an uploaded homework object can be replaced multiple times (any times) by its owner. To obtain the original version of a homework, a query needs to trace back through a large number of edges. Similarly, it is interesting to evaluate the performance of our framework against a provenance graph with high depth.

We design different experiments to evaluate the performance overhead of our framework prototype when the size and shape of the provenance graph is in different configurations. Note that we assume that the provenance graph remains unchanged to accurately evaluate the performance overhead of our framework though granting an access request might result in additional information being captured and the underlying provenance graph being changed. Specifically, we proceed to perform the experiments in quantities of 2000, 4000, 6000, 8000, 10000, and 12000 edges traced in evaluating access requests. That means we simulated both wide provenance graphs and deep provenance graph having various edges. In a wide provenance graph, a query may trace from 2000 up to 12000 edges in width. In a deep provenance graph, a query may trace from 2000 up to 12000 edges in depth.

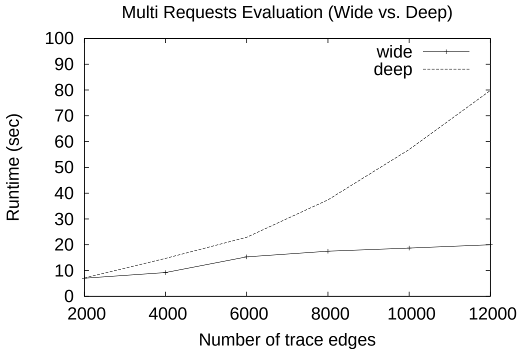

Fig. 7. Throughput evaluation per 500 requests

The results in Figure 7 show that in the width-tracing scenario, the performance overhead increases linearly along with the increase in the number of traced edges. In the depth-tracing scenario, the performance overhead increases still linearly, but at a higher slope. Here, for the most heavy tracing query, we obtain the result of 500 requests per 80 seconds (0.16 second per deep request) in depth-tracing scenario and 500 requests per 20 seconds (0.04 second per wide request) in width-tracing scenario. At the same time, for the lightest tracing query, the result is 500 requests per 7 seconds (0.014 second per deep request) and 500 requests per 7 seconds (0.014 second per wide request). That means that at the lightest tracing scenarios, the performance overhead of both wide-tracing and deep-tracing is very low and hardly differentiable. In fact, according to canonical data structure textbooks, the asymptotic time complexity of traversing a graph is *O(n+e)*, where *n* is the number of nodes and *e* the number of edges in the traversed graph. Our experiments consolidate the theoretical analysis. We believe the heavier runtime increase of the depth-tracing query is due to the SPARQL implementation of query execution that utilizes more recursive calls for each successive process step in depth-tracing scenario than that in width-tracing scenario.

The result of the above analysis only demonstrates the feasibility of our framework when the provenance graph can be fully loaded into memory and its depth and width traces would not exceed specific quantities. When a provenance store grows extremely large in a real application and it cannot be loaded into memory as one Jena model, some queries would involve provenance retrieval from disks, which apparently costs more time. Also note that the performance depends heavily on the underlying provenance query engine. However this is outside of the scope of this paper. With these restrictions in mind, while we think the proposed framework can provide improved performance for example by caching the abstracted graph, we did not provide any further discussion on achieving this. An extensive investigation is required to properly address the performance issues.

8 Related work and discussion

This section first discusses related work related to provenance management systems, provenance model, and secure provenance, and then discusses practical issues of applying our framework in real settings.

8.1 Related Work

Existing literature on provenance was mainly concerned with the conceptual or functional issues of PAS, such as what is provenance in databases [4], [5], [35], why provenance is important for the future [36], how to capture provenance and by who [18], [37], how to efficiently store and query provenance [22], [24], [31], [38], how to communicate provenance across multiple PAS [7], [8], what a provenance-aware system is and how to build it in an engineering manner [20]. Accordingly, a series of provenance managements systems have been built to answer these questions from different perspectives and to different extent. Freire et al. surveyed the important concepts related to provenance management so that potential users can make informed decisions when selecting or designing a provenance solution [17].

One of the most fundamental problems of building a PAS is to clarify its provenance model [6], [7], [8], [16]. A provenance model, such as PROV-DM and OPM, is usually an application or domain independent representation framework which tells the generic types of elements and causality dependencies among those elements that can be captured as provenance of data items. In PROV-DM or OPM, these types are generic enough so that the resulted provenance graph is inter-operable across multiple PAS.

This paper introduced TPM on top of community common consensus on provenance, such as that captured by the core structure of PROV-DM or OPM, to enable the developers to capture application-specific provenance semantics. Note that although PROV-DM and OPM also provides a subtyping mechanism to capture application-specific provenance semantics, TPM provides more flexible composition mechanisms for developers to do that. Developers can define a dependency type in a TPM as not only a subtype of PROV-DM core types but also a composition of available dependency types in a TPM. Each dependency type in a TPM can be instantiated into one or more paths in a provenance graph that denote causality dependencies among elements of specific types, such as *Homework* and *Stud* in HGS.

Furthermore, unlike PROV-DM that is mainly used to shape the query and storage infrastructure of PAS, a TPM that captures application-specific provenance semantics is intended to drive specific PAS development, such as to enable the efficient specification of provenance-aware policies during PAS development. A TPM of specific application is actually its prospective provenance that will be instantiated and captured as retrospective provenance at run-time.

Secure provenance is critical to verify trustworthiness of data items in a PAS [9], [14]. Traditional security models and policy languages are not appropriate for PAS [10], [14]. Correspondingly, the underlying enforcement framework and policy authoring tools which worked well for traditional access control policies would not work well in PAS [3], [39]. To this end, researchers have presented several access control models [10], [11] as well as policy languages [13], [16] for either PAC or PBAC. However, these solutions are not practical for efficiently specifying provenance-aware policies during PAS development and then efficiently enforcing these policies at run-time due to their lack of considering the complexity of provenance graph and several engineering issues.

In contrast, we introduce a novel provenance model–TPM to facilitate the efficient specification of provenance-aware policies during PAS development, and build a provenance-aware access control framework with TPM and a set of TPM interpreters as key components to enforce these policies. TPM enables flexible composition of novel composite types to capture high-level provenance semantics that are involved in emerging security requirements. So our framework enables efficient and flexible specification of provenance-aware policies even for newly identified security requirements. Furthermore, our framework is extensible to work with provenance repositories in different storage models by introducing appropriate TPM interpreters. In addition, we facilitate the compatibility of our framework with attribute-based access control ones, such as XACML-compliant ones, by treating *provenance-type* as a special *attribute* having provenance types as its possible values that will be further processed by TPM interpreters. As a proof of concept, we show that provenance-aware policies can be specified in XACML with minor extensions. Note that Sun et al. have introduced the initial idea of TPM with an emphasis on the engineering process of defining provenance-aware policies using TPM [15].

8.2 Discussion

In order to apply our framework in real settings, several issues need to be carefully considered.

Our framework prototype assumes that PEPs and context handlers in a XACML architecture are available. So developers need to either integrate existing PEPs and context handlers into our framework prototype or build their own ones when these components are not available. Our framework prototype currently only implements a TPM interpreter for RDF-based provenance store. In real settings, new TPM interpreters are likely to be built for other PROV-DM compliant provenance stores. Note that our framework prototype assumes that underlying provenance stores are PROV-DM (or OPM)-compliant. It cannot be directly integrated into a PAS that creates a provenance store following a proprietary provenance model rather than PROV-DM or OPM. Even so, developers could be able to implement the layered architecture shown in Figure 6 in their own settings to get a proprietary framework.

Currently, the proposed framework only considers provenance captured by application systems, but not those observed at operating system level mainly because they hardly reveal any application-specific semantics that are necessary for provenance-aware access control [18]. Our framework provides an application-independent meta-model for TPM and a set of application-independent TPM interpreters. When deployed for a PAS, TPM is designed typically by system/security architects. Specifically, system and security architects could define TPM at first on the basis of system models to drive the PAS development and then security administrator can use them to define provenance-aware policies. Sometimes, a security architect may need to define new dependency types for specifying provenance-aware policies.

A construction of TPM involves many engineering issues, such as who should create TPM, when and how to create and evolve TPM, and how to use TPM to drive PAS development. We have discussed some of these issues in our previous work [15]. A more extensive investigation on these issues is still a must. Note that entities in a TPM can be derived from special elements in system models of PAS, such as UML models, and some primitive dependency types could be automatically derived from the UML models. So the size of TPM of an application should be roughly propositional to the size of UML models of the application. In that sense, we believe that the manual labor of constructing TPM would be not unacceptable.

9 Conclusion

This paper argues that security architects should be able to efficiently specify provenance-aware policies during PAS development, and to have these policies enforced according to provenance graph captured at run-time. This paper achieves these goals by introducing an abstract provenance model, called Typed Provenance Model (TPM) and by designing and implementing a provenance-aware access control framework with a layered architecture. The proposed framework accommodates provenance-aware policies in the same way of accommodating generic attributed-based policies by treating *provenance-type* as a special *attribute*. Furthermore, it is flexible enough in accommodating provenance-aware policies involving provenance in different level of abstraction. It is extensible to work with provenance stores in different physical representations. We implement a prototype for the proposed framework. We then analyze its performance and evaluate its compatibility with XACML. Our future work includes deploying and evaluating the implemented prototype in real settings, exploring and optimizing the performance overhead introduced by provenance query engines, designing more practical engineering methodology to guide the usage of our framework in practice.

Acknowledgments

This work is partially supported by NSF (No. CNS-1111925), NSF of China (No. 61202019), and Shaanxi Provincial Education Department (No. 14JK1098).

References

[1] P. Samarati and S. D. C. d. Vimercati, "Access control: Policies, models, and mechanisms," ser. FOSAD '00. London, UK: Springer-Verlag, 2001, pp. 137–196.

[2] R. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40 –48, sept. 1994.

[3] T. Moses. (2005) eXtensible Access Control Markup Language TC v2.0 (XACML). OASIS.

[4] P. Buneman, S. Khanna, and W. C. Tan, "Data provenance: Some basic issues," ser. FST TCS 2000. London, UK, UK: Springer-Verlag, 2000.

[5] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Foundation and Trends in databases*, vol. 1, no. 4, Apr. 2009.

[6] P. Groth, S. Jiang, S. Miles, S. Munroe, V. Tan, S. Tsasakou, and L. Moreau, "An architecture for provenance systems," University of Southampton, Technical Report, February 2006.

[7] L. Moreau, B. Clifford, J. Freire, and et al., "The open provenance model — core specification (v1.1)," *Future Generation Computer Systems*, December 2009.

[8] K. Belhajjame, R. B'Far, J. Cheney, and et al., "Prov-dm: The prov data model," Tech. Rep., 2012.

[9] R. Hasan, R. Sion, and M. Winslett, "Introducing secure provenance: problems and challenges," ser. StorageSS '07. New York, NY, USA: ACM, 2007, pp. 13–18.

[10] U. Braun and A. Shinnar, "A security model for provenance," Harvard University, Tech. Rep. TR-04-06, Jan 2006.

[11] J. Park, D. Nguyen, and R. Sandhu, "A provenance-based access control model," in *10th Annual Conf. on Privacy, Security and Trust*. IEEE, July 2012.

[12] C. Ringelstein and S. Staab, "Papel: Provenance-aware policy definition and execution," *IEEE Internet Computing*, vol. 15, no. 1, pp. 49–58, Jan. 2011.

[13] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham, "A language for provenance access control," ser. CODASPY '11. New York, NY, USA: ACM, 2011, pp. 133–144.

[14] U. Braun, A. Shinnar, and M. Seltzer, "Secure provenance," in *The 3rd USENIX Workshop on Hot Topics in Security*. Berkeley, CA, USA: USENIX Association, July 2008, pp. 1–5.

[15] L. Sun, J. Park, and R. Sandhu, "Engineering access control policies for provenance-aware systems," ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 285–292.

[16] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han, "An access control language for a general provenance model," ser. SDM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 68–88.

[17] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science and Engg.*, vol. 10, no. 3, pp. 11–21, May 2008.

[18] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *IPAW'06*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 171–183.

[19] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," ser. ATEC'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 4–4.

[20] S. Miles, P. Groth, S. Munroe, and L. Moreau, "Prime: A methodology for developing provenance-aware applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 8:1–8:42, Aug. 2011.

[21] A. Marinho, L. Murta, C. Werner, V. Braganholo, S. M. S. da Cruz, E. S. Ogasawara, and M. Mattoso, "Provmanager: a provenance management system for scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1513–1530, 2012.

[22] R. S. Barga and L. A. Digiampietri, "Automatic capture and efficient storage of e-science experiment provenance," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 5, pp. 419–429, Apr. 2008.

[23] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, "Prov-o: The prov ontology," Tech. Rep., 2012.

[24] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi, "Opql: A first opm-level query language for scientific workflow provenance," ser. SCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 136–143.

[25] S. Dey, S. Köhler, S. Bowers, and B. Ludäscher, "Datalog as a lingua franca for provenance querying and reasoning," in *TaPP'12*, 2012.

[26] X. Jin, R. Krishnan, and R. S. Sandhu, "A unified attribute-based access control model covering DAC, MAC and RBAC," in *DBSec*, 2012, pp. 41–55.

[27] A. Syalim, Y. Hori, and K. Sakurai, "Grouping provenance information to improve efficiency of access control," in *Advances in Infomation Security and Assurance*. Springer, 2009, pp. 51–59.

[28] D. Nguyen, J. Park, and R. Sandhu, "Dependency path patterns as the foundation of access control in provenance-aware systems," in *TaPP'12*, Boston, MA, USA, June 2012.

[29] E. Yuan and J. Tong, "Attributed based access control (abac) for web services," ser. ICWS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 561–569.

[30] J. Y. Halpern and V. Weissman, "Using first-order logic to reason about policies," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 4, pp. 21:1–21:41, Jul. 2008.

[31] A. P. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 993–1006.

[32] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C, Tech. Rep., 2006. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[33] "Apache Jena Framework." [Online]. Available: http://jena.apache.org/about_jena/architecture.html

[34] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, 2004.

[35] W. C. Tan, "Provenance in databases: Past, current, and future." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 3–12, 2007.

[36] J. Cheney, S. Chong, N. Foster, M. I. Seltzer, and S. Vansummeren, "Provenance: a future history," in *OOPSLA Companion*, 2009, pp. 957–964.

[37] J. Frew, D. Metzger, and P. Slaughter, "Automatic capture and reconstruction of computational provenance," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 5, pp. 485–496, Apr. 2008.

[38] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara, "Querying and managing provenance through user views in scientific workflows," ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1072–1081.

[39] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong, "Expandable grids for visualizing and authoring computer security policies," ser. CHI '08. NY, USA: ACM, 2008, pp. 1473–1482.

Lianshan Sun received his PhD degree in software engineering from Peking University. He is currently an associate professor at Shaanxi University of Science and Technology. His research interests include methodologies, infrastructures, and tools on engineering secure software system in the whole software development life-cycle. He is currently focusing on infrastructures for engineering secure provenance-aware systems.

Jaehong Park received the PhD degree in information technology from George Mason University. He is currently a research associate professor at the Institute for Cyber Security, University of Texas at San Antonio. His research interests include data and application security and privacy, access and usage control, cloud computing security, secure provenance and social computing. He has pioneered an area of usage control.

Dang Nguyen is currently a PhD candidate at University of Texas at San Antonio where he received a BS (09) and MS (2013) in Computer Science. His main area of interest is on the application and security foundations of provenance data in multi-tenant cloud environment. He is currently working on provenance-based access control mechanisms in OpenStack platforms.

Ravi Sandhu is founding Executive Director of the Institute for Cyber Security at the University of Texas San Antonio, and holds an Endowed Chair. He is an ACM, IEEE and AAAS Fellow and inventor on 29 patents. He is past Editor-in-Chief of the IEEE Transactions on Dependable and Secure Computing, past founding Editor-in-Chief of ACM Transactions on Information and System Security and a past Chair of ACM SIGSAC. He founded ACM CCS, SACMAT and CODASPY, and has been a leader in numerous other security conferences. His research has focused on security models and architectures, including the seminal role-based access control model. His papers have accumulated over 26,000 Google Scholar citations, including over 6,400 citations for the RBAC96 paper.