Engineering Access Control Policies for Provenance-aware Systems *

Lianshan Sun
Dept. of Computer Science
Shaanxi Univ. of Sci. & Tech.
Xi'an, Shaanxi, China.
sunlianshan@gmail.com

Jaehong Park
Insitute for Cyber Security
Univ. of Texas at San Antonio
San Antonio, TX, USA
jae.park@utsa.edu

Ravi Sandhu
Insitute for Cyber Security
Univ. of Texas at San Antonio
San Antonio, TX, USA
ravi.sandhu@utsa.edu

ABSTRACT

Provenance is meta-data about how data items become what they are. A variety of provenance-aware access control models and policy languages have been recently discussed in the literature. However, the issue of eliciting access control requirements related to provenance and of elaborating them as provenance-aware access control policies (ACPs) has received much less attention. This paper explores the approach to engineering provenance-aware ACPs since the beginning of software development. Specifically, this paper introduces a typed provenance model (TPM) to abstract complex provenance graph and presents a TPM-centric process for identification, specification, and refinement of provenance-aware ACPs. We illustrate this process by means of a homework grading system.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Representation, Methodologies*; D.4.6 [**Security and Protection**]: Access Controls

Keywords

Provenance; typed provenance model; provenance-aware access control policy; PAC; PBAC; TPM; OPM

1. INTRODUCTION

Provenance captures the origins and processes by which a data item became what it is [3]. In the last decade we have seen the emergence of provenance-aware systems (PAS), which generate, store, process, and disseminate provenance to improve trustworthiness of data items in domains such as scientific workflow, intelligence, and healthcare systems [7].

Provenance itself must be securely protected when it is used to verify trustworthiness and integrity of data items in PAS [9]. However, provenance differs from traditional data items and meta-data in that it is an immutable directed graph incrementally captured at run-time. We refer to the provenance in a PAS as a provenance graph. It includes nodes for artifacts (data objects), processes, or agents involved in producing a piece of data and edges for causality dependencies (provenance dependencies) among nodes [12]. Note that a sub-graph of a provenance graph as a unit may show meaningful provenance semantic and could be treated as sensitive resources or be used to adjudicate access requests [4, 15]. Traditional access control models, policy languages, policy authoring tools, enforcement infrastructures, as well as methodologies for engineering access control policies can not be straightforwardly adopted for provenance-aware access control [2, 9].

Researchers have recently proposed a variety of provenance-aware access control models [1, 2, 9, 15] and corresponding policy languages [4, 14]. Provenance could affect access control in at least two ways. The first is the so-called provenance access control or PAC, where provenance itself could be sensitive and should be properly protected [1]. Note that data items with provenance do not have to be sensitive when their provenance is sensitive. For example, provenance of an insensitive report, such as which agent had authored a report and how, could be sensitive. The second is the provenance-based access control or PBAC [15], where provenance as the immutable history about data items can be used to adjudicate access requests on the data items. For example, in a homework grading system, a professor can grade homework of her students if and only if the homework was not graded before and has been reviewed at least two times.

However, provenance-aware access control policies (ACPs), including PAC policies and PBAC policies, could be very complex due to the complexity of provenance. Even with the availability of provenance-aware access control models and policy languages, several practical issues need to be further examined and corresponding solutions need to be developed for engineering provenance-aware ACPs.

First, provenance is usually very complex and hard to be understood in a semantically meaningful way. Also, while existing policy languages [4, 15] assume the availability of some form of provenance graphs [12], the provenance graph is usually not available when eliciting security requirements and defining provenance-aware ACPs at the beginning of PAS development [13, 15]. Therefore, we suggest provenance should be modeled in abstractions, which are meaningful enough and readily available for policy authors to efficiently define provenance-aware ACPs. Recently, Park et al pro-

*This work is partially supported by NSF (No. CNS-1111925), NSF of China (No. 61202019), and SUST Foundation (No. BJ09-13).

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

pose to construct provenance-aware ACPs from the named abstract dependency path patterns of a provenance graph, called dependency names [13, 15]. However, the nature of abstracted dependency names, relationships among them, and the manner of modeling and using them to construct provenance-aware ACPs should be further studied.

Second, some access control requirements that can be implemented as provenance-aware ACPs are originally functional requirements in a traditional provenance-unaware system and were implemented inside functional modules. For example, in workflow systems, one common constraint is that an activity A can start only after another activity B is finished. Such constraints are typically implemented as hard-coded logic in a provenance-unaware system. However, as shown in the rest of this paper, these constraints can be implemented as PBAC policies. That means, from the beginning of PAS development, developers need to decide what parts of general user requirements can be and should be captured as access control requirements that will be implemented as provenance-aware ACPs.

Inspired by the idea of security engineering[1] [6, 21], this paper explores the issues of engineering provenance-aware ACPs, specifically the issues of identifying access control requirements out of user requirements that can be implemented as provenance-aware ACPs, modeling provenance in abstraction for efficient specification of ACPs, and engineering provenance-aware ACPs in a systematic manner. Specifically, this paper introduces a typed provenance model (TPM) to abstract complex provenance graph at design level. A TPM captures semantically meaningful provenance abstractions which can be used to efficiently define and manage provenance-aware ACPs. These ACPs can then be evaluated according to the provenance graph. Furthermore, this paper presents a TPM-centric policy development process for identification, specification, and refinement of provenance-aware ACPs from the beginning of PAS development. We illustrate the concept of TPM and the process by a homework grading system.

We organize the rest of this paper as follows. Section 2 introduces preliminaries used in the rest of this paper. Section 3 presents the typed provenance model. Section 4 introduces a TPM centric process for engineering ACPs and applies it in a homework grading system. Section 5 discusses related work and section 6 concludes the paper.

2. PROVENANCE-AWARE SYSTEMS

This section introduces the Open Provenance Model, the general form of provenance-aware ACPs, and a homework grading system used as an example in the rest of the paper.

2.1 Provenance Data Model

A provenance data model defines the scheme of the provenance to be captured. Recently, a community-developed provenance data model, called Open Provenance Model (OPM), has been crafted, refined, and formally defined following a series of challenge workshops. OPM has gained attention as it provides a foundation for the causality dependencies of provenance and enables inter-operability of provenance across systems [12]. As shown in Figure 1, OPM organizes provenance as a directed acyclic graph, which includes three types of entities, three types of direct dependencies (shown in solid line) and two types of indirect dependencies (shown in dashed line). A provenance graph conforming to OPM is referred as an OPM graph in the rest of this paper.

Entities include artifacts, processes, and agents. In PAS, artifacts are data objects; processes are modules which take some artifacts as inputs, generate some artifacts as outputs; and agents are users who control the processes. Direct dependencies are dependencies directly related to a process and consist of *used, wasGeneratedBy*, and *wasControlledBy*, which are abbreviated as *u, g,* and *c* respectively. Figure 1 shows that a process $p2$ used an artifact $a2$ which was generated by a process $p1$. Process $p1$ used another artifact $a1$ and was controlled by an agent Ag.

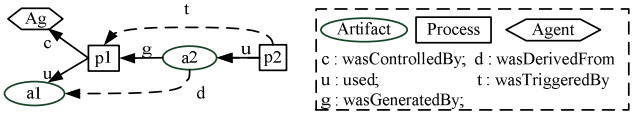

Figure 1: OPM Causality Dependencies.

OPM also defines indirect dependencies, such as *wasDerivedFrom* and *wasTriggeredBy*. The edge $d(a2, a1)$ in Figure 1 denotes that $a2$ was derived from $a1$ and $t(p2, p1)$ denotes that $p2$ was triggered by $p1$. The semantic of an indirect dependency could to some extent be characterized by a path in the OPM graph. For example, $d(a2, a1)$ could be mapped to the path $g(a2, p1) \cdot u(p1, a1)$, where '$\cdot$' operator concatenates two adjacent edges. This paper assumes only direct dependencies will be captured in an OPM graph. Indirect dependencies can be derived from the direct ones.

2.2 Provenance-aware Access Control Policies

Access control is the process of meditating each access request and evaluating it against some access control policies according to given facts to grant or deny the access request [17, 18]. An access control policy is generally a set of predicates and can be defined as a first-order formula [8]:

$$\forall x_1 \cdots x_m (f \Rightarrow \mathbf{P}(s, a, o)), \quad (1)$$

where f is a first-order formula, x_i (i=1,...,m) is a variable used in formula f, the predicate $\mathbf{P}(s, a, o)$ being true means the subject s is allowed to perform action a on an object o and the vector (s, a, o) forms an access request, '$\Rightarrow$' denotes that f is sufficient in granting an access request.

A provenance-aware access control policy is a policy defined with regard to provenance data. Policy architects need to define ACPs by utilizing some meaningful control units of provenance, which are often not the individual nodes or edges of a provenance graph but the entire subgraphs of a provenance graph. It is necessary to apply some grouping strategies on a provenance graph to identify these control units. One strategy is to define regular expression of path patterns on a provenance graph [4]. Park et al further named the regular expressions by literally meaningful dependency names to improve the efficiency of specifying PBAC Policies [13, 15]. Each dependency name can be seen as a short name of a provenance question against a particular starting node v to compute all nodes which caused v in a provenance graph. For example, *OwnedBy* is a short name of the provenance question "who are the owners of a document?".

[1]Security engineering believes that access control models and policy languages regulate what kind of ACPs can be specified. However, it is sometimes necessary or at least helpful to identify, specify, and verify ACPs from the beginning of software development.

In PAC, the access request (s, a, o) in the policy equation 1 could be extended to include some provenance questions. An access request in PAS can be defined as a 3-tuple as follows $(s, a, [q](o))$. Here $[q](o)$ denotes provenance answers of a set of optional provenance questions against the object o. $[q]$ denotes that provenance questions are optional.

In PBAC, the formula f in the policy equation 1 could be defined using provenance, which is the answer of some special provenance questions for the purpose of access control. That is, f could include a sub-formula $f' = p_1\ (\wedge|\vee)\ p_2\ \cdots\ (\wedge|\vee)\ p_n$, where each access condition p_i is an expression of applying set operations, common logical operations, and basic mathematical operations on the answers of provenance questions against either s or o. Similar to the formula 1, a provenance-aware access control policy can be formally specified in a form:

$$\forall x_1 \cdots x_m (f' \Leftarrow \mathbf{P}(s, a, [q](o))), \tag{2}$$

where f' and $(s, a, [q](o))$ has semantics as explained above. Note that the left arrow '$\Leftarrow$' means that the provenance assertions are conditions necessary but not sufficient for granting an access request because PAC and PBAC usually do not work alone but together with other traditional access control models, such as role-based ones.

For example, consider a policy that says a user u can see who is the owner of a homework h if u is the grader of the homework. This policy can be formulated as follows

$$\forall h \forall u (u \in \mathit{GradedBy}(h) \Leftarrow \mathbf{P}(u, \mathit{query}, \mathit{OwnedBy}(h)),$$

where $\mathit{OwnedBy}$ and $\mathit{GradedBy}$ are two provenance questions which query who owned and graded the homework h.

2.3 A Homework Grading System

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

- R_1. A student can upload, replace and submit her own homework. A homework can be submitted only once but can be replaced many times before final submission.
- R_2. A student or a professor can review a submitted homework if she was neither the owner nor one of existing reviewers of the homework, and the homework has been reviewed less than three times and not been graded.
- R_3. A professor can grade a homework by appending some existing reviews of the homework if the homework has been reviewed at least two times.
- R_4. A student can see how many times her homework has been reviewed or graded.
- R_5. A reviewer can see whether her review on a homework was appended to a grade or not.
- R_6. A professor who grades a homework can see who are the author and reviewers of the homework, and whether a homework was reviewed with conflict of interests (i.e. a homework was reviewed by its owner).

3. PROVENANCE IN ABSTRACTION

A PAS needs to answer various provenance questions. Most of them are proposed by domain users without enough technical knowledge. For example, a professor may ask who owned a homework and whether and when a homework was submitted. Domain users usually do not understand the complex provenance graph, not to mention how to use it to answer the obscure provenance questions in natural language [4, 10]. Because a provenance graph can only be captured at run-time, what developers can perceive at design time are only elements of conceptual software models, such as use cases, scenarios, or classes. An intuitive idea is to allow developers to design application specific types of possible provenance dependencies among different conceptual entities such as classes, actors, and business operations.

This section introduces an abstract model of a provenance graph, the typed provenance model (TPM). TPM is about application-specific conceptual provenance types while OPM is about the application-independent scheme of representing provenance instances. TPM captures entity types, provenance dependency types among entity types, as well as relationships and constraints among dependency types.

3.1 Entity types and dependency types

Each entity type in a TPM is a class which can be instantiated into nodes in an OPM graph. In the provenance graph of the HGS, an artifact node could be instantiated from a class in design model, for example $\mathit{Homework}$. A process node could be instantiated from a business operation in requirements model or a method of a class in design model, for example *upload* and *submit*. An agent could be instantiated from an actor of the target system, for example organizational roles the *Student* and *Professor*.

Dependency type is the core concept of a TPM. It has roots in the notion of "dependency name" [13, 15] and is actually a classifier of similar semantics of multiple dependency paths in an OPM graph. We formally define a dependency type T as a composition of its literal name (N), an effect entity type E, and a cause entity type C as follows.

$$T := N(E, C). \tag{3}$$

Note that N is a unique name of T and literally shows semantics of T. We will use the symbolic identifier T or the literal name N interchangeably 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 the effect node type and cause node type. For example, a dependency type $\mathit{ReviewedBy}(\mathit{Homework}, \mathit{User})$ can be instantiated into $\mathit{ReviewedBy}(hw_1, u_1)$ to denote that a homework instance hw_1 was reviewed by a user u_1. If we view each dependency type as a mapping from the effect node to its cause nodes, $\mathit{ReviewedBy}(hw_1)$ returns the set of users who reviewed the homework hw_1, and we have $\mathit{ReviewedBy}(hw_1, u_1) \equiv u_1 \in \mathit{ReviewedBy}(hw_1)$.

TPM includes two kinds of dependency types, the primitive ones, which can be instantiated into individual edges of an OPM graph, and the complex ones, which cannot be instantiated into individual edges rather into subgraphs of an OPM graph. Each complex type needs to be mapped to a composition of primitive types to make itself interpretable according to an OPM graph.

3.2 Primitive dependency types

Each primitive dependency type abstracts the semantics of a set of edges in a provenance graph and could be from a process type to either an artifact type or an agent type, or from an artifact type to a process type. Primitive dependen-

cy types related to the same process type can be grouped together to form a process-centered directed graph of provenance in abstraction, called provenance type graph.

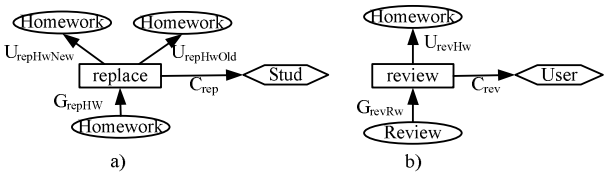

Figure 2: Primitive dependency types.

Figure 2-a is a provenance type graph centered at the *replace* process in the HGS. It shows that a process *replace* (*rep*) will take two homework as inputs. One is the old version of a homework *(hw)* to be replaced and the other is the new homework that is used to replace the old one. The process *replace* is controlled by students (*stud*) and generates a new version of homework as its output. Figure 2-a defines four primitive dependency types T_1 to T_4 as follows. Note that we assume that data is never overwritten or updated in place. Any modification to a data object will create a uniquely new instance of it in a provenance graph.

$$T_1 := U_{repHwOld}(rep, Hw), \quad T_2 := U_{repHwNew}(rep, Hw),$$
$$T_3 := G_{repHW}(Hw, rep), \qquad T_4 := C_{rep}(rep, Stud).$$

Here, $U_{repHwOld}$, $U_{repHwNew}$, G_{repHW}, and C_{rep} are unique type names, which literally expose the application specific semantics of dependency types. *U, G*, and *C* indicates that the type is an application specific subtype of *Used, wasGeneratedBy*, and *wasControlledBy* dependencies in OPM respectively. For example, $U_{repHwOld}$ means a process *replace* used an *old* version of *homework* as its input. Figure 2-b is a provenance type graph centered at the *review* process.

Each process type that can be finally instantiated into processes at run-time is a method in software design model or a business operation that can be refined into a set of methods. Each method signature could leads to a provenance type graph as shown in Figure 2.

3.3 Complex dependency types

TPM also includes the so-called complex dependency types to encapsulate the complex semantics which cannot be captured by individual primitive dependency types. Each complex dependency type can be defined as a composition of primitive dependency types.

First, the most basic complex dependency type is the concatenation of two primitive dependency types with the operator '$\cdot$'. To put formally, a complex dependency type $T := T_i \cdot T_j$ means that the effect node type and the cause node type of T are the effect node type of T_i and the cause node type of T_j respectively, and the cause node type of T_i is same as the effect node type of T_j. For example, T_7 below captures the semantics that a homework (Hw) was uploaded (up) by a student ($Stud$).

$$T_5 := G_{upHw}(Hw, up); \quad T_6 := C_{up}(up, Stud)$$
$$T_7 := UploadedBy(Hw, Stud) := T_5 \cdot T_6,$$

where G_{upHw} and C_{up} mean that an upload process generates homework and is controlled by students respectively.

Second, each edge of a provenance graph is directional. Its tail node is the cause that its head node became what it is. However, users could ask which are the effect nodes of a given cause node. We can formally 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. For example, T_{12} below computes the set of reviews related to a specific homework. T_8 - T_{10} are primitive dependency types in Figure 2-b.

$$T_8 := G_{revRw}(Rw, rev), T_9 := U_{revHw}(rev, Hw),$$
$$T_{10} := C_{rev}(rev, User), T_{11} := ReviewOn(Rw, Hw) := T_8 \cdot T_9,$$
$$T_{12} := {T_{11}}^{-1} := {ReviewOn}^{-1}(Hw, Rw) := {T_9}^{-1} \cdot {T_8}^{-1}.$$

Third, some semantics of provenance dependency can be denoted by paths with arbitrary lengths in OPM graph. We can apply regular expression operators over existing dependency types to concisely compose dependency types modeling these semantics. They are the operator "*" for 0 or more of the preceding element, the operator "+" for 1 or more of the preceding element, and the operator "?" for 0 or one of the preceding element. For example, an uploaded homework can be replaced many times before final submission by its author. T_{15} captures the provenance dependency between the submitted homework and its history versions.

$$T_{13} := G_{subHw}(Hw, sub), T_{14} := U_{subHw}(sub, Hw),$$
$$T_{15} := SubmisionOn(Hw, Hw) := T_{13} \cdot T_{14} \cdot (T_3 \cdot T_1) * .$$

Fourth, some semantics can be validated by several paths in a provenance graph either disjunctively or conjunctively. To this end, we introduce two operators to reason about the provenance. The conjunctive and disjunctive operator $\wedge$ and $\vee$ enable the refinement of a dependency type into multiple conjunctive or disjunctive sub-types. Note that a dependency type can be composed with another via the operator $\vee$ or $\wedge$ if and only if they have the same cause node type and effect node type. In the HGS, only the owner of a homework can upload, replace, and submit it and a homework reviewed by its owner is involved in conflict of interests. T_{18} and T_{20} below capture semantics between *Homework* and *Student*.

$$T_{16} := ReplacedBy(Hw, Stud) := ((T_3 \cdot T_1)*) \cdot T_3 \cdot \ T_4;$$
$$T_{17} := SubmittedBy(Hw, Stud) := T_{13} \cdot C_{sub}(sub, Stud);$$
$$T_{18} := OwnedBy(Hw, Stud)$$
$$:= T_{17} \vee (T_{15}? \cdot T_{16}) \vee (T_{15}? \cdot (T_3 \cdot T_1) * \cdot T_7);$$
$$T_{19} := ReviewedBy(Hw, User) := {T_9}^{-1} \cdot T_{10};$$
$$T_{20} := ReviewedCOI(Hw, User) := T_{18} \wedge T_{19},$$

where T_{18} is defined as three disjunctive sub-types. In T_{18}, T_{17} says that the user who submitted the homework is its owner; $(T_{15}? \cdot T_{16})$ says that the user who replaced it is its owner; $(T_{15}? \cdot (T_3 \cdot T_1) * \cdot T_7)$ says that the user who uploaded it is its owner. Notice that *Stud* is a subclass of $User$.

Note that developers have to ensure that complex dependency types are correctly composed to ensure the correctness of the ACPs specified with them. Various constraints may exist among dependency types. For example one dependency type can be combined with another via the operator $\vee$ or $\wedge$ provided they have the same cause node type and effect node type. Another typical constraint would be that one dependency type cannot be preceded by another one. In the HGS, suppose a dependency type T_r denotes a homework is the new replacement of another homework, and T_s denotes a homework is a submitted version of another homework.

$T_s \cdot T_r$ is semantically right while $T_r \cdot T_s$ is wrong because the *replace* process on a homework cannot be activated after the *submit* process on the same homework happened. Note that we do not explore the formalization and verification of various syntactical and semantical constraints in this paper but leave them as our future work.

The set of composition operators introduced in this section is complete in a sense that the concatenation operator '$\cdot$' and the inversion operator '$^{-1}$' provide sufficient expressiveness of our TPM in terms of abstracting any paths of a provenance graph which carry meaningful provenance dependencies among two nodes. However, new operators can still be added because developers can define their own reasoning rules on provenance types and formulate the rules as new operators of composing complex dependency types for convenience. By introducing complex dependency types and their mapping to compositions of primitive dependency types, a TPM enables the developers to efficiently define ACPs for PAS even when the provenance graph is not available, and ensures that the ACPs defined using dependency types can be evaluated according to the provenance graph captured at run-time.

4. ENGINEERING ACPS BASED ON TPM

This section first describes a TPM-centric process for engineering provenance-aware ACPs, then applies it in the HGS.

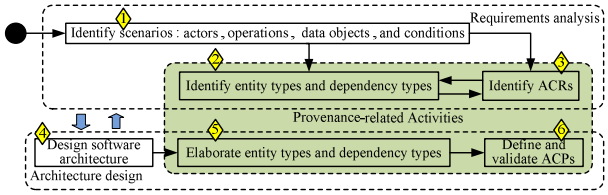

Figure 3: A TPM-centric process.

4.1 Process Overview

The process for engineering ACPs is embedded in the overall process of PAS development. As shown in Figure 3, rectangles in solid lines are activities and arrows are control flows between two activities. Two white rounded rectangles in dashed lines are containers for requirements analysis activities and architecture design activities. As the dark area shows, the activities for defining ACPs over typed provenance model spread across at least requirements analysis and architecture design phases. We introduce these activities and the related artifacts as follows.

1. *Identify scenarios: actors, operations, data objects, and conditions.* Identify scenarios of the target system by general requirements analysis methods. Each scenario is a short story of how a user interacts with the system to accomplish her task in a specific circumstance. In particular, some provenance questions may be identified as special scenarios [11], which are the sources of identifying data objects with provenance. Each scenario usually describes who can perform what operations on what objects under what conditions.

   A set of tuples (*actor, operation, object, conditions*) can be identified from each scenario. Each user appears only as a general actor, for example organizational roles *Student* and *Professor* in the HGS. Some conditions could be identified as access control requirements guarding business operations against data objects. For provenance questions, the operations are mainly various queries on provenance of data objects and conditions defined using provenance.
2. *Identify entity types and dependency types.* Analyzing each provenance question, identify a set of data objects whose provenance is needed to answer the provenance question, as well as actors and operations related to these data objects and define them as entity types in typed provenance model. Identify dependency types that are necessary for answering provenance questions. Choose appropriate and unique type names to carry the semantics of provenance dependencies.
3. *Identify access control requirements (ACRs).* After activities 1 and 2 are done, developers can start identifying ACRs related to provenance. On one hand, some provenance dependencies denoted by provenance types are sensitive and corresponding ACRs should be defined to protect them. On the other hand, some conditions guarding the operations against data objects identified in step 1 can be recognized as ACRs related to provenance, which could have not been identified as dependency types in step 2. So an iteration exists between activities 2 and 3.
4. *Design software architecture (SA).* Designing a software architecture entails allocating requirements into different components and defining connections among them. Each component provides a set of interfaces consumed by other components. Each interface includes a set of method signatures. In our opinion, the method is the minimal unit out of which primitive dependency types can be derived. For example, a *replace* method may lead to primitive dependency types of T_1 to T_4. A software architecture often includes multiple views. This paper only shows the role of the class diagram in deriving the primitive dependency types. Other views, such as state diagram, might also affects the way of modeling provenance types.
5. *Elaborate entity types and dependency types.* Refine both process types (the business operations) and artifact types (the classes at requirements level) into method signatures and classes in the software architecture. Derive necessary primitive dependency types out of the software architecture model. Map each complex dependency type to a composition of primitive dependency types, which is the key to refine ACRs identified in step 3 as ACPs that can be evaluated against the OPM graph.
6. *Define and validate ACPs.* Formally specify ACPs corresponding to the ACRs identified in step 3 according to the refined entity types and dependency types. Validate consistency among the ACPs and ACRs.

This process emphasizes the role of typed provenance model in specifying provenance-aware ACPs in the overall PAS development process. Note that the process itself is intuitive and does not promise to produce a good enough set of ACPs, which heavily depends on the experiences of policy architects. Several issues including the automatic derivation of primitive dependency types out of software architecture and the formalization and automatic verification of the correctness of a TPM model should be solved in the future.

Table 1: A list of scenarios of the HGS.

No.	actor	operation	data objects	conditions
1.1	Student	upload	a **homework**	C0: Any student in the HGS
1.2	Student	replace/submit	a **homework** to get a new or submitted **homework**	C1: Owner of homework C2: Homework was non-submitted.
2	Student or Professor	review	a **homework** to produce a **review**	C3: (Not C1) Not Owner of the homework C4: (Not C2) The homework was submitted C5: Not reviewed the homework before C6: The homework was not graded C7: Number of reviews of the homework < 3
3	Professor	grade	a **homework** to produce a **grade**	C8: Number of reviews of the homework ≥ 2
4	Student	query-prov-of	a **homework** on its reviewed times and graded state	C9: (C1) Owner of the homework
5	Student or Professor	query-prov-of	a **review** as part of a grade	C10: Owner of the review
6	Professor	query-prov-of	a **homework** on its author, reviewers, and whether it is reviewed by a user involved in conflict of interests.	C11: Any professor in the HGS

4.2 Process in Action

In this section, we further articulate the provenance-related activities shown in Figure 3 by applying them in the HGS.

4.2.1 Step 1: Identify and specify scenarios

Identifying scenarios is a common task in primary software development methodologies. We do not discuss in detail but just present the resulted scenarios in table 1, which are elicited from the requirements of the HGS in section 2.3. Scenarios 1.1-1.2 are from the requirement R_1. Scenarios 2-6 are from the requirements R_2-R_6 respectively. Note that scenarios 4-6 are provenance questions.

4.2.2 Step 2: Define entity and dependency types

Based on provenance questions 4-6 in Table 1, we can easily identify the data classes whose provenance should be captured. They are *Homework*, *Review* and *Grade*. We can further identify actors (UT) and operations (PT) related to these data classes (AT) as shown in Table 2. Most operations are literally comprehensible while *query-prov-of* is an operation to query provenance of either a homework, a review, or a grade.

Table 2: Entity types of the HGS.

ET	$:= UT \mid AT \mid PT.$
UT	$:= User \mid Stud \mid Prof.$
AT	$:= Homework \mid Review \mid Grade.$
PT	$:= upload \mid replace \mid submit \mid review \mid grade \mid query\text{-}prov\text{-}of.$

We identify dependency types as shown in Table 3 that are necessary to answer provenance questions (scenarios 4-6) in Table 1. Most of them have been introduced in section 3 except for the second and third types, which denote that a homework (Hw) or a review (Rw) is used by a *grade (grd)* process. Table 3 also shows the provenance questions that can be answered using a specific dependency type.

Table 3: Dependency types from prov. questions.

No	Dependency types	Question (type description)
1	$U_{revHw}(rev, Hw)$	4 (review used Hw)
2	$U_{grdHw}(grd, Hw)$	4 (grade used Hw)
3	$U_{grdRw}(grd, Rw)$	5 (grade used Rw)
4	$ReviewedBy(Hw, User)$	6 (Hw was ReviewedBy user)
5	$OwnedBy(Hw, Stud)$	6 (Hw was OwnedBy stud)
6	$ReviewCOI(Hw, User)$	6 (Hw was Reviewedby owner)

The dependency types listed in Table 3 include both primitive types (1-3) and complex types (4-6) but are not exhaustive. Dependencies types could be identified and further specified among any two entity types in ET. In addition, both the entity types and dependency types may need to be refined while designing a software architecture.

4.2.3 Step 3: Identify Provenance-aware ACRs

Based on proper access control models, developers can decide which conditions in Table 1 can be and should be ACRs. This paper focuses on identifying ACRs that can be implemented as provenance-aware ACPs under the guidance of both PAC model [1, 2, 9], where provenance is sensitive resources to be protected, and PBAC model [15], where provenance is used to adjudicate access requests.

In the HGS, we first identify PAC requirements from provenance questions (scenarios 4-6) in Table 1. Second, we identify access conditions that can be defined using provenance, i.e. identifying PBAC requirements. Note that all scenarios with conditions C1-C10 in Table 1 can be defined as PBAC requirements except for those only related to roles, such as C0 and C11. These PBAC requirements introduce new dependency types (such as 2, 5, 6 in Table 4) that were not identified in Table 3.

Table 4: Dependency types from ACRs.

No	Dependency types used to specify	conditions in Table 1
1	$OwnedBy(Hw, Stud)$	C1, C3, C9
2	$G_{subHw}(Hw, sub)$	C2, C4
3	$ReviewedBy(Hw, User)$	C5
4	$U_{grdHW}(grd, Hw)$	C6
5	$ReviewOn(Rw, Hw)$	C7, C8
6	$ReviewOf(Rw, User)$	C10

Finally, we need to identify new PAC requirements for the newly identified dependency types. For simplicity, we assume that provenance is not accessible to users if it is not explicitly permitted by provenance questions in table 1.

4.2.4 Step 4: Design software architecture

Based on the scenarios in Table 1, we define the software architecture of the HGS as a class diagram. Figure 4 defines a class hierarchy of various documents, including $Document$, $Homework$, $Review$, and $Grade$, and a class hierarchy of organizational roles which include *User*, *Stud*, and *Prof*. These classes except for the super class *Document* have been introduced as entity types in table 2. In Figure 4, each descendant class of *Document* overloads its methods to implement different business operations in Table 2. For example, the method $Hw.create(...)$ is a refinement of the business operation *upload* while $Rw.create(...)$ is a refinement of *review*. For simplicity, we omit the details of the

signature of each method. The class diagram is an important source for identifying primitive dependency types.

4.2.5 Step 5: Elaborate entity and dependency types

With software architecture models, we can elaborate some entity types, especially the process types and artifact types, define primitive dependency types according to method signatures implementing business operations in requirements model, and elaborate complex dependency types in Table 3 and 4 into compositions of primitive dependency types.

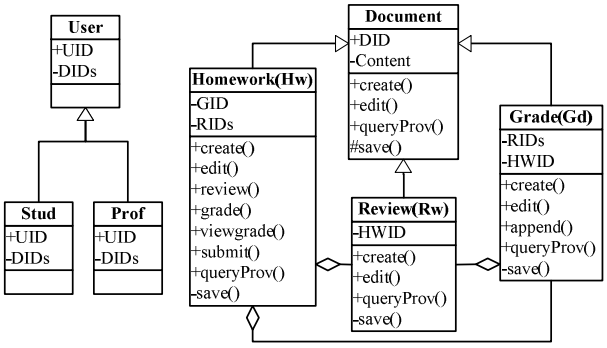

Figure 4: The class diagram of the HGS.

Some business operations identified as process types should be elaborated into method signatures according to software architecture. For example, $Hw.create(\cdots)$ is a process type elaborated from the process type *upload* given in table 2 while $Rw.create(\cdots)$ is elaborated from the process type *review* given in table 2. According to the elaborated process types, we can define primitive dependency types. For example, the method $Hw.create(\cdots)$ takes the homework content which is a string with arbitrary length as its input and produces a homework object as its output. So we can define corresponding primitive dependency types to capture provenance dependencies related to the process type $Hw.create(\cdots)$ as follows. T_{22} is the more accurate form of T_5 introduced in section 3.3, and T_{23} for T_6.

$T_{21} := U_{upHw}(Hw.create, String)$,
$T_{22} := G_{upHW}(Hw.create, Hw), T_{23} := C_{up}(Hw.create, Stud)$.

If one business operation is implemented as a chain of methods concatenated via method invocations in software architecture, the corresponding process type could be elaborated into multiple process types and the primitive dependency types related to the original process type should then become complex dependency types.

Complex dependency types in Table 3, 4 can be elaborated into compositions of available dependency types. For example, the dependency type *OwnedBy* captures the provenance dependency between a homework and its owner. However, designers have to clearly define how the owning relationship was established, such as from the successful execution of the methods *create*, *edit*, and *submit* of the *Homework* class. We have defined the *OwnedBy* dependency type as T_{18} in section 3.3, where T_1, T_3, T_4, T_6, and T_7 need to be redefined because the process types (the business operations, *upload* and *replace*) they relied on have been refined as $Hw.create$ and $Hw.edit$ in software architecture. In this way, we can map all complex dependency types in Table 3, 4 into compositions of available dependency types along with the refinement of software architecture.

4.2.6 Step 6: Define and validate ACPs

Using dependency types in Table 3 and 4 and their inversions, we can formally define ACPs in Table 5 to implement access conditions in Table 1. We formalize ACPs as a first-order formula given in equation 2. In Table 5, the letters 'u', 'h', 'g', and 'r' denotes an instance of *User*, *Homework*, *Grade*, and *Review* respectively. Each dependency type serves as a mapping from the effect node to its cause nodes. For example, $ReviewedBy(h)$ returns a set of users who have reviewed the homework h.

Note that ACPs in table 5 are defined on business operations so that we can easily validate them against access control requirements (access conditions) in table 1. However, each business operation may be implemented as a series of methods of architectural components in real settings. Developers need to further refine the ACPs given in Table 5 to get ACPs defined over architectural methods and validate them against access control requirements. Sun et al have discussed this issue in role-based access control systems built on component middleware [20, 22]. We will conduct the similar research on provenance-aware access control systems in the future.

5. RELATED WORK AND DISCUSSION

Provenance differs from traditional data and meta-data in that it is immutable and in a form of directed graph [2]. Not only individual nodes and edges but paths with arbitrary length among nodes in a provenance graph will be treated as a whole unit when it is protected or used to adjudicate access requests. Traditional security models and policy languages are not appropriate for PAS [1, 2]. Correspondingly, policy authoring tools that worked well for traditional access control policies would not work well in the PAS [16].

Policy languages for provenance-aware ACPs have employed regular expressions to dynamically identify protected resources as a control unit [4, 14]. Park et al further introduced the dependency names that envelops a path pattern specified in regular expressions to improve the efficient specification of PBAC policies [13, 15]. The dependency names inspired us to introduce the typed provenance model, which explicitly differentiate the provenance types and their instances, and serves as the basis of efficiently specifying provenance-aware ACPs. As far as we know, no research has previously modeled provenance as we do in this paper.

Provenance-aware security models like PAC [1, 2, 9, 15] and PBAC [13, 15] often do not work alone in real systems. They need to integrate with other access control models, such as RBAC [17, 18]. This paper emphasized on specification of ACPs over provenance in a similar way that is found in role engineering [19] on RBAC policies over roles. Issues of co-design of provenance-aware and other provenance-unaware ACPs are beyond the scope of this paper though we believe our approach is somewhat analogous with role-engineering and it is possible to integrate them together.

This work is the first step of our efforts in realizing idea of security engineering [5, 6, 19, 21] in provenance-aware access control systems. As far as we know, No similar work has been done on engineering provenance-aware ACPs with considerations of the special characteristics of provenance in contrast to general data and meta data. Similar to a

Table 5: Access Control Policies of HGS.

No.	Policies
1.2	$\forall u \forall h\ (u \in OwnedBy(h) \wedge\ Submittedby(h) = \phi\ ) \Leftarrow \mathbf{P}(u, replace/submit, h)$
2	$\forall u \forall h \left(\begin{array}{l} u \in OwnedBy(h)\ \wedge\ G_{subHW}(h) \neq \phi \wedge u \notin ReviewedBy(h) \wedge \\ {U_{grdHw}}^{-1}(h) = \phi \wedge |ReviewOn^{-1}(h)| < 3 \end{array} \right) \Leftarrow \mathbf{P}(u, review, h)$
3	$\forall u \forall h\ (|ReviewOn^{-1}(h)| \geq 2) \Leftarrow \mathbf{P}(u, grade, h)$
4	$\forall u \forall h\ (u \in OwnedBy(h)) \Leftarrow \mathbf{P}\left(u, \textit{query-prov-of}, \{|{U_{revHw}}^{-1}(h)|, {U_{grdHw}}^{-1}(h)\}\right)$
5	$\forall u \forall r\ (u \in ReviewOf(r)) \Leftarrow \mathbf{P}(u, \textit{query-prov-of}, {U_{grdRw}}^{-1}(r))$
6	$\forall u \forall h\ (u\ \in\ Prof \Leftarrow \mathbf{P}(u, \textit{query-prov-of}, \{OwnedBy(h), ReviewedBy(h), ReviewedCOI(h)\}))$

general security engineering solution, our method should be integrated with the overall software development process. So fat there exists only one development methodology specific for PAS, called PrIMe [11]. PrIMe instructs developers to identify provenance questions and then to adapt the target system to collect provenance for answering provenance questions [11]. However, PrIMe did not identify and model provenance at an abstract enough level for efficient specification and management of ACPs.

6. CONCLUSION AND FUTURE WORK

In provenance-aware systems, developers need to identify some original functional requirements as access control requirements that can be implemented as provenance-aware ACPs. They also need some kinds of provenance in abstractions to efficiently define the provenance-aware ACPs before a real provenance graph is captured at run-time. To solve these issues of developing provenance-aware ACPs, this paper argues that it is necessary to engineer provenance-aware ACPs from the beginning of PAS development. We introduce a typed provenance model (TPM) to abstract complex provenance graph. TPM forms a solid basis for efficient definition of provenance-aware ACPs. Furthermore, we present a TPM-centric process embedded in the overall software development process to guide the identification, specification, and refinement of provenance-aware ACPs from the very beginning of PAS development. In the future, we will apply our approach in additional systems to empirically evaluate its practicality of working together with the overall PAS development methodology, such as PrIMe [11], and other security engineering methodology, such as role-engineering [19]. We will also try to define a formal modeling language for TPM to automatically validate its consistency.

7. REFERENCES

[1] U. Braun and A. Shinnar. A security model for provenance. Technical Report TR-04-06, Harvard University Computer Science, Jan 2006.

[2] U. Braun, A. Shinnar, and M. Seltzer. Secure provenance. In *The 3rd USENIX Workshop on Hot Topics in Sec.*, pages 1–5, Berkeley, CA, USA, 2008.

[3] P. Buneman, S. Khanna, and W. C. Tan. Data provenance: Some basic issues. FST TCS 2000, pages 87–93, 2000.

[4] T. Cadenhead, V. Khadilkar, and et al. A language for provenance access control. CODASPY '11, pages 133–144, 2011.

[5] R. Crook, D. Ince, and B. Nuseibeh. On modelling access policies: relating roles to their organisational context. RE'05, pages 157–166, 2005.

[6] B. Fabian, S. Gürses, and et al. A comparison of security requirements engineering methods. *Requir. Eng.*, 15(1):7–40, Mar. 2010.

[7] P. Groth, S. Jiang, and et al. An architecture for provenance systems. Technical report, University of Southampton, February 2006.

[8] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4):21:1–21:41, July 2008.

[9] R. Hasan, R. Sion, and M. Winslett. Introducing secure provenance: problems and challenges. StorageSS '07, pages 13–18, 2007.

[10] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. Opql: A first OPM-level query language for scientific workflow provenance. SCC '11, pages 136–143, 2011.

[11] S. Miles, P. Groth, and et al. Prime: A methodology for developing provenance-aware applications. *ACM Trans. Softw. Eng. Methodol.*, 20(3):8:1–8:42, 2011.

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

[13] D. Nguyen, J. Park, and R. Sandhu. Dependency path patterns as the foundation of access control in provenance-aware systems. Tapp '2012, 2012.

[14] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han. An access control language for a general provenance model. SDM '09, pages 68–88, 2009.

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

[16] R. W. Reeder, L. Bauer, and et al. Expandable grids for visualizing and authoring computer security policies. CHI '08, pages 1473–1482, 2008.

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

[18] R. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40 –48, sept. 1994.

[19] M. Strembeck. Scenario-driven role engineering. *IEEE Security and Privacy*, 8:28–35, 2010.

[20] L. Sun and G. Huang. Towards accuracy of role-based access control configurations in component-based systems. *J. Syst. Archit.*, 57(3):314–326, Mar. 2011.

[21] L. Sun, G. Huang, and et al. An approach for generation of J2EE access control configurations from requirements specification. QSIC '08, pages 87–96. IEEE Computer Society, 2008.

[22] L. Sun, G. Huang, and H. Mei. Validating access control configurations in J2EE applications. CBSE '08, pages 64–79, 2008.