Towards Provenance-Based Access Control with Feasible Overhead

Lianshan Sun
College of Electrical and Information Engineering,
Shaanxi University of Science & Technology,
Xi'an, Shaanxi, China. 710021
sunlianshan@gmail.com

Jaehong Park and Ravi Sandhu
Institute for Cyber Security,
University of Texas at San Antonio,
San Antonio, TX, USA. 78249
{jae.park, ravi.sandhu}@utsa.edu

Abstract

Provenance is a directed graph that explains how a data item became what it is. It is recently proposed to use provenance to enable the so-called provenance-based access control (PBAC) in provenance-aware systems. Evaluating a PBAC policy usually involves one or more queries against provenance store. However, directly reusing existing provenance query engines in a PBAC enforcement framework may introduce unacceptable performance overhead because provenance store might grow to immense size. This paper argues that feasible performance overhead for evaluating a PBAC policy must be under a nearly fixed threshold that is tolerable for users no matter how big the provenance store is. This paper designs several tactics, in particular a PBAC-specific tactic–adding *shortcuts* in a provenance graph, to partially satisfy this requirement. Finally, we analyze several open questions with respect to adopting these tactics.

1 Introduction

Provenance captures the origins and processes by which a data item became what it is. Provenance is usually used to verify the trustworthiness and integrity of data items [7] or to enable better interpretation, examination, and reproduction of steps and results of scientific experiments in provenance-aware systems. It is recently proposed to use provenance in access control of provenance-aware systems to do so-called provenance based access control (PBAC) [12]. For example, in a homework grading system, a student cannot re-submit his/her homework if the homework has already been graded.

Provenance differs from traditional data items and metadata in that it is an immutable directed graph [2], which can incrementally grow to immense size at run-time [4]. A PBAC policy usually includes one or more assertions on sub-graphs with meaningful provenance semantics as a whole to adjudicate access requests [12]. A PBAC enforcement framework needs to execute one or more queries on the underlying provenance store to get these subgraphs of a provenance graph necessary for evaluating a PBAC policy. An intuitive solution for this requirement is to reuse existing provenance query engines in a PBAC enforcement framework. However this solution may introduce high performance overhead when the provenance store grows to immense size.

Nowadays a large amount of data items are being generated, transmitted, and used by inter-connected software systems. Considerably larger quantity of provenance of these data items could be captured, stored, shared, and queried [4]. The performance overhead of querying a provenance store via existing provenance query languages and their query engine prototypes, is known to increase very fast along with the increasing size of the provenance store [8, 1]. Evaluating a PBAC policy, that usually involves one or more queries against provenance stores which can grow to immense size [4], could introduce unacceptable performance overhead if existing provenance query engines are directly reused in a PBAC framework without any customization or auxiliary support.

With these insights, this paper argues that the feasible performance overhead of evaluating a PBAC policy must be under a nearly fixed threshold that is tolerable for users no matter how big the provenance store is. This paper further presents several possible tactics that can be taken in building a PBAC enforcement framework to partially satisfy these requirements, and discusses the possible challenges of the adopting these tactics in real settings. This paper is a first step towards a PBAC framework with feasible performance overhead.

2 Provenance-Based Access Control

Provenance captures various entities that are involved in producing a data item, such as artifacts (data objects), action processes, and agents (people), as well as the casuality dependencies among these entities. As shown in Figure 1, provenance is usually organized as a directed graph with n-

978-1-4799-3197-2/14/$31.00 ©2014 IEEE

odes for entities and edges for dependencies. Note that we assume that a data object 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. Each directional edge denotes that the tail node is partly caused by the head node or, to put in another way, the head is part of provenance of the tail.

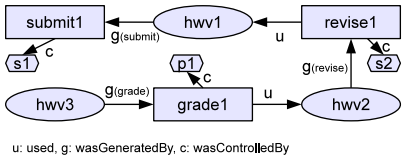

Figure 1. An example of a provenance graph [11].

For example, Figure 1 records the provenance of a homework in an online course management system. A student $s1$ submitted a homework $hwv1$. Later a team member $s2$ revised the submitted homework to get $hwv2$. Finally a professor $p1$ graded $hwv2$ and produced a graded homework $hwv3$. Note that a path with multiple edges may also carry some provenance semantics among entities [12]. In Figure 1, the path from $hwv3$ to $s1$ indicates that $hwv3$ was originally authored by student $s1$ and we name this path as "$AuthoredBy(hwv3, s1)$" for later use.

Traditional access control protection on data objects are usually provided through predefined constructs, such as roles in Role-Based Access Control (RBAC). PBAC provides access control protection on data objects by using the casuality dependencies among the requested data objects and its predecessors [12].

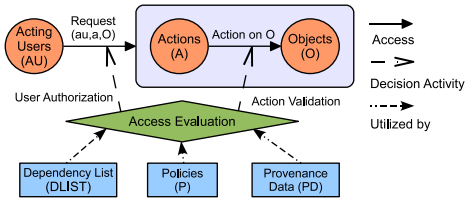

Figure 2. The basic PBAC model (simplified from [12]).

Figure 2 shows the basic model of PBAC, which is briefly illustrated as follows.

Acting users (AU) represent human beings who initiate requests for actions with respect to objects.

Actions (A) are actually action types that can be initiated by users as action instances against objects.

Objects (O) are resources that are accessed by users.

A **request (au,a,o)** consists of an acting user, an action instance, and a set of objects to be accessed.

Provenance data (PD) stores information about the performed actions and includes provenance graphs which start from current entities in the system.

Dependency List (DL) includes pairs of abstracted dependency names (DN) and corresponding dependency path patterns (DPATH). A dependency name is the short name for a DPATH that defines the pattern of similar provenance dependencies among entities by regular expression using the edge labels (types) and wildcards. For example, if the homework can be revised multiple times by different team members, the path $AuthoredBy(hwv3, s1)$ may have various forms in different situations. By using wildcards, the dependency path pattern named $AuthoredBy$ can be defined as follows:

$$AuthoredBy := (g_{(grade)} \cdot u)? \cdot (g_{(revise)} \cdot u) * \cdot g_{(submit)} \cdot c,$$

where "?" means 0 or 1, "*" means 0 or more and "$\cdot$" concatenates two adjacent edges. A dependency name can be seen as a pre-defined question that can be applied to multiple data objects of same type to compute their predecessors. In Figure 1, both $s1 \in AuthoredBy(hwv3)$ and $s1 \in AuthoredBy(hwv2)$ are true.

Policies (P) include a set of rules that need to be evaluated for either user authorization or action validation. Each rule may include one or more dependency names as a short reference to pre-defined provenance questions about a starting node v, which is either the subject or one of the objects in a request. For example, a PBAC policy that "a user can publish the graded homework if she/he is the author of the homework" is expressed as follows

$$allow(u, publish, o) \Rightarrow u \in AuthoredBy(o) \wedge |GradedBy(o)| > 0,$$

where $GradedBy$ is a dependency name for the provenance question "who graded a homework".

Access Evaluation function evaluates a request against policies and returns a boolean value. Note that PBAC framework needs to evaluate the policies by querying the provenance data one or more times. The existing provenance query engines utilizing regular path expressions on directed acyclic graphs, such as SPARQL [13], that can be reused in building PBAC framework, can introduce high performance overhead when provenance stores are very big.

3 PBAC-Specific Performance Overhead

Modern access control systems usually consist of a set of policies, a set of policy enforcement points (PEPs), and one or policy decision points (PDPs) [10]. A PDP is responsible for evaluating requests from PEPs with respect to rules in policies [10]. Both PEPs and PDP will unavoidably introduce some performance overhead relative to systems without access control [3]. Researchers have identified several sources and corresponding solutions of performance overhead of traditional access control [6, 5, 9].

However, traditional access control, such as RBAC, only needs to query the finite set of roles when evaluating a role-based policy. In contrast, PBAC needs to query the provenance store, which will continuously grow along as the system runs and could become extremely big. For example, Chapman et al show that an online protein interaction database named MiMI is 270 MB, but its provenance store can be accumulated up to 6GB [4]. Furthermore, it has been widely recognized that the overhead of querying a provenance store will grow as the provenance store gets bigger [8, 1]. From these observations, we can extrapolate that the provenance store used in PBAC to evaluate policies might grow to immense size and the performance overhead of PBAC will become intolerable. Note that this performance issue is specific to PBAC because traditional access control relied on facts, such as roles, that are finite sets and the performance overhead of querying these finite sets is usually low and even negligible.

This paper argues that the feasible performance overhead of evaluating a PBAC policy must be under a nearly fixed threshold (or at least be a linear function of the number of queries in the policy with multiple queries on provenance stores) that is tolerable for users no matter how big the provenance store is. This requirement is illustrated more clearly in Figure 3, which visually illustrates the desired form of performance overhead for a feasible PBAC framework.

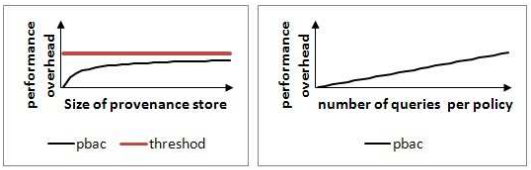

Figure 3. Desired form of performance overhead for PBAC.

Figure 3-a) indicates that no matter how big the provenance store is, the performance overhead of PBAC should not exceed a fixed threshold, which denoted by the horizontal bold line. Note that the threshold can vary from application to application, from scenario to scenario. That reflects the subjectiveness of the tolerance of users in different situations. Next section will present possible tactics for achieving the fixed threshold in a specific situation. Note that Figure 3-a) assumes that one policy does not include large number of queries against provenance store. That assumption usually holds true. However, in some extreme situations, there might be more than one and possibly many queries on provenance store issued during evaluation of a PBAC policy. Figure 3-b) indicates that the performance overhead of PBAC should be a linear function of the number of queries in a PBAC policy.

4 Tactics and Open Questions

The issue of minimizing performance overhead of access control itself is not new to PBAC. There are many issues that could cause performance overhead of access control, such as the inefficient storage of both policies and requests and the inefficient utilization of computing resources [6]. To address these issues, the tactics of policy numericalization, policy normalization [9], and policy refactoring [5], as well as implementing access control decision engine in technologies that can better utilize computing resources [6] have been introduced. Although these tactics are general enough to be employed in PBAC, they are not fully capable of solving the PBAC-specific performance issue identified in section 3. We introduce a PBAC-specific tactic for solving the identified performance issue as follows.

PBAC-specific performance overhead is introduced by querying provenance store that might grow to immense size when evaluating PBAC policy. In order to mitigate the performance overhead, an intuitive idea is to shorten the time used to query a provenance graph. Each query on a provenance graph is actually a process of recursively traversing the provenance graph from one starting node to its predecessors. The traversal is guided by the dependency path pattern corresponding to a dependency name, such as $AuthoredBy$. Obviously, the longer the traversed path is, the more steps of recursion exist and the higher the performance overhead will be. To this end, we believe that the performance overhead of querying a provenance store according to a dependency name can be mitigated if the path to be traversed can be shortened. To achieve this goal, we propose to add "shortcuts" of some long paths in the provenance graph. For example, Figure 4 includes an additional "shortcut" for the path "$AuthoredBy$". By this "shortcut", the time used to traverse the longest path $AuthoredBy(hwv3, s1)$ could be considerably shortened, from 6 steps of recursion to 1.

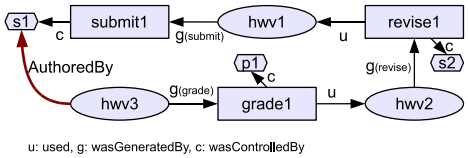

Figure 4. An example of shortcut in provenance graph.

We can theoretically prove that the "shortcut" tactic can work. Suppose the length (number of edges) of the path to be traversed is N and each node on the path may have M direct predecessors that can be categorized into A types in terms of provenance dependencies. Suppose the traversal algorithm starts from a starting node v and will be recursively executed N times guided by a given path pattern. Let

$T(N)$ denotes the total performance overhead of the traversal algorithm.

Each recursion queries the direct predecessors of the current node. That includes two steps. The first step is to locate all edges of a target type specified in the given dependency path pattern, such as g_{grade} and u. Suppose that all edges starting from the current node are sorted in terms of their types so that we can use the binary search to locate the edges of a target type in $O(log_2(A))$ time. The second step is to further traverse the graph along K identified edges of the target type, where K is a number less than M. So we can assume that the further traversal on the graph will cost less than $M \times T(N-1)$ time. Let $T(0) = 0$ denote the end of the traversal. So the recursive equation of the performance complexity of a query is:

$$\begin{aligned} T(N) &= log_2(A) + M \times T(N-1) \\ &= log_2(A) + log_2(A) \times M + M^2 \times T(N-2) \\ &= ... \\ &= log_2(A) + log_2(A) \times M + log_2(A) \times M^2 + \cdots \\ &\quad + log_2(A) \times M^{N-1} + M^N \times T(0). \\ &= log_2(A) \times (1 + M + \cdots + M^{N-1}) \\ &= log_2(A) \times \frac{(M^N - 1)}{(M-1)}. \end{aligned}$$

The approximate performance overhead of a query $T(N) = O(log_2(A) \times M^{N-1})$ mainly depends on the length N of a path and M the width of the graph. By adding "shortcuts" of long paths into a provenance graph, N can be maintained as a relatively small number even in a huge provenance store. In this way, it is possible to achieve performance goals shown in Figure 3.

Note that several questions need to be further clarified for successfully adopting the "shortcut" tactic in building a feasible PBAC framework. For example, when should these shortcuts be added, at the time of capturing provenance, at leisure time of the provenance store, or at the time appointed by provenance store manager? Where should these shortcuts be stored, in a separate store, in memory as cache, or in the original provenance store? What is the performance overhead of adding these shortcuts? Further research needs to be done to answer these questions to build a feasible PBAC framework.

5 Conclusion

This paper identifies a PBAC specific performance issue caused by querying provenance from the huge provenance store. This paper explicitly argues that the performance overhead of a feasible PBAC framework must be under a nearly fixed threshold, and further presents tactics for partially satisfying this requirement, especially the PBAC-specific tactic of adding the *shortcut* of provenance dependencies represented by a long path in a provenance graph. We further analyze the theoretical possibility and open questions of meeting the performance requirement by adopting the PBAC-specific tactic in real settings. This paper is our first step towards a feasible provenance-based access control framework.

6 Acknowledgments

This work is partially supported by National Science Foundation (No. CNS-1111925), National Science Foundation of China (No. 61202019).

References

[1] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *Proc. of the 13th Int. Conf. on Extending Database Technology*, EDBT '10, pages 287–298, New York, NY, USA, 2010. ACM.

[2] Uri Braun, Avraham Shinnar, and Margo Seltzer. Secure provenance. In *The 3rd USENIX Workshop on Hot Topics in Security*, pages 1–5, Berkeley, CA, USA, July 2008. USENIX Association.

[3] B. Butler, B. Jennings, and D. Botvich. An experimental testbed to predict the performance of XACML policy decision points. In *2011 IFIP/IEEE Int. Symp. on Integrated Network Management*, IM'11, pages 353 –360, may 2011.

[4] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proc. of the 2008 ACM SIGMOD Int. Conf. on Management of data*, SIGMOD '08, pages 993–1006, New York, NY, USA, 2008. ACM.

[5] Donia El Kateb, Tejeddine Mouelhi, Yves Le Traon, JeeHyun Hwang, and Tao Xie. Refactoring access control policies for performance improvement. In *Proc. of the third joint WOSP/SIPEW Int. Conf. on Performance Engineering*, ICPE '12, pages 323–334, New York, NY, USA, 2012. ACM.

[6] L. Griffin, B. Butler, E. de Leastar, B. Jennings, and D. Botvich. On the performance of access control policy evaluation. In *2012 IEEE Int. Symp. on Policies for Distributed Systems and Networks*, POLICY'12, pages 25 –32, july 2012.

[7] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *Proc. of the 2007 ACM workshop on Storage security and survivability*, StorageSS '07, pages 13–18, New York, NY, USA, 2007. ACM.

[8] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, and Farshad Fotouhi. OPQL: A first OPM-level query language for scientific workflow provenance. In *Proc. of the 2011 IEEE Int. Conf. on Services Computing*, SCC '11, pages 136–143, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: a fast and scalable XACML policy evaluation engine. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 265–276, New York, NY, USA, 2008. ACM.

[10] Tim Moses. eXtensible Access Control Markup Language TC v2.0 (XACML), February 2005.

[11] Dang Nguyen, Jayhong Park, and Ravi Sandhu. Dependency path patterns as the foundation of access control in provenance-aware systems. In *4th USENIX Workshop on the Theory and Pratice of Provenance*, TAPP'12, 2012.

[12] Jayhong Park, Dang Nguyen, and Ravi Sandhu. A provenance-based access control model. In *Tenth Annual Int. Conf. on Privacy, Security and Trust*, PST'12, pages 137–144. IEEE, July 2012.

[13] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. *W3C Recommendation*, 4:1–106, 2008.