An Attribute-based Authorization Policy Framework with Dynamic Conflict Resolution

Apurva Mohan
School of Electrical and Computer Engineering
Georgia institute of Technology
Atlanta, GA, USA
apurva@gatech.edu

Douglas M. Blough
School of Electrical and Computer Engineering
Georgia institute of Technology
Atlanta, GA, USA
doug.blough@ece.gatech.edu

ABSTRACT

Policy-based authorization systems are becoming more common as information systems become larger and more complex. In these systems, to authorize a requester to access a particular resource, the authorization system must verify that the policy authorizes the access. The overall authorization policy may consist of a number of policy groups, where each group consists of policies defined by different entities. Each policy contains a number of authorization rules. The access request is evaluated against these policies, which may produce conflicting authorization decisions. To resolve these conflicts and to reach a unique decision for the access request at the rule and policy level, rule and policy combination algorithms are used. In the current systems, these rule and policy combination algorithms are defined on a static basis during policy composition, which is not desirable in dynamic systems with fast changing environments.

In this paper, we motivate the need for changing the rule and policy combination algorithms dynamically based on contextual information. We propose a framework that supports this functionality and also eliminates the need to recompose policies if the owner decides to change the combination algorithm. It provides a novel method to dynamically add and remove specialized policies, while retaining the clarity and modularity in the policies. The proposed framework also provides a mechanism to reduce the set of potential target matches, thereby increasing the efficiency of the evaluation mechanism. We developed a prototype system to demonstrate the usefulness of this framework by extending some basic capabilities of the XACML policy language. We implemented these enhancements by adding two specialized modules and several new combination algorithms to the Sun XACML engine.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and protection; D.4.6 [**Operating Systems**]: Security and protection

IDtrust '10 April 13-15, 2010, Gaithersburg, MD

General Terms

Security, Languages, Performance

Keywords

Attribute-based authorization, authorization policy, conflict resolution

1. INTRODUCTION

As information systems become more complex and distributed in nature, system administrators and users need authorization systems which can help them share their resources, data and applications with a large number of users without compromising security and privacy. Although traditional authorization systems address the basic problem of granting access to only authorized individuals, they do not provide a number of desired features of modern authorization systems. These include 1) easily changing authorization based on accessor roles, group memberships, institutional affiliations, location etc., 2) multiple authorities jointly making authorization decision, 3) dynamically changing authorization based on accessor attributes, and 4) GUI-based general purpose tools for description and management of authorization rules. Some traditional authorization systems provide some of these functions on an ad-hoc basis. Although policies have always been part of authorization systems, they were mostly buried in other functional code and hence were difficult to compose and analyze.

Modern policy-based authorization systems provide most of these features. They have a separate policy module that can be queried to make authorization decisions. This module makes decisions taking into consideration all applicable policies for a particular access request. These policies may be defined by multiple authorities. The policies may have different or even conflicting authorization decisions for the same access request. Policy languages use policy combination algorithms (PCA) to resolve such conflicts. These algorithms take the authorization decision from each policy as input and apply some standard logic to come up with a final decision[1].

These PCAs are currently chosen at the time of policy composition and hence they are static. In highly dynamic environments, this is not desirable and there may be a need to select these PCAs dynamically. In this case, it will be useful to have a mechanism to select a suitable PCA based

[1]For efficiency reasons, policy engines only evaluate policies until they reach a final decision based on the combination algorithm.

on the dynamic contextual information available to the system. More discussion on this issue along with a motivating scenario is presented in Section 3.

PCAs used in current systems are also very restricted. There are a number of conflict resolution logics in general purpose computing which are not expressible as PCAs in authorization languages. Examples of these logics include hierarchy-based resolution, priority-based resolution, taking a simple majority vote, and taking a weighted majority vote. There is a need to include algorithms such as these as PCAs in authorization languages to provide more functionality and flexibility in defining policies.

Having a context-aware authorization system also provides the capability to define different policies for different contexts. These contexts can be distinguished by contextual data or environmental attributes. In this case, the policies will be modular making them easy to comprehend and analyze. Without the ability to choose the applicable policies based on contextual information, the policy composer is forced to duplicate each access control rule with and without the contextual information in the same policy. Although the same access control decision can be achieved in both approaches, the latter makes it difficult to analyze the policies and the effect of making changes to them. Also if policies are chosen dynamically, only a small set of rules will be evaluated for their applicability for this request. This reduces the number of matches with potential policy targets thereby lowering computation time.

Another advantage of using context-aware authorization is that a specialized policy created for some specific purpose can be added and removed from consideration dynamically without changing the existing policies. This is especially useful for systems that have to adhere to certain temporary authorization requirements which require special authorization rules. This is also useful in cases where the specialized policy is composed by some entity other than the one who usually creates and maintains authorization policies.

The main contributions of this paper are: 1) proposing a framework where authorization for a particular access request is decided dynamically based on context information, 2) supporting dynamic conflict resolution where PCAs are chosen at run time based on context information, 3) providing the ability to dynamically include (remove) specialized, short-term or add-on policies to (from) the authorization policy set, 4) increasing the efficiency of policy target matching during authorization, 5) increasing the modularity and clarity of the policies, 6) building a prototype authorization system to demonstrate the concepts, and 7) evaluating efficiency of the policy evaluation for the proposed framework.

2. ATTRIBUTE-BASED AUTHORIZATION SYSTEMS

In this section, we first introduce the basic constructs of attribute-based policy languages. We then describe some basic concepts of attribute-based authorization systems, define attribute-based policies, and policy combination algorithms used in conflict resolution.

2.1 Brief Introduction to Policy Languages

In this sub-section, we introduce the basic elements of attribute-based authorization policy languages. Although here we use eXtensible Access Control Markup Language (XACML) as an example to introduce the primary elements, these elements are similar in other policy languages as well.

XACML is an OASIS standard that describes a policy language for representing authorization policies and an access control decision request/response language [2]. XACML is based on XML. It describes general access control requirements while allowing for extensions for defining new functions, data types and combination logics. The language has syntax for defining authorization policies and building a request/response to validate authorization requests against the policies. The response contains one of the four possible outcomes of policy evaluation - Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request can't be answered by this service).

XACML has a Policy Enforcement Point (PEP) that actually protects the resource and a Policy Decision Point (PDP) that evaluates the access request against the policies. The PEP receives the access request from the requesting user and forwards it to the PDP which makes the decision in consultation with the policies. If the access is allowed, the PEP release the resource to the requesting user. The main components of a XACML policy are described below:

Policy - An XACML policy contains a set of rules with the subject and environment attributes, resources and corresponding actions. If multiple rules are applicable to a particular request, then the rule combination algorithm (RCA) combines the rules and resoles any conflict in their decisions. XACML supports the following RCA's - Deny-overrides (Ordered and Unordered),Permit-overrides (Ordered and Unordered), and First-applicable.

Policy Set - A policy set is a container which contains other policies or policy set. One or more of these policies or policy sets may be applicable to a particular access request. If more than one are applicable, then the Policy Combination Algorithms (PCA) are used to combine the policies and resolve any conflicts in their decisions. XACML supports the following PCA's - Deny-overrides (Ordered and Unordered),Permit-overrides (Ordered and Unordered), First-applicable, and Only-one-applicable.

Target - A Target is basically a set of conditions for the Subject, Resource and Action that must be met for a Policy Set, Policy or Rule to apply to a given request.

Rule - The rule is the core representation of the access control logic with the subject, resource, action and environment fields. It is a boolean function, which evaluates to true if the subject, resource, action and environment fields in the request matches with the fields in the rule.

2.2 Authorization Policy

In an attribute-based system, objects are protected by administrator (or object owner) defined policies. These policies define a set of verifiable attributes (with pre-defined values) against each resource for a set of privileges. These attributes are either the characteristics of the user or the environment. These attributes must be presented to the authorization module and verified by it in order to authorize the accessing user to access the requested object with specific privileges. Since the attributes have to be verifiable, they have to be certified by some entity which is trusted by the authorization module.

An attribute-based authorization policy is formally defined below.

Definition 1 : Let $\mathcal{SA}$, $\mathcal{RA}$ and $\mathcal{EA}$ represent the Subject, Resource and Environmental attributes respectively, each of which is well defined set of finite cardinality, given as $\mathcal{SA} = \{sa_1, sa_2.....sa_l\}$, $\mathcal{RA} = \{ra_1, ra_2.....ra_m\}$ and $\mathcal{EA} = \{ea_1, ea_2.....ea_n\}$. These attributes can take values $val\_sa_i \subseteq dom(sa_i)(1 < i < l)$, $val\_ra_j \subseteq dom(ra_j)(1 < j < m)$ and $val\_ea_k \subseteq dom(ea_k)(1 < k < n)$.

Attributes can be of two types, one which can take distinct and unconnected values (for e.g. 'role'='doctor' or 'role'='nurse') and another type which can take a single or range of values (for e.g. 'time' is between t_1 and t_2 or 'age' ≤ 21). In the latter case, the values that an attribute can take are connected. Without loss of generality, we define the latter group as attributes which can take either a single value or a range of values. For example, for a range of sa_j, the domain and values are defined as follows:
Attribute Type 1 -
$dom(sa_j) = [sa_j\_val_1, sa_j\_val_2...sa_j\_val_n]$, $val\_sa_j \in dom(sa_j)$;
Attribute Type 2 -
$dom(sa_j) = [low, high]$, $val\_sa_j = [low', high'] \subseteq dom(sa_j)$;
where, $(low' \geq low)$ and $(high' \leq high)$. If val_sa_j takes a distinct value in $[low, high]$, then $low' = high'$.

Definition 2 : Let Action define a set of actions which a subject can execute on resources. $\mathcal{ACT} = \{act_1, act_2.....act_p\}$. For example, the set of actions on a file can be {read, write, delete, append, execute}. Let $\mathcal{D}$ be the set of decisions that can result as a response to a predicate evaluating to true. $\mathcal{D} = \{d_1, d_2.....d_q\}$.

Definition 3 : An access request ($\mathcal{AR}$) is a tuple of the form $< s, r, a >$, where $s \subseteq \{\mathcal{SA}, \mathcal{EA}\}$, $r \subseteq \{\mathcal{RA}\}$ and $a \subseteq \{\mathcal{ACT}\}$. It represents that s is requesting to access r with rights a. A Rule $\mathcal{R}$ has the same format but defines the set s required to access r with rights a.

Definition 4 : A policy is a list of rules given as $\mathcal{P} = (\oplus, < \mathcal{R}_1, \mathcal{R}_2.....\mathcal{R}_s >)$. $\oplus$ is a combination function, which combines the rules to produce a single decision for the policy.

Definition 5 : A Policy Set (PS) is a container which contains a list of policies. It may also contain other policy sets. It is given as $\mathcal{PS} = (\oslash, < \mathcal{PS}_1, \mathcal{PS}_2, ...\mathcal{PS}_i >)$. Each $\mathcal{PS}_t$ represents either a policy set or a single policy[2]. $\oslash$ is a combination function, which combines all the policy sets. This combination function is used to combine policies and policy sets and has no direct relation with the rule combination algorithm.

Conceptually, a policy is a deliberate plan to implement authorization to a particular resource or group of resources. A rule is a component of the policy that defines a specific authorization predicate. A policy set is a container that contains a number of logically connected policies. In a multi-authority setting where the authorization policies for a particular resource are defined by a number of entities, all policies for that particular resource will form a logical policy set. For example, at a university, the firewall policies to protect a lab computer may be a combination of the policy defined centrally by the office of information technology, a specific department policy, a lab firewall policy, and the administrator defined policy for that computer. A policy set encompasses all of these policies. The policies can be defined in a number of policy description languages. Each has its advantages and disadvantages. In describing the policies in this paper, we will use the syntax and structure of XACML [2], which is an OASIS standard. XACML is an attribute-based policy description language and is used for implementing our prototype system. Although we use XACML for discussion and implementation, the model we present in this paper is generic and can be implemented in other policy languages like P3P [4] or EPAL [1].

[2]In which case the set has a single policy and no PCA.

2.3 Combination Algorithms and Conflict Resolution

In a large system, there may be multiple authorities who specify the authorization policies. As such, there can be multiple groups of policies. When a request is evaluated in the system, the authorization module determines which policy sets apply to the particular request. Then it checks which policies among those groups and which rules among those policies are applicable to the request. There can be multiple policy sets and multiple policies in each set applicable to a single access request. Even within each policy there can be multiple rules which apply to the access request. These rules and policies can have a different or even conflicting decision for the request. As such, a mechanism is needed to resolve these conflicts. Policy languages have some rule combination algorithms (RCAs), which evaluate the applicable rules based on the logic of the algorithm and resolve any conflict in their decisions.

Definition 6 : In a single policy, $\mathcal{E}(\mathcal{AR}, \mathcal{R}_i) \rightarrow d_i$, where $\mathcal{E}$ represents the evaluation of the i^{th} rule and d_i is the corresponding decision. The set of all the decisions is given as $\mathcal{D}^{Rule} = (< d_1, d_2, ..., d_x >)$. Rule Combination Algorithm (RCA) is defined as $\{\mathcal{RCA}\ \phi\ \mathcal{D}^{Rule}\} \rightarrow d$, where $d\ \epsilon\ \mathcal{D}$. ϕ represents *'applied to'*.

For example, a policy may use 'deny-overrides' as its RCA. In this case, if the algorithm finds even a single rule that denies the access, its final decision is 'deny'; otherwise its decision is 'permit' even if a single rule permits. If none of the rules either 'permit' or 'deny' the access, then the result is 'Not Applicable'.

For combining the policies and policy groups, policy languages have policy combination algorithms (PCAs). These algorithms work on similar logic as the RCAs. Each policy give a single decision for the access request. The PCA combines these decisions into a single decision by using the PCA logic.

Definition 7 : In the final policy list, $\mathcal{E}(\mathcal{AR}, \mathcal{PS}_i) \rightarrow d_i$, where $\mathcal{E}$ represents the evaluation of the i^{th} policy set and d_i is the corresponding decision. The set of all the decisions is given as $\mathcal{D}^{PS} = \{d_1, d_2, ..., d_x\}$. Policy Combination Algorithm (PCA) is defined as $\{\mathcal{PCA}\ \phi\ \mathcal{D}^{PS}\} \rightarrow d$, where $d\ \epsilon\ \mathcal{D}$.

In the current systems, these RCAs and PCAs are static and are determined at the time of composing the policies.

3. DYNAMIC CONFLICT RESOLUTION

In the last section, we saw how RCAs and PCAs resolve the conflicts among rules and policies to give a unique decision for an access request. We also noted that, in existing systems, these RCAs and PCAs are chosen at the time of composing the policies and hence do not change. This static composition may not be suitable for highly dynamic environments where there is a need to adapt the policies dynamically. If such a mechanism is available, then it can also serve as an easy tool for the policy composer, if he wishes

to change the RCAs and PCAs without recomposing the authorization policies.

Some researchers have proposed static conflict detection and avoidance, arguing that detecting and resolving conflicts in systems with a large number of policies in real time can be a daunting task [26]. We argue that, even though it is a challenging problem, it is a superior approach. Organization policies, regulatory polices, and user policies change regularly. If we perform static conflict analysis, whenever one of the policy changes, new conflicts can arise requiring some party to change their policies. Also, some policies that conflicted before one of the policies changed and were never composed, may now become acceptable. There is no mechanism to reconsider these rejected policies. Also, the static model does not take into account adding and removing specialized and time limited policies to provide flexibility in policy composition and maintenance.

3.1 Motivating Scenario

Let us consider a motivating scenario from the health care domain. Alex is a patient who stores his personal health record (PHR) with his health maintenance organization(HMO) called Superior Health Care (SHC). At SHC the patients' PHRs are stored in a repository where the access to the repository is mediated through a proxy. The proxy stores all the authorization policies. The policies may have multiple groups with policies defined by patients like Alex himself, the hospital which created the record, SHC's organizational policies, federal regulatory policies, and so on. When someone tries to access an EMR for a particular patient, the system will consult the applicable policies to check whether this access is allowed. Assume that, in normal circumstances, the policy combination algorithm used is 'deny-overrides', which is a secure and stringent policy. Suppose that Alex wishes to use a more lenient policy in case of an emergency, where he will share his PHR with any accessor who is authorized by at least one of the applicable policies. In this case, he needs to dynamically change his PCA from 'deny-overrides' to 'permit-overrides' whenever there is an emergency and back to 'deny-overrides' once the emergency is over. The traditional method would require him to change his policies twice to achieve this. If Alex want to have several dynamic options, he will have to change his policy description each time such a dynamic change occurs.

In the proposed model, Alex can define all such dynamic conditions as an attribute-based policy and the evaluation of these policies will determine what PCA will be used for the current access request. The model extends this concept to the selection of the RCA dynamically. It is desirable that the user has the ability to define several dynamic conditions simultaneously, need not change his policy descriptions every time one such condition changes, and also need not keep track of the dynamic changes. This is one of the key advantages of using the proposed system. If Alex tries to achieve the same effect in current policy-based systems with static conflict analysis, when an emergency occurs he will have to recompose his policy with 'permit-overrides' and resolve all conflicts created in the process. When the emergency is over, he will have to recompose his policies with 'deny-overrides' and resolve all conflicts again. He cannot create a special policy for an emergency, because his two policies are inherently contradictory. This puts a heavy burden on the user and also, by definition an emergency comes unexpectedly, therefore Alex cannot be expected to recompose policies when an emergency has already occurred. In current systems, users like Alex do not change their policies on such events. Our novel framework enables users to achieve this with little effort and provides an important new functionality.

3.2 Proposed Model

In this section, we present a novel mechanism to dynamically determine the policies applicable to an access request and to evaluate only the applicable policies. In this model, we evaluate the authorization policies in two stages. In the first stage, we determine which policies are applicable to the current access request and we also dynamically determine which PCA will be used to resolve the conflicts in the authorization decisions. In the second stage, we evaluate only the applicable policies using the PCA selected in the first stage.

During stage one, the total applicable policy set (TAPS) is determined by selecting only those policies where at least one of the authorization rules is applicable to the current access request. If $PS_1, PS_2...PS_n$ are the authorization policy sets, then the TAPS for a particular $\mathcal{AR}$ is given as $TAPS = \oslash\{PS_1, PS_2....PS_n\}$.

The combination algorithm $\oslash$ used is *'all-that-apply'*, which is a new rule combination algorithm defined in Appendix A. The *'all-that-apply'* algorithm has been implemented in our modified XACML engine (see Section 5). To evaluate TAPS, all available policy sets are evaluated as explained in Definition 6. If a policy set has at least one rule that applies to the current access request, we include it in the TAPS. To find an applicable rule, we consider the subject and environment attributes in the access request (which is the set $\{EA \cup SA\}$) along with their boolean relationships. We then match that with the rules in the policy level target. We try to find a rule with the same set $\{EA \cup SA\}$ with the same relationships so that at least one of the attribute combinations matches with those in the $\mathcal{AR}$. EA, SA and RA are specified in Definition 1.

To aid in determining applicable policy sets, we create a meta-policy file called the M-Policy. This file contains one rule for each authorization policy set in the system. This rule is a copy of the policy level target rule included in each set. This rule is a method, in a language such as XACML, to define whether a particular policy is applicable to the given access request and it makes the processing faster. Including it in the M-Policy file has two advantages, namely the processing of the M-Policy file is much faster compared to evaluating the policy level target rule in each file. These rules are optional in XACML. If they are not present, policy evaluation will take longer. Also, we do not use any rule level targets in the XACML policies. As such, we compare the best case performance XACML can offer with our TAPS algorithm. The *'all-that-apply'* algorithm makes it possible to evaluate all target rules at the same place. Each rule in the M-Policy is evaluated (refer to Definition 6). If a rule evaluates to 'permit', it means that the target rule representing the respective policy is true and that policy is applicable. We then include that policy in the TAPS.

To apply the TAPS algorithm to current XACML based systems, we can create an M-Policy file if all the XACML policies in the target system have a policy level target and no overriding rule level target. In systems where either there

are no policy level targets or overriding rule level targets are present, an efficient way to implement the TAPS algorithm is to broadly categorize the available policies and use these categories to select the applicable policies. Although this selection will neither be fine-grained nor accurate, it will still improve the performance of the evaluation system because by using TAPS we can filter out non-applicable policies at an early stage. So, although the performance will not be optimal in this case, it will still be better than the current performance.

The next step in stage one is to determine the applicable PCA (PCA_{apply}) based on a set of environmental attributes, which define the specific conditions under which each of the PCAs is applicable. These environmental attributes essentially define the context of the $\mathcal{AR}$. Some of these attributes might accompany the $\mathcal{AR}$ while others can be provided by an internal or external system entity. We assume that the dynamic decision of which PCA to select is itself based on a policy. Thus, there is a policy set containing the rules governing PCA selection. The PCA rules are defined so that they are mutually exclusive and only one of them is applicable in a particular situation. Although this might seem complex, it is not really so because there are typically a small number of combination algorithms to choose from. This is enforced by using the combination algorithm $\oslash$ *'only-one-applicable'* to choose among the PCAs. *'only-one-applicable'* returns the applicable PCA if one and only one rule evaluates to 'permit'. If zero or more than one rule (and hence the PCA) evaluates to 'permit', then an error code is returned. All rules in the policy set are evaluated and the applicable PCA is selected to be used for resolving conflicts for this access request.

Now in stage two, the final authorization decision is calculated by evaluating the TAPS as $\mathcal{E}(\mathcal{TAPS}) = \{TAPS^{PCA_{apply}}, AR\}\ \phi\ \mathcal{D}^{PS} \rightarrow d$. As defined in Definition 7, in this evaluation, we consider all policies present in the TAPS and evaluate them against the access request $\mathcal{AR}$. The $\oslash$ used in this case is PCA_{apply}, which is calculated in the previous step.

As an example, using this model, Alex can create a PCA selection rule to the effect that if the $\mathcal{E}_{\mathcal{A}} = (\text{'}emergency' = \text{'}true')$, then the PCA 'permit-overrides' is used. The effect will be to allow access to anyone who can satisfy at least one of the applicable policies. On the other hand, in case where $\mathcal{E}_{\mathcal{A}} = (\text{'}emergency' = \text{'}false')$, PCA 'deny-overrides' can be used. This will limit access to holders of those attribute combinations that are not denied by any policy and are allowed access by at least one applicable policy. Since this evaluation is done during each access request, the PCA will change dynamically whenever there is an emergency.

In addition to providing this novel functionality, our framework proposes the use of TAPS to reduce the policy set to be evaluated for each access request. As shown in Section 6, this improves the real time system performance by 4-8 times. Formulation and evaluation of these rules is explained in more detail in Section 4.1.

4. SYSTEM DESIGN AND BACKGROUND MODULES

In this section, we will first present the system design for a generic implementation of this authorization framework, and then describe some background modules used for building the prototype.

4.1 System Design

The proposed system has a two stage authorization process, where in the first stage the applicable policy set and the applicable PCA is determined and in the second stage the applicable policies are evaluated to reach an authorization decision. For the first stage, the policy is created with an index rule for each policy in the TAPS. An index rule is of the form $< \{\mathcal{SA}, \mathcal{RA}, \mathcal{EA}\} : PolicyId >$, where PolicyId is the index id of a particular policy. For example, if policy 'P1234' is applicable to requests in an emergency scenario, then the index rule will be represented as -
$< \{EMT.EMTLicense = \text{'}valid'\} : P1234 >$
$< \{CompanyY.Dispatched = \text{'}true'\} : P1234 >$
$< \{EMT.Employer = \text{'}CompanyY'\} : P1234 >$

The attribute in the index rule is directly provided by an attribute provider (AP)[3]. In this example, the three attributes jointly establish that the EMT's license is valid, he works for company Y and company Y was dispatched to the emergency by the 911 operator. These attributes will be provided by distinct entities. Using them together can establish a complex fact, which cannot be verified by any single entity in the whole system. Note that if an index rule does not contain any attributes i.e. $< * : PolicyId >$, then it is true by default and that policy is always included.

For an access request, the attributes present in the request are compared against the index rules and, in many cases, only a small number of policies will be included in the TAPS. As a result, the policy evaluation stage will be much faster in these cases. The diagram in Figure 1 describes the dynamic authorization process. A similar policy is created with an index rule for each available PCA. Based on the attributes in the index rules, we determine which PCA will be applied to this particular request.

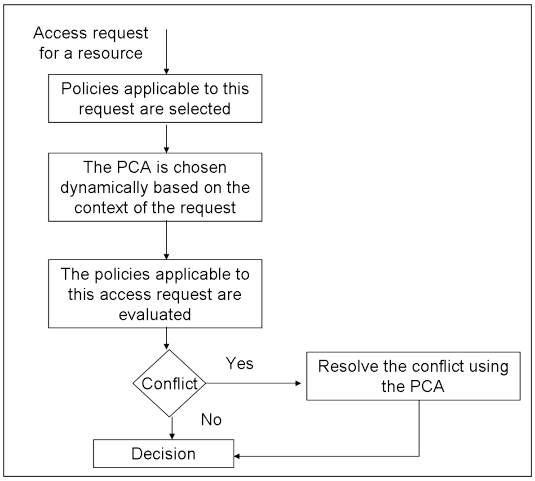

Figure 1: Block diagram of policy evaluation using the proposed framework.

[3]An AP is an entity similar to an identity provider. We define an AP as an entity that can certify certain attribute values for an individual due to its special relationship with the individual. For example, an employer can certify an employee's role in an organization.

4.2 Application Scenario

To understand the implication of using context information in the total applicable policy set (TAPS) evaluation and using dynamic PCA selection, let us again consider the previous health care domain scenario. Assume that Alex's HMO where he stores his PHRs has access policies for data based on criteria like data type, membership type, etc. Alex's policies also apply to his PHR, as described earlier. Now Alex, who lives in Atlanta is planning a trip to Florida for a week and he wants his PHR to be accessible to any physician or 'paramedic in Florida' during that week in case he needs medical help. Using our proposed model, he can add a special policy saying $< \{startdate \leq date \leq enddate\} : P2345 >$, where P2345 describes the special permission to 'physicians' in general and 'paramedics in Florida'. Upon evaluating this index rule, Alex's authorization system will compare the current date with the date range in the index rule and will include P2345 during that particular week. Since the proposed model is attribute based, Alex can take advantage of this by adding multiple attribute combinations. Assume that Alex's location can be tracked from his mobile phone, which communicates that to his authorization system over a secure channel. Then Alex can set the index rule as follows : $< \{startdate \leq date \leq enddate\}, \{location = Florida\} : P2345 >$.

This additional attribute will make sure that the lenient PCA is chosen only when he is physically in Florida[4]. Alex's mobile phone is used to provide his location, but the PHR will be primarily be accesses by the paramedics and physicians using their systems. In the event that he has to cancel his trip, his more lenient policy will not be in effect and his information will not be available to any paramedic in Florida. He also has the convenience of setting this rule once and then forgetting about it, irrespective of whether he actually makes the trip or not.

It is important here to note the difference between creating a new access rule in Alex's policy vs. creating an add-on access policy. While the former is possible using the current authorization systems, it will require Alex to modify his policy by adding new access rules and probably changing the rule combination algorithm. The effects of doing both these actions is hard for an average user to comprehend. If Alex has set his RCA as 'deny-overrides' and he wants to add his new rules to permit access during that particular week, he will need to either change the RCA to 'permit-overrides' or change each of the deny rules in the policy. Doing either is not desirable because his deny rules will be bypassed. In the proposed system, Alex can add a policy to the policy set defining his access policies and change the PCA to 'permit-overrides' for the specified period. Doing so will still keep all of Alex's deny rules unmodified and his policy set will allow access when at least one of his policies allow access, which is what he intended to do. This is hard to do in current systems, because PCA cannot be changed according to dynamic requirements. The resulting policy set is also more modular and analyzing such a policy set is easier. Finally, it saves the effort and complexity of analyzing the effects of changing the RCA or policy rules, not to mention restoring the original state once the specified time has passed. An example XACML policy for Alex is shown in Appendix B.

An additional benefit of our framework is that SHC can create index rules using attributes like 'username'[5], 'datatype', and 'data source' to create index rules to quickly select relevant policies when a physician tries to access Alex's PHR. These relevant policies form the TAPS for this access request. Suppose policy P880 contains Alex's disclosure policies, P130 contains data source's policy, P110 contains HIPPA policy, P112 contains the electronic privacy act[6], and P21 contains the SHC's disclosure policies. SHC's index rules for Alex's PHR are shown below :

$< \{`username = Alex'\} : P880 >$
$< \{`datasourceId = 814820'\} : P130 >$
$< \{`datatype = PHR'\} : P110, P112 >$
$< \{*\} : P21 >$

Note that, in the last index rule, the attribute value is left blank, which results in P21 being included every time. Using this efficient evaluation of TAPS, SHC can quickly determine the policies that need to be evaluated for an access request to Alex's PHR. We report some performance results of the efficiency of TAPS evaluation in Section 6.

5. PROTOTYPE IMPLEMENTATION

In this section, we describe the prototype implementation of the proposed framework. The prototype implementation of the framework extends the functionality of the policy language. The implementation is done using Sun's open-source XACML engine implementation, where we implemented additional modules and PCAs using Java. The generated policies are written in XACML. We use the Sun XACML PDP implementation because its loading and evaluation times are both reasonable when compared to other popular XACML implementations like XACMLLight and XACML Enterprise. Its overall performance is much better than XACMLLight and close to XACML Enterprise. A detailed comparison of the three implementations is done in [25].

The authorization policy consists of multiple policy sets. These sets consist of the system policy, the patient policy, and the data source policy. The system can be extended to consider the data accessor's policy to ensure that the obligations associated with the access request will be honored. The authorization module is set up as shown in Figure 2. The 'Policy Load and Evaluation' and 'Ancillary' modules are part of the standard XACML engine and the 'PSS' and 'PCA Selector' (explained later in this section) are added to the XACML engine. To make the proposed model closely compliant with the existing XACML engine, we have modeled the two new sub-modules as XACML policy sets, so that the XACML policy engine can be used to do these evaluations as well.

Policy Set Selector (PSS) - The PSS takes the authorization policy as the input, which contains all the available policy sets. The schema of the TAPS as a policy file is shown in Figure 3. It is organized in the Subject, Resource, Action and Environment structure. The PSS evaluates each policy set to find out all the sets that are applicable to this access request. The PCA used here is 'all-that-apply', which is es-

[4]We assume that Alex always carries his mobile phone with him because in essence the service is tracking a device and not Alex himself.

[5]The system can use any pseudonym to link Alex's PHR to his policies.

[6]The assumption here is that the rules in these acts can be encoded in a high level language like EPAL or XACML.

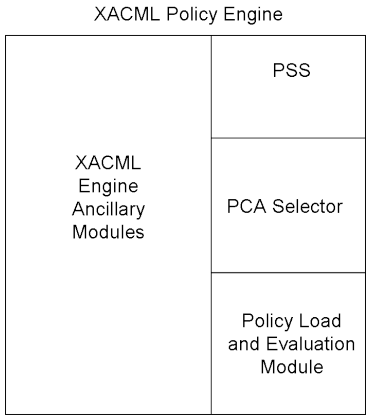

Figure 2: Modified XACML policy engine.

pecially developed for the PSS. The function of this PCA is to evaluate all the policy sets and output all that apply. All the policy sets selected by the PSS are stored in a data structure and only those policy sets are considered in the evaluation phase. As mentioned earlier, this reduces the number of policies to be evaluated for an access request and results in considerable run time performance improvement. A detailed discussion of the performance improvement is given in Section 6.

```
<PolicySet Combination Algo: all-that-apply>
<Policy Description : Policy 1>
    <Subjects>
        Required Attribute Sets
    </ Subjects >
    <Resources>
        Policy set or policy
    </Resources>
    <Actions>
        Include Policy set / policy
    </Actions>
</ Policy>
<Policy Description : Policy 2>
:::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::
</Policy>
<Policy Description : Policy 3>
:::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::
</Policy>
</ PolicySet>
```

Figure 3: Policy set selector module as a XACML policy set.

PCA Selector - The PCA selector reads the PCA selection file, which is described as a XACML policy. This description is created by the entity that is responsible for making sure that all the relevant policies are taken into consideration. This entity should make sure that the all the available PCAs are encoded as individual policies as shown in Figure 4. This system can be used as a static system by defining the selected PCA with no attributes (hence always applicable) and defining all the other PCAs with attributes that are never true. Although such a configuration may not provide some of the key benefits of the proposed framework, it may sometimes be required for backward compatibility.

The PCA selector file is a policy set as shown in Figure 4. All the PCAs are described as contained policy sets and the combination algorithm used is 'only-one-applicable', which is a standard XACML PCA. It returns 'permit' if one of the policy sets is applicable and 'deny' if zero or more than one policy set are applicable. In case the result is 'permit', the applicable policy set returns the name of the PCA to be used in combining policies. This module provides the novel functionality of selecting the PCA dynamically as described in Section 3.2.

```
<PolicySet Combination Algo: only-one-applicable>
<Policy Description : Policy 1>
    <Subjects>
        Required Attribute Sets
    </ Subjects >
    <Resources>
        PCA Algorithm 1
    </Resources>
    <Actions>
        Use this PCA algorithm
    </Actions>
</ Policy>
<Policy Description : Policy 2>
:::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::
</Policy>
<Policy Description : Policy 3>
:::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::
</Policy>
</ PolicySet>
```

Figure 4: PCA selector module as a XACML policy set.

To continue with the example in Section 3, the PCA selection policy set will be set as shown in Figure 4. Initially, when there is no emergency, the PCA 'deny-overrides' will be selected. This will be indicated by the attribute 'emergency' being set to false. When there is an emergency, the attribute is set to true and the PCA evaluation will give the output as 'permit-overrides'. The output PCA again becomes 'deny-overrides' once the emergency is over and the corresponding attribute is set to false.

This attribute can be provided by a number of entities like the 'emergency operations center', the '911 operations center', the patient himself or any other entity that the patient's agent trusts to provide this attribute. Although it sometimes might be difficult to ascertain that this particular patient is involved in an emergency, the patient would give more priority to making his PHR available to medical personnel in an emergency rather than to his privacy. Since the entire system can be audited, any breach of privacy can be discovered on audit.

6. PERFORMANCE EVALUATION

In this section, we will discuss the performance evaluation of the various components of the proposed framework. We are basically measuring the following parameters: 1) overhead in evaluating the total applicable policy set (TAPS), 2) overhead in dynamic selection of the PCA, and 3) time saved in evaluating just the TAPS (and evaluating applicable policies) compared to performing a target match on all the available policies (and evaluating applicable policies).

To measure these parameters, we evaluate the following - 1) TAPS evaluation time vs. total number of available policies , 2) PCA evaluation time vs. number of attributes in each index rule, 3) evaluation time vs. number of policies (with and without TAPS). Reasons for choosing these parameters and the evaluation results are discussed in detail in Section 6.2.

6.1 Evaluation Setup

In the evaluation setup, we create XACML policies for the modules described in Section 5. For evaluating the TAPS, we use the schema shown in Figure 3. We setup a XACML policy file with one index rule representing each available policy file (or policy set). Each index rule contains two attributes, both of which are required for access. There are 16 attributes in total and we select 2 out of them randomly. For the experiments, we use 1,2,4 and 8 index rules for each policy file in each run of the experiment. We also vary the total number of available policies from 1 to 10,000 increasing the number of policies by an order of magnitude each time. Most of the real world policies use 10-20 user attributes coming from the organizations LDAP server [22], [3], hence we feel 16 is a representative number. Moreover, this is a configuration parameter and not a limitation because it can be scaled easily. We also scale the number of attributes in one of the experiments (as described in this Section 6.2.2). We believe that most of the real world systems use much less than 10,000 policies. We evaluate performance up to 10,000 policies to observe the system performance over a broad range.

For selecting the PCA, we use the schema shown in Figure 4. Since we have a fixed number of PCA's in the system, we use this evaluation to scale up the number of attributes from 2 to 10,000 in each index rule. This evaluation gives us an estimate of the evaluation time in a system with large number of attributes.

For evaluating the actual policies, we have created policies with 1,2,4 and 8 rules per policy to be used in different runs of the experiment. We created sets of 10, 100, 1,000, and 10,000 policies.

All experiments were run on a single 2.4GHz Intel Dual Core Pentium machine with 2 GB of physical memory.

6.2 Evaluation Results

In this subsection, we present the performance results for the different cases just described.

6.2.1 Case 1

In this case, we evaluate the time consumed in evaluating the TAPS with varying number of total available policies. The RCA used is 'all-that-apply', so the evaluation considers all the policies that apply to a particular access request. We change the number of policies from 1 to 10,000 by increasing the number of policies by an order of magnitude in each step. We also vary the number of index rules applicable to each policy to 1,2,4, and 8 in different runs of the experiment. The result is shown in Figure 5. We observe that the evaluations take almost linear time as shown in this semi-log graph. The evaluation time is within 2 seconds even with 1,000 policies with 8 rules each, whereas with 100 policies with 8 rules each the evaluation time is within 250 milli-seconds.

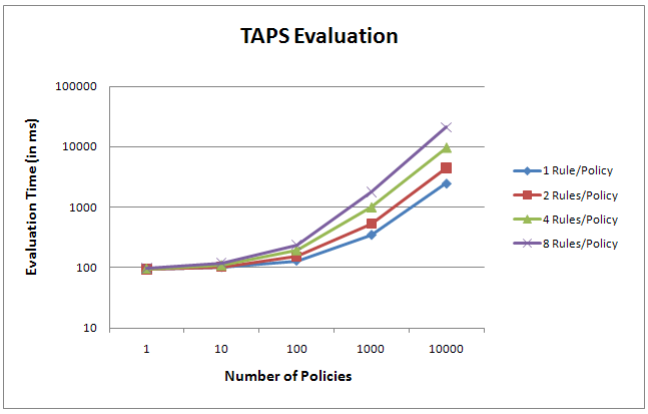

Figure 5: Evaluation time vs. number of available policies.

6.2.2 Case 2

In this case, we evaluate the applicable PCA from a list of PCAs supported by the system. In our prototype system, we have seven PCAs, each denoted as a policy set with its own index rule. We increase the number of attributes used in each index rule to understand the effect of scaling the attributes on performance. We increase the number of attributes from 2 to 10,000. The run time performance is shown in Figure 6. We observe that even with 100 attributes per index rule, the total evaluation time is under 280 milliseconds.

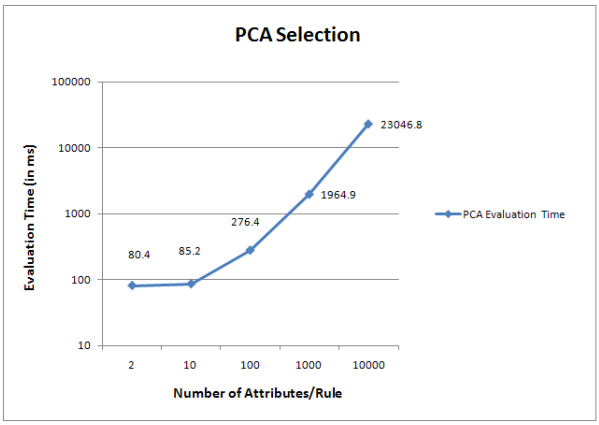

Figure 6: Evaluation vs. number of attributes per index rule.

6.2.3 Case 3

In this case, we evaluate the same set of policies with and without the PSS module and compare the performance of

the two systems. The setup is described in Section 6.1. In each policy file, we have a policy target set up, which is the default method XACML uses to check whether the current policy (file) is applicable to the current request. This target can be set up by resources, subjects, actions, or environments. We set up these targets with applicable subjects values. This allows us to make a direct comparison with our experimental setup. Also, this does not limit the use of target in the experiments conceptually or physically[7]. We first run the test with all the files and let XACML engine perform target matches with all the available policies and evaluate policies where the target matches. Figure 7 shows the result of this evaluation with about 1% of the policies being evaluated.

For comparison with our proposed system, we run the experiment with the same policy set with the PSS module included. We evaluate the TAPS using the index rule method for all the available policies and force the TAPS to be 1% of the total available policies. The resulting TAPS is stored in an array and the XACML engine then performs evaluation of all the files in this array. The combined time for determining the TAPS and evaluating it is shown in Figure 8. We include 1 percent of the total policies in the TAPS, which we believe is more than what most access requests would require, especially in systems with large number of policies. We chose this percentage so that we have a view of the worst case system performance and expect that most real systems will have fewer policies to evaluate per access request and the evaluation times will be lower that what is observed in Figure 8.

Comparing the results in Figure 7 and Figure 8, we observe that using TAPS evaluation with the index rules and then evaluating the applicable policies is about 4-8 times faster than the conventional method. This is specially important in large systems with a lot of policies. Considering the worst case scenario (10,000 policies, 8 rules/policy), the conventional evaluation takes about 210 seconds compared to 26 seconds on our system. In a more common scenario (100 policies, 8 rules/policy), the evaluation times are 1.8 seconds and 0.5 seconds respectively. We argue that this performance improvement is not only significant, but critical for real time systems.

6.2.4 Case 4

In this case, we fix the total number of available policies to 1000 and change the percentage of applicable policies to each access request. We perform this experiment with 15access request. We repeat this experiment for 1,2,4 and 8 rules per policy with and without the PSS system and compare their performance. The results are shown in Figure 9 and Figure 10. We observe that in our proposed model the system evaluation time starts from a very low value and increases linearly. On the other hand in existing systems, it starts at near maximum value and remains almost constant.

7. RELATED WORK

[7]Using target in the policy file is optional in XACML. If no target is used, the only way to check the applicability of the policy is to evaluate it and see if it applies to the current request. This will be slower than matching the target and hence we believe that our comparison is fair because we compare our results with the faster version.

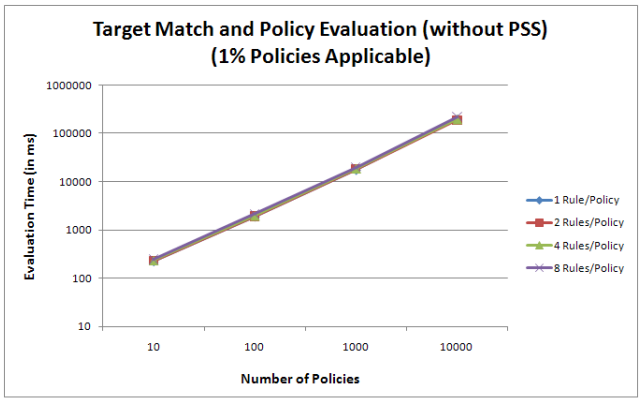

Figure 7: Evaluation time vs. number of total available policies (conventional XACML).

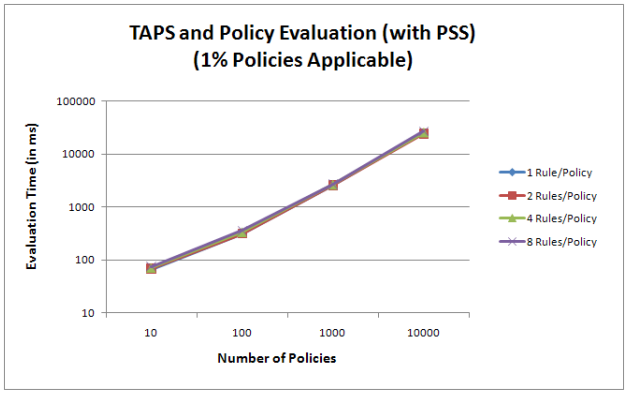

Figure 8: Evaluation time vs. number of total available policies (our proposed framework).

In this section, we review related work in the area of conflict detection, avoidance and resolution works and compare them to our proposed framework.

7.1 Conflict resolution

Mazzoleni, et. al, presented a system for integrating authorization policies for different partners organizations [20]. Their core idea is to find the similarity between a set of policies and to use that information to transform the set of policies into a single transformed policy which applies to the request. In their case, the PCA are static there is no way to choose policies dynamically, whereas in our framework we can choose the PCA dynamically. Our framework also allows multiple policies for the same resource, one of which can be chosen at run time.

Another idea for policy conflict resolution in active databases was proposed by Chomicki et. al, in [10]. Their system is based on the Event-condition-action paradigm in which policies are formulated using ECA rules. A policy generates a conflict when its output contains a set of actions that the policy administrator has specified cannot occur together. This work is specific to dynamically resolving conflicts among actions in a system, whereas our focus is more on a generic policy-based system to protect the resources. In our framework, the policy composers need not have any idea

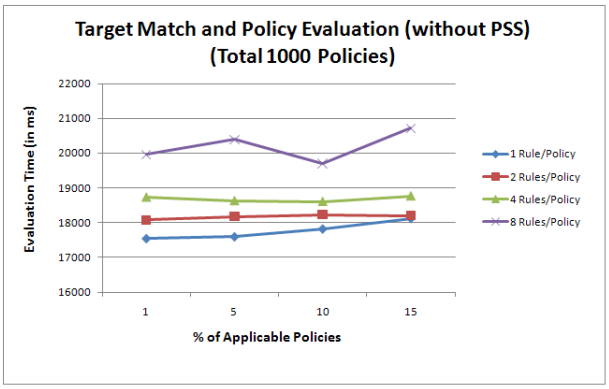

Figure 9: Evaluation time vs. number of total available policies (conventional XACML).

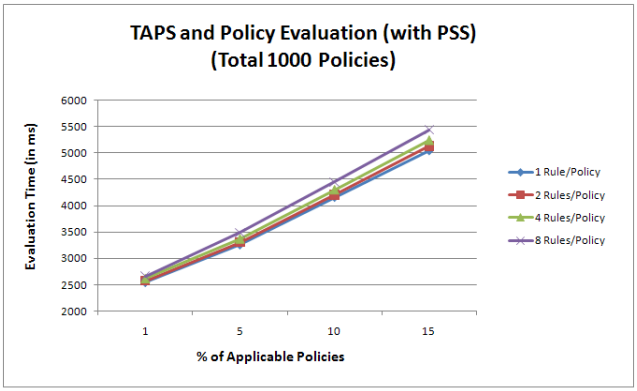

Figure 10: Evaluation time vs. number of total available policies (our proposed framework).

of the possible conflicts in the system, whereas in Chomicki the system administrator specifically defines conflicting actions. Moreover, in our system there can be a number of authorities who can compose the policies and it is not possible for any one authority to have an idea of all the possible conflicts in advance.

7.2 Conflict avoidance

One approach to avoid conflicts in authorization rules is presented by Yu et. al, in [26]. They argue that a large number of rules may apply to a service and detecting and resolving conflicts in real time can be a daunting task. Their system is completely static and assumes that is it always possible to determine priorities ahead of time and avoid conflicts. We argue that this is not possible in dynamic environments and is based on multiple factors like the context of the access request, authorities defining the policies, mandatory policies (like regulatory) vs. optional policies, and environmental factors.

Another approach for avoiding conflicts in policy specification is proposed by Agrawal, et. al, for defining authorization policies for hippocratic databases [5] and [6]. Their system allows system administrators to specify system policies for administration and regulatory compliance and these policies have the highest priority. Users are allowed to specify their privacy preference as long as their policies do not conflict with the system policies. In our framework, the users can specify their preferences even if they have conflicts with the other policies. The users policies may override other polices or be overridden based on context information. Agrawal's framework also does not consider changing system and regulatory policies that may create more conflicts with accepted user policies. Also, it may result in removal of conflicts between the new system policy and previously rejected user policies, which is not handled in this system. In our framework, this will be naturally handled without any action on anyone's part to resolve the conflict.

7.3 Hybrid Approach

Bertino, et. al, presented an approach which is a hybrid of conflict avoidance and conflict resolution [9]. In this work, the authors propose a scheme for supporting multiple access control policies in database systems. Here policies may have 'strong' authorization which are without conflicts or 'weak' authorization with possible conflicts. Compared to this framework, we believe that our approach is more generic because it allows conflicting policies to be composed and resolves conflicts based based on context information. To implement Bertino's proposed system, there should be some static hierarchy (or first specified rule overrides others) for conflict avoidance among strong authorizations. In contrast, our framework will allow dynamic overriding among the authorities.

Another approach to resolving policy conflicts in a hybrid manner is proposed by Jin, et al. [14]. In their work they mention that although resolving conflicts using the static method is easier, it may not be feasible in large systems with large number of policies. The main difference with our framework is that the combination algorithms in their model are defined statically, whereas in our case we decide the combination algorithm at run time based on context information. Also, our framework enables the user to add (remove) PCAs or policies dynamically, an aspect not considered in [14].

8. CONCLUSION

In this paper, we discussed policy-based authorization systems and attribute-based systems. We focus on the multi-authority case, where multiple policies are used to authorize a single access request. In particular, we expose the problems in choosing the PCAs ahead of time i.e. during the policy description. We present a framework to choose the PCA dynamically during run time based on dynamic attributes. The framework also supports choosing the applicable policy sets based on dynamic attributes. This increases the policy evaluation efficiency of the system and modularizes the policies enhancing their analyzability. Using dynamic attributes to determine applicable policy sets at run time provides a novel method to add and remove specialized policies dynamically. We implemented and evaluated a prototype of the authorization system as a module of a modified version of Sun's XACML engine.

9. REFERENCES

[1] Enterprise Privacy Authorization Language (EPAL). *http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/.*

[2] eXtensible Access Control Markup Language (XACML). *www.oasis-open.org/committees/xacml/.*

[3] Ldap authentication attributes. In *http://docs.sun.com/source/817-7647/ldapauth.htmlwp19608*.

[4] P3P: The Platform for Privacy Preferences. *http://www.w3.org/P3P/*.

[5] R. Agrawal, D. Asonov, R. Bayardo, T. Grandison, C. Johnson, and J. Kiernan. Managing disclosure of private health data with hippocratic databases. *IBM Research White Paper*, Januray 2005.

[6] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, pages 1013–1022, April 2005.

[7] A. Barth, J. Mitchell, and J. Rosenstein. Conflict and combination in privacy policy languages. In *Workshop on Privacy in the Electronic Society*, October 2004.

[8] E. Bertino, C. Brodie, S. B. Calo, L. F. Cranor, C. Karat, J. Karat, N. Li, D. Lin, J. Lobo, Q. Ni, P. R. Rao, and X. Wang. Analysis of privacy and security policies. *IBM Journal of Research and Development*, 53, 2009.

[9] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[10] J. Chomicki, M. J. Lobo, and S. Naqvi. Conflict resolution using logic programming. *IEEE Transactions on Knowledge and Data Engineering*, 15(1), Januray/February 2003.

[11] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change impact analysis of access control policies. In *International Conference on Software Engineering*, May 2005.

[12] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *IEEE Computer Security Foundations Workshop*, 2003.

[13] J. Jin, G.-J. Ahn, M. J. Covington, and X. Zhang. Toward an access control model for sharing composite electronic health record. In *4th International Conference on Collaborative Computing*, 2008.

[14] J. Jin, G.-J. Ahn, H. Hu, M. J. Covington, and X. Zhang. Patient-centric authorization framework for sharing electronic health records. In *SACMAT*, 2009.

[15] H. Kamoda, M. Yamaoka, S. Matsuda, K. Broda, and M. Sloman. Policy conflict analysis using free variable tableaux for access control in web services environments. In *WWW Conference*, 2005.

[16] H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *ACM Workshop on XML Security*, October 2003.

[17] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: Theory meets practice. In *ACM SACMAT*, 2009.

[18] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. In *IEEE Transactions on Software Engineering*, pages 852–869, Nov/Dec 1999.

[19] A. Masoumzadeh, M. Amini, and R. Jalili. Conflict detection and resolution in context-aware authorization. In *21st International Conference on Advanced Information Networking and Applications Workshops*, May 2007.

[20] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino. Xacml policy integration algorithms. In *ACM Transactions on Information and System Security (TISSEC)*, pages 852–869, February 2008.

[21] A. Mohan, D. Bauer, D. Blough, M. Ahamad, B. Bamba, R. Krishnan, L. Liu, D. Mashima, and B. Palanisamy. A patient-centric, attribute-based, source-verifiable framework for health record sharing. CERCS Tech Report GIT-CERCS-09-11, Georgia Tech, 2009.

[22] L. Ngo and A. Apon. Using shibboleth for authorization and authentication to the subversion version control repository system. In *IEEE ITNG*, 2007.

[23] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of xacml policies. In *CERIAS Tech Report 2008-21, Purdue University*, 2008.

[24] M. Rouached and C. Godart. Reasoning about events to specify authorization policies for web services composition. In *IEEE International Conference on Web Services (ICWS)*, September 2007.

[25] F. Turkmen and B. Crispo. Performance evaluation of xacml pdp implementations. In *ACM workshop on Secure Web Services*, October 2008.

[26] W. Yu and E. Nayak. An algorithmic approach to authorization rules conflict resolution in software security. In *Annual IEEE International Computer Software and Applications Conference*, July 2008.

APPENDIX

A. 'ALL-THAT-APPLY' COMBINATION ALGORITHM

Definitions:

$P_i = i^{th}$ Authorization policy.
FID = File Identifier.
FID(P_i) = File Identifier for i^{th} authorization policy file.
TAPS = An array to store FIDs. M-Policy = A policy file with index rules to define applicability of authorization policies.

Algorithm:

```
1   Load M-Policy, Access Request (AR)
2   Define TAPS, initialize i=0, counter=0
3   While (M-Policy (index rules))
4       decision = evaluate(index rule i) against AR
5       if (decision == permit)
6       {       { TAPS[counter++] = FID(Pi) }
7       else
8       {       { continue }
9       increment i
10  return TAPS
```

B. ALEX'S POLICY

```
- <Policy PolicyId="Alex's Policy" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  - <Description>
      This is Alex's policy to authorize access to his PHR
    </Description>
  - <Target>
    - <Subjects>
      - <Subject>
        - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">doctor</AttributeValue>
            <SubjectAttributeDesignator AttributeId="role" DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </SubjectMatch>
        </Subject>
      - <Subject>
        - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">paramedic</AttributeValue>
            <SubjectAttributeDesignator AttributeId="role" DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
    - <Resources>
        <AnyResource/>
      </Resources>
    - <Actions>
        <AnyAction/>
      </Actions>
    </Target>
  - <Rule RuleId="CommitRule" Effect="Permit">
    - <Target>
      - <Subjects>
        - <Subject>
          - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">doctor</AttributeValue>
              <SubjectAttributeDesignator AttributeId="role" DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </SubjectMatch>
          - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">florida</AttributeValue>
              <SubjectAttributeDesignator AttributeId="Alex-Location" DataType="http://www.w3.org/2001/XMLSchema#string"/>
```

```
            </SubjectMatch>
          </Subject>
        - <Subject>
          - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">paramedic</AttributeValue>
              <SubjectAttributeDesignator AttributeId="role" DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </SubjectMatch>
          - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">florida</AttributeValue>
              <SubjectAttributeDesignator AttributeId="requester-location" DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </SubjectMatch>
          - <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">florida</AttributeValue>
              <SubjectAttributeDesignator AttributeId="Alex-Location" DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
      - <Resources>
        - <Resource>
          - <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PHR</AttributeValue>
              <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" DataType="http://www.w3.or
              /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">read</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>
  </Policy>
```

Figure 11: An example policy for Alex.