Access Control Policy Combining: Theory Meets Practice

Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao
Purdue University, Department of Computer Science
305 N. University Street, West Lafayette, IN 47907,USA
{ninghui, qwang, wqardaji, bertino, prao}@cs.purdue.edu

Jorge Lobo
IBM T.J. Watson Research Center
Hawthorne, NY, USA
lobo@us.ibm.com

Dan Lin
Missouri University of Science and Technology
500 West 15th Street, Rolla, MO 65409
lindan@mst.edu

ABSTRACT

Many access control policy languages, e.g., XACML, allow a policy to contain multiple sub-policies, and the result of the policy on a request is determined by combining the results of the sub-policies according to some policy combining algorithms (PCAs). Existing access control policy languages, however, do not provide a formal language for specifying PCAs. As a result, it is difficult to extend them with new PCAs. While several formal policy combining algebras have been proposed, they did not address important practical issues such as policy evaluation errors and obligations; furthermore, they cannot express PCAs that consider all sub-policies as a whole (e.g., weak majority or strong majority). We propose a policy combining language PCL, which can succinctly and precisely express a variety of PCAs. PCL represents an advancement both in terms of theory and practice. It is based on automata theory and linear constraints, and is more expressive than existing approaches. We have implemented PCL and integrated it with SUN's XACML implementation. With PCL, a policy evaluation engine only needs to understand PCL to evaluate any PCA specified in it.

Categories and Subject Descriptors

D.4.6 [**Security and protection**]: Access controls

General Terms

Security, Language

Keywords

Policy Combination, XACML

1. INTRODUCTION

Many access control policy languages allow a policy to contain multiple sub-policies, and the effect of the policy on a request is determined by combining the effects of the sub-policies according to some algorithms. Examples of such policy languages include XACML [22], XACL [10], EPAL [1], SPL [17], and firewall policies. Existing languages usually specify a fixed set of policy combining algorithms (PCAs), but none of them provides a formal language for specifying new PCAs. For example, firewall policy languages typically use first-applicable algorithm to combine results from rules; that is, the decision for a request is decided by the first rule applicable to the request. Among existing policy languages, XACML offers the most flexible approach which consists of four strategies: deny-overrides, permit-overrides, first-applicable, and only-one-applicable. However, many other desirable strategies exist. For example, a common strategy for combining two policies is to permit a request only when both sub-policies permit it. This simple and intuitive strategy cannot be specified in XACML. In addition, the following strategies cannot be specified in XACML.

Weak-consensus. Sub-policies should not conflict with each other: Permit a request if some sub-policies permit a request, and no sub-policy denies it. Deny a request if some sub-policies deny a request, and no sub-policy permits it. Yield a value indicating conflict if some permit and some deny.

Strong-consensus. All sub-policies must agree: Permit a request if all sub-policies permit a request. Deny a request if all sub-policies deny a request. Yield conflict otherwise.

This differs from weak-consensus because a sub-policy may neither permit nor deny a request, i.e., it may not be applicable to the request. When some sub-policies permit a request and others are not applicable to it, weak consensus permits the request, but strong consensus yields conflict.

Weak-majority. A decision (permit or deny) wins if it has more votes than the opposite. Permit (deny, resp.) a request if the number of sub-policies permitting (denying, resp.) the request is greater than the number of sub-policies denying (permitting, resp.).

Strong-majority. Permit a request if over half of all sub-policies permit it, and deny the request if over half deny it.

Super-majority-permit. Permit a request if over $2/3$ of all policies permit it, and deny the request otherwise.

XACML has become the *de facto* standard for specifying access control policies for various applications, especially web services. Extensibility and flexibility of policy combining are thus desirable to meet the needs of these applications. XACML explicitly allows additional user-defined combining algorithms. However, it does not provide a standard approach (or a specification language) for doing so. Before any PCA can be used, one needs to make sure that existing policy evaluation engines can handle the PCA. Without a formal specification language, automated processing is impossible, and each new PCA must be hard-coded by programmers in every policy evaluation engine. This makes wide deployment of user-defined PCAs infeasible in practice.

In the mean time, there have been a number of theoretical studies in formal approaches for specifying policy combining behaviors [6, 8, 3, 11, 20, 21]. All these studies take an algebraic approach, where unary and binary algebraic operators are defined to combine sub-policies. These theoretical studies, however, do not address several important issues that are encountered in practice. First, one issue that significantly complicates policy combining is policy evaluation errors. This is exemplified by XACML, as we will show in Section 2. Evaluating a policy on a request may result in errors due to invalid or missing information for example.These issues are not considered in existing theoretical studies, yet a real system (such as XACML) does not have the luxury to ignore them. Second, when policies also have obligations, policy combining must consider their effects. As we will show, combining decisions is intimately connected with the combining of obligations and they need to be considered together. With the exception of [3], all existing algebras do not consider combining obligations. Third, a policy combining algebra defines unary and binary operators. This has two limitations. One is that it does not specify a combining behavior that can be applied to an unbounded (but finite) number of sub-policies (such as the XACML combination strategies do). This can be solved by introducing a notion of closure to a binary operator, which we use in this paper. The second limitation is that the operators provide only a local view, and cannot specify strategies such as weak-majority or strong-majority.

Motivated by the gap in both theory and practice of access control policy combining, we propose the Policy Combining Language (PCL), which can succinctly and precisely express a variety of PCAs. PCL uses two novel approaches for specifying PCAs, a local-view approach that extends a binary policy combining operator to combine an unbounded number of sub-policies, and a global-view approach that uses linear constraints on the number of sub-policies that permit or deny a request. Using PCL, one can specify all the standard combining algorithms in XACML as well as the combining algorithms described earlier. PCL handles policy errors by viewing them as uncertainties about evaluation outcomes, and tries to resolve these uncertainties whenever possible. PCL also allows flexible specification of obligation combining behaviors. With PCL, the policy evaluation engine now only needs to understand one language and can then evaluate policies using any PCA specified in this language. We have implemented PCL and were able to integrate it with SUN's XACML implementation [19]. Our contributions are summarized as follows:

- PCL advances the theory of access control policy combining in several ways. It is more expressive than previous algebras. It systematically treats policy evaluation errors as uncertainty over a set of possible values. It also considers combining obligations as well as decisions. Though PCL is primarily motivated by XACML, the underlying formalism and approaches can be applied to other languages.
- On the practice side, we provide a detailed and principled analysis of policy combining in XACML and identify several problems. We also show that PCL can be implemented and integrated with XACML, demonstrating the feasibility of being included in future XACML standard.

The rest of this paper is organized as follows. In Section 2, we analyze PCAs in XACML. In Sections 3 and 4, we introduce the language PCL. Discussions of implementations and other issues are in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. POLICY COMBINATION IN XACML

XACML [22] is the OASIS standard language for the specification of access control policies. In this section, we describe and analyze how XACML handles policy combining. Our descriptions are based on XACML 2.0 [22]. The purposes of this section are multiple. First, we want to illustrate the complexity and subtleties of dealing with practical issues such as policy evaluation errors and obligations (which XACML does). Second, we want to come up with principles that can explain why XACML does things in certain ways. These principles will guide the design of our PCL. Third, we will discuss some of XACML's decisions that we consider to be anomalies. These decisions are counter to the principles underlying other decisions in XACML.

Rules, policies, and policy-sets. XACML defines three policy elements: rules, policies, and policy-sets. A *rule* is the most basic policy element; it has three main components: a *target*, a *condition*, and an *effect*. The target defines a set of subjects, resources and actions that the rule applies to; the condition specifies restrictions on the attributes in the target and refines the applicability of the rule; the effect is either `Permit`, in which case we call the rule a *permit rule*, or `Deny`, in which case we call it a *deny rule*. If a request satisfies both the rule target and rule condition, the rule *is applicable* to the request and yields the decision specified by the effect element; otherwise, the rule *is not applicable* to the request and yields the decision `NotApplicable`.

A *policy* consists of four main components: a *target*, a *rule-combining algorithm (RCA)*, a set of *rules*, and *obligations*. The policy target decides whether a request is applicable to the policy and it has similar a structure as the rule target. The RCA specifies how the decisions from the rules are combined to yield one decision. The *obligations* represent functions to be executed in conjunction with the enforcement of an authorization decision.

A *policy-set* also has four main components: a *target*, a *policy-combining algorithm (PCA)*, a set of *sub-policies*, and *obligations*. A sub-policy can be either a policy or a policy-set. The PCA specifies how the results of evaluating the sub-policies are combined to yield one decision.

`PDP`, `PEP`, and `PIP`. In XACML there are the Policy Enforcement Point (`PEP`), the Policy Decision Point (`PDP`), and the Policy Information Point (`PIP`). When an access request is received, the `PEP` sends the request to the `PDP`. To handle the request, the `PDP` may need additional information about subjects, resources, actions and environment attributes from the `PIP`. Once the `PDP` receives the required information from the `PIP`, it makes a decision according to the policies and returns the result to the `PEP`.

In XACML, a rule, a policy, or a policy-set returns one of the following four decisions for each request: P (`Permit`), D (`Deny`), NA (`NotApplicable`), and IN (`Indeterminate`). The value IN occurs when there is a policy evaluation error. Several kinds of errors may occur during policy evaluation. Some are due to network communication and database querying. For example, the `PIP` may be down and thus unable to answer queries from the `PDP`. Some are due to erroneous policies. For example, the condition of a rule may perform a division by 0. Others are due to missing attributes. For example, evaluating a policy that denies a request if the subject's income level is below a threshold yields an error if the income information is not provided. When this policy is a sub-policy of a policy-set that permits a request if the subject is a member of a particular club, the income information is not necessary for the overall policy to permit the request. Hence, a requestor may provide only proof of his club membership but no information about his income, resulting in an evaluation error. In all the above cases, the PDP does not have enough information to determine whether a policy applies to the access request. It is the responsibility of the PCA to handle such errors.

The PDP returns a value in $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$ to the PEP, and the PEP decides whether to permit or deny a request. XACML defines three types of PEPs: *Base PEP* , *Deny-based PEP* , and *Permit-based PEP* . All PEPs yield the permit (or deny) decision if the value returned by the PDP is P (or D). The difference among these PEPs lies in the way they handle NA and IN. The Base PEP's behavior is undefined when receiving NA or IN. The Deny-based PEP treats NA and IN as Deny, while the Permit-based PEP treats NA and IN as Permit.

2.1 Rule and Policy Combining Algorithms

XACML has five standard RCAs and six standard PCAs. They are *"Deny-overrides"*, *"Ordered-deny-overrides"*, *"Permit-overrides"*, *"Ordered-permit-overrides"*, *"First-applicable"* and *"Only-one-applicable"* ("Only-one-applicable' is only defined as a PCA). Ordered-deny-overrides and ordered-permit-overrides are the same as deny-overrides and permit-overrides, respectively, except that rules and policies have to be evaluated in the order they appear. While the intuition of these combining algorithms are easy to understand and formalize, they have subtle details which are often ignored by existing attempts to formalize them.

Permit-overrides RCA. The "Permit-overrides" RCA prefers P to D to NA. That is, if any rule evaluates to P, the result is P; otherwise if no rule evaluates to P, and some rule evaluates to D, the result is D; finally if all rules evaluate to NA, the result is NA. We write this as $\mathsf{P} > \mathsf{D} > \mathsf{NA}$.

When the evaluation of a rule encounters an error, this rule's result is IN. The treatment of IN, however, depends on whether IN is from a permit rule or a deny rule. When IN is from a permit rule, the RCA uses $\mathsf{P} > \mathsf{IN} > \mathsf{D} > \mathsf{NA}$, and when IN is from a deny rule, the RCA uses $\mathsf{P} > \mathsf{D} > \mathsf{IN} > \mathsf{NA}$. The reason for this difference is not explained in the official XACML standard specifications. We believe that it is due to the following implicit principle in XACML.

PRINCIPLE 1. *When a permit rule gives* IN*, this is treated as an uncertainty over* $\{\mathsf{P}, \mathsf{NA}\}$*. When combining an uncertain value* $\{\mathsf{P}, \mathsf{NA}\}$ *with* $x \in \{\mathsf{P}, \mathsf{D}, \mathsf{NA}\}$*, use the following approach: if combining* P *with* x *and combining* NA *and* x *both give* x*, the combining outcome is* x*; otherwise, the outcome is* IN.

The rationale is that when a rule evaluates to an error, then either this rule is applicable or it is not. These should be treated as two possible worlds. If the same decision is reached in both worlds, then the combining outcome is that decision; otherwise, the outcome should be IN. The approach for handling a deny rule is similar. Any existing attempt on using multi-valued algebra or logic to formalize XACML will have difficulty dealing with the situation that the same value IN behaves differently in different contexts.

Permit-overrides PCA. XACML "Permit-overrides" PCA combines outcomes from policies. When combining policies (rather than rules), PDP does not know the source of an error and hence XACML chooses $\mathsf{P} > \mathsf{D} > \mathsf{IN} > \mathsf{NA}$, one of the two orderings used in Permit-overrides RCA. However, one could argue that $\mathsf{P} > \mathsf{IN} > \mathsf{D} > \mathsf{NA}$ is a more natural choice. Recall that the cause of an IN could be uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$, preferring D over IN goes against what the name "Permit-overrides" suggests.

Deny-overrides RCA. The "Deny-overrides" RCA is similar to the "Permit-overrides" RCA, except that the preference is $\mathsf{D} > \mathsf{P} > \mathsf{NA}$. The IN is also treated similarly as uncertainty. That is, if IN is from a permit rule, we have $\mathsf{D} > \mathsf{P} > \mathsf{IN} > \mathsf{NA}$, and if IN is from a deny rule, we have $\mathsf{D} > \mathsf{IN} > \mathsf{P} > \mathsf{NA}$.

Deny-overrides PCA. The "Deny-overrides" PCA also uses $\mathsf{D} > \mathsf{P} > \mathsf{NA}$; however, it treats IN very differently. The PCA treats IN as always equivalent to D.

There are a number of issues with this design. First, this is very different from how Permit-overrides PCA treats IN. In all other aspects, Permit-overrides and Deny-overrides are symmetric. We believe that this asymmetry will be unexpected for many policy authors. Second, as we will discuss in Section 2.2, this design is incompatible with the rationale for obligation propagation. Third, the following example illustrates cases that one logically expects a request to be permitted, but will be denied.

Example 1. Consider a request q and the following policy-set:

$$S = \text{Deny-PCA}(P_1 = \text{Deny-RCA}(R_1, R_2), \quad P_2),$$

where S consists of two policies P_1 and P_2, and P_1 consists of two rules R_1 and R_2. Suppose that R_1 is a deny rule that does not apply to q, R_2 is a permit rule the evaluation of which on q encounters an error, and P_2 is a policy that permits q. The policy-set S denies the request via the following evaluation:

$$\begin{aligned} S = &\ \text{Deny-PCA}(\text{Deny-RCA}(\mathsf{NA}, \{\mathsf{P}, \mathsf{NA}\}),\ \mathsf{P}) \\ = &\ \text{Deny-PCA}(\mathsf{IN}, \mathsf{P}) \\ = &\ \mathsf{D} \end{aligned}$$

However, this is undesirable for the following reasons. Among R_1, R_2, and P_2, the only deny rule (R_1) does not apply to q, and there is a policy (P_2) that permits q. There is no reason for denying the request! This becomes clearer if we consider the facts that (1) the *cause* of the value IN is the uncertainty of whether R_2 is applicable or not, and (2) in either case S should permits the request. When R_2 is applicable, R_2 permits the request, and so does P_1. Since both P_1 and P_2 permit the request, so should S. When R_2 is not applicable, the policy P_1 is not applicable. Since P_2 permits the request, S should permit the request.

First-applicable. The "First-applicable" RCA returns the effect of the first applicable rule as the result if no errors occur. Whenever an error occurs during rule evaluation, the RCA returns IN.

One could argue that the "First-applicable" RCA should be defined in a different way. Following the spirit that an error occurring in a permit rule means uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$, if the next rule permits the request (i.e. it has a permit effect and is applicable), the policy should return P (rather than IN), as the request will be permitted whether the first rule applies or not.

The above argument should not be used with the "First-applicable" PCA, because the first sub-policy, which results in IN, may have obligations different from the second sub-policy. In this case, returning IN appears to be the simplest decision. This issue does not exist for RCA, because a rule does not have obligations.

Only-one-applicable. The "Only-one-applicable" PCA returns the effect of the unique policy in the policy-set that applies to the request. If there are more than one applicable policies, the PCA reports the conflict by returning IN. Furthermore, if an error occurs during evaluation of any policy, the PCA also returns IN.

Empty policies. When a policy or a policy-set is empty, the combining result is always NA, according to the pseudo-code of the combining algorithms in XACML.

2.2 Obligations in XACML

When combining sub-policies, one also needs to determine how the obligations with these sub-policies are combined. XACML handles obligations as follows [22]

> If the PDP's evaluation is viewed as a tree of policy-sets and policies, each of which returns "Permit" or "Deny", then the set of obligations returned by the PDP to the PEP will include only the obligations associated with those paths where the effect at each level of evaluation is the same as the effect being returned by the PDP.

We point out that XACML's PCAs may not evaluate all sub-policies. For example, a Permit-overrides PCA will return P as soon as one sub-policy gives P. The evaluation tree in the above quotation refers to such an incomplete tree.

Obligations on some branches are not propagated, intuitively because they are overruled by other branches. This approach can be explained using the following principle.

PRINCIPLE 2. *Only when a policy (or policy-set) returns* P *or* D*, will it carry obligations. When a policy returns* P *or* D*, it depends upon all evaluated sub-policies that have the* same *decisions. Obligations from these dependent sub-policies are propagated up.*

For the above rationale to hold, the implicit assumption is that when a policy returns P or D, it is due to some of its sub-policies that return the same decision. (Otherwise, this policy-set is making a decision sort of "out of the blue", as reflected by the fact that no obligation from any sub-policy is propagated up.) This assumption, however, is violated by "Deny-overrides" PCA, which will return D when encountering IN and will propagate no obligations. As a result, a policy-set could return D even when all the rules contained in sub-policies are permit rules.

We argue that XACML's "Deny-overrides" PCA causes too much anomaly that it should be changed to $\mathsf{D} > \mathsf{IN} > \mathsf{P} > \mathsf{NA}$. We doubt that this change would adversely affect any existing policy, as it seems unlikely that policy authors would rely on the abnormal behavior in expressing their policies.

3. PCL WITHOUT OBLIGATIONS

Our design requirements for the Policy Combining Language PCL are as follows.

1. One should be able to specify the exact behavior of current XACML PCAs and RCAs, for backward compatibility purpose, which we believe is critical for ease of deployment and adoption of PCL in XACML. PCL achieves compatibility with the exception of "Deny-overrides" PCA, which we have argued to be problematic in several ways. Deny-overrides interpreted as $\mathsf{D} > \mathsf{IN} > \mathsf{P} > \mathsf{NA}$ or $\mathsf{D} > \mathsf{P} > \mathsf{IN} > \mathsf{NA}$ can be specified in PCL.
2. One should be able to specify some of the alternatives to XACML's PCAs that we argue to be more natural (e.g., $\mathsf{P} > \mathsf{IN} > \mathsf{D} > \mathsf{NA}$ for permit-overrides PCA). And one should be able to specify PCAs that consider the root cause of IN and resolve it accordingly. For example, one should be able to specify deny-overrides in a way to permit the request as shown in Example 1.
3. One should be able to specify other natural PCAs, such as the ones presented in the beginning of Section 1.
4. The specification should be concise, precise, and user-friendly. It should be based on sound theory, with their expressive power clearly understood.

In the rest of this section, we present the part of PCL that does not deal with obligations. How obligations are handled in PCL will be presented in Section 4. Some policy combining situations do not involve obligations, in which case the approach described in this section can be applied. In the rest of this section, we use PCA to refer to both RCA and PCA, which are specified in the same way when not considering obligations.

3.1 Overview of PCL

PCL also uses $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. In PCL evaluation errors can be represented as *uncertainty* denoted by non-singleton subsets of Σ. For example, if an error occurs when evaluating a permit rule, the result will be $\{\mathsf{P}, \mathsf{NA}\}$ which means that it is uncertain whether this permit rule is applicable or not to the request. Similarly, a deny rule evaluates to $\{\mathsf{D}, \mathsf{NA}\}$ when an error occurs.

The value IN is used for two purposes. In one purpose, IN represents a conflict between two policies (or rules) that a PCA chooses not to resolve but to expose to a higher level. Some examples are: (1) when the only-one-applicable PCA is used, there will be a conflict if more than one policies are applicable; (2) when the weak-consensus PCA is used, there will be a conflict if one policy permits while another denies; (3) when the strong-consensus PCA is used, there will be a conflict if one policy permits while another is not applicable. In the second purpose, IN represents a policy evaluation error that one does not want to expose the uncertainty to a higher level.

Intuitively, a PCA combines a sequence of results from sub-policies into one of the four outcomes. Ignoring the issue of policy evaluation errors for now, a PCA is a function $\Sigma^* \rightarrow \Sigma$, where

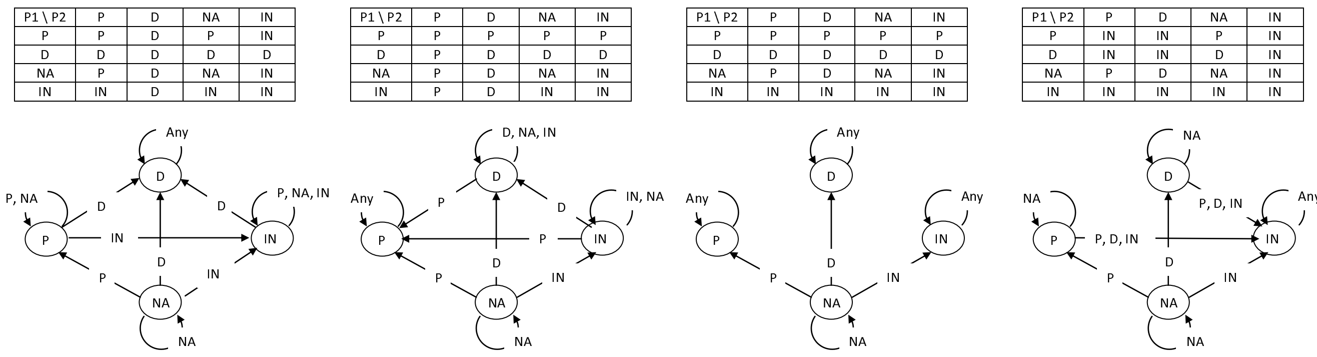
$$\Sigma^* = \{\epsilon\} \bigcup \Sigma \bigcup \Sigma \times \Sigma \bigcup \Sigma \times \Sigma \times \Sigma \bigcup \cdots$$

where ϵ is the empty string. Many approaches exist for specifying such functions. PCL uses two approaches with complementary strengths: a PCA either extends a binary Policy Combining Operator, or uses linear constraints on the numbers of sub-policies that return P, D, IN, NA, respectively. The former can specify all strategies in XACML as well as weak-consensus and strong-consensus, but cannot specify counting-based strategies such as weak-majority or strong-majority. The latter can specify all the strategies we have discussed so far, except for first-applicable.

3.2 Using Policy Combining Operators

Most standard PCAs in XACML are easily explained when combining just two sub-policies. We call a PCA that combines only two sub-policies a policy combining operator (PCO). A PCO g can be represented as a binary operator g over the four values $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$, i.e., $g : \Sigma \times \Sigma \rightarrow \Sigma$. Such a binary operator can be expressed using a matrix. Examples of the four standard PCAs in XACML are shown in Figure 1. (For deny-overrides, we choose a variant; the exact XACML version can be encoded using a matrix as well.)

While a matrix has 16 cells, in practice one does not need to give 16 pieces of information to specify a PCO. More compact representations are possible. One can specify some matrices using an ordering, e.g., $\mathsf{P} > \mathsf{D} > \mathsf{IN} > \mathsf{NA}$ for permit overrides. Or, one can exploit the fact that some matrices are symmetric. One can also exploit the fact that all cells in some row or column of a matrix have the same value. Our proposed XML encoding of PCL uses these approaches to enable more succinct specifications of PCOs. (See Figure 5 on Page for the BNF representation of the XML encoding



P1 \ P2	P	D	NA	IN
P	P	D	P	IN
D	D	D	D	D
NA	P	D	NA	IN
IN	IN	D	IN	IN

(a) Deny-overrides

P1 \ P2	P	D	NA	IN
P	P	P	P	P
D	P	D	D	D
NA	P	D	NA	IN
IN	P	D	IN	IN

(b) Permit-overrides

P1 \ P2	P	D	NA	IN
P	P	P	P	P
D	D	D	D	D
NA	P	D	NA	IN
IN	IN	IN	IN	IN

(c) First-applicable

P1 \ P2	P	D	NA	IN
P	IN	IN	P	IN
D	IN	IN	D	IN
NA	P	D	NA	IN
IN	IN	IN	IN	IN

(d) Only-one-applicable

Figure 1: Figure of several PCOs in matrix and the corresponding general PCAs in the form of DFA.

of PCL. We use BNF rather than the actual XML schema because BNF is more compact.)

From PCO to PCA. In general, the number of policies a PCA needs to combine cannot be bounded at the time when the PCA is specified. Therefore, we propose the following approach to construct a general PCA from a PCO.

DEFINITION 1. *Let* $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. *Given a PCO* $g : \Sigma \times \Sigma \to \Sigma$, *its* recursive *PCA is the function* $f : \Sigma^* \to \Sigma$ *defined as follows:*

- $f(\epsilon) = \mathsf{NA}$
- $f(x) = x$
- $f(x_1, x_2) = g(x_1, x_2)$
- $f(x_1, \ldots, x_n) = g(f(x_1, \ldots, x_{n-1}), x_n)$, *for* $n > 2$.

According to the above definition, combining a sequence of sub-policies proceeds from the first to the last. The results of the first two are combined, the outcome of the combination is then combined with the third, and so on.

One can view the evaluation of a policy with such a PCA for a request q as a deterministic finite automaton (DFA). The DFA has Σ as its set of input symbols, and $\Sigma \cup \{\mathsf{S}\}$ as its set of states, where S denotes the start state. The transitions out of S are fixed for all PCAs; on symbol $x \in \Sigma$, it will go to the state x. After at least one input symbol, the current state is in Σ and represents the result of combining so far. The next input symbol is the result of the next rule or policy on q. When the evaluation terminates, the state of the DFA is the result of the combination. Examples of the DFA-representation of four standard PCAs in XACML are shown in Figure 1, where the start state is omitted in for clarity. Given an empty policy, the DFA will end at the start state, in which case the combining outcome is defined to be NA.

All states in the DFA are accept states; however, on which accept state a DFA stops is significant. Such a DFA simultaneously defines four languages, each corresponding to a state in $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. For instance, if a string in Σ^+ leads the DFA to end at state P (meaning the result of combination is to permit the request), then that string is in the language corresponding to P.

Handling evaluation errors. Recall that we treat errors as uncertainties and allow each rule/policy/policy-set to evaluate to a subset of $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. The following definition specifies how to combine results that represent uncertainty.

DEFINITION 2. *Let* $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. *Let* Γ *denote the set containing all non-empty subsets of* Σ, *i.e.,* $\Gamma = 2^{\Sigma} \setminus \{\emptyset\}$. *The PCO* g *is extended to be a function* $g : \Gamma \cup \{\emptyset\} \times \Gamma \to \Gamma$, *defined as follows.*

$$\begin{array}{lll} g(\gamma_1, \gamma_2) & = \{g(x_1, x_2) \mid x_1 \in \gamma_1 \wedge x_2 \in \gamma_2\}, & \textit{for } \gamma_1, \gamma_2 \in \Gamma \\ g(\emptyset, \gamma) & = \gamma, & \textit{for } \gamma \in \Gamma \end{array}$$

For example, assume that g is a "Deny-overrides" PCO (its matrix representation is shown in Figure 1). We have

$$\begin{array}{lll} g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{D}\}) & = \{g(\mathsf{P}, \mathsf{D}), g(\mathsf{NA}, \mathsf{D})\} & = \{\mathsf{D}\} \\ g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{P}\}) & = \{g(\mathsf{P}, \mathsf{P}), g(\mathsf{NA}, \mathsf{P})\} & = \{\mathsf{P}\} \\ g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{NA}\}) & = \{g(\mathsf{P}, \mathsf{NA}), g(\mathsf{NA}, \mathsf{NA})\} & = \{\mathsf{P}, \mathsf{NA}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{D}\}) & = \{g(\mathsf{D}, \mathsf{D}), g(\mathsf{NA}, \mathsf{D})\} & = \{\mathsf{D}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{P}\}) & = \{g(\mathsf{D}, \mathsf{P}), g(\mathsf{NA}, \mathsf{P})\} & = \{\mathsf{D}, \mathsf{P}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{NA}\}) & = \{g(\mathsf{D}, \mathsf{NA}), g(\mathsf{NA}, \mathsf{NA})\} & = \{\mathsf{D}, \mathsf{NA}\} \end{array}$$

It is worth noting that the behavior of the deny-overrides and permit-overrides RCA in XACML (as we discussed in Section 2.1) can be explained using the above approach.

The behavior of combining while treating errors as uncertainties can still be modeled using a Finite State Automaton (FSA). However, now each input to be combined may be a set consisting of two or more values, e.g., $\{\mathsf{P}, \mathsf{NA}\}$. As a result, the next state may also be uncertain. We call this an uncertain-input FSA. The behavior of such an FSA can be equivalently simulated by a DFA with 16 states and 15 input symbols, where each state is a subset of $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$ and each input is a non-empty subset of Σ. The initial state is the empty set $\emptyset$, and the state transitions can be defined according to Definition 2. Given a state γ_1 and the input γ_2, the new state is given by $g(\gamma_1, \gamma_2)$. All 16 states are accept states. Each non-empty state defines a different language over Σ^+.

We observe that an uncertain-input FSA is conceptually related to but different from a non-deterministic finite state automaton (NFA), whose state transition is non-deterministic. In an uncertain-input FSA, the state transition relation is deterministic, but we may have uncertain inputs. Converting an NFA to DFA increases the number of states, but not the number of input symbols. Converting an uncertain-input FSA to a DFA increases both the number of states and the number of input symbols.

We stress that our approach is different from an approach using 15 values. Our approach requires a 4-by-4 matrix for combining the four basic values, and infer the combining behavior of other, uncertain values. In a 15-valued approach, one specifies a 15-by-15 matrix. Our approach implicitly rules out many 15-by-15 matrices as illegal, and represent the legal ones compactly.

3.3 Using Linear Constraints

We have introduced the approach of specifying PCAs by naturally extending a binary PCO. This approach, however, cannot express PCAs that consider the combination of sub-policies as a whole, rather than through a step-by-step process of combining two results. In particular, this approach cannot express counting-based strategies such as weak-majority or strong-majority.

This motivates PCL's second approach for PCA specification, which uses linear constraints on the number of sub-policies that return P, D, NA, and IN.

DEFINITION 3. *A* linear constraint *is an expression that connects a number of linear equations or inequations on variables $\#\mathsf{P}$, $\#\mathsf{D}$, $\#\mathsf{NA}$, and $\#\mathsf{IN}$ using conjunctive operator $\wedge$ and disjunctive operator $\vee$, where $\#\mathsf{P}$, $\#\mathsf{D}$, $\#\mathsf{NA}$, and $\#\mathsf{IN}$ stand for the number of sub-policies that return* P, D, NA, *and* IN, *respectively.*

For example, the linear constraint $\#\mathsf{P} \geq 1$ is satisfied when at least one sub-policy returns P, while $\#\mathsf{D} > (\#\mathsf{P} + \#\mathsf{NA} + \#\mathsf{IN})$ requires that the number of sub-policies that return D is greater than the number of sub-policies that return other results (in other words, more than half of the sub-policies return D).

A PCA can be specified by associating a linear constraint with each of P, D, and IN. We require the three constraints associated with P, D, and IN be disjoint with each other (i.e. two constraints cannot be both satisfied). Checking whether two linear constraints ϕ_1 and ϕ_2 can be both satisfied can be performed using existing techniques [2, 7]. To combine a number of sub-policies, we count the number of sub-policies that evaluate to P, D, NA, and IN, respectively, and then check the linear constraints. If the linear constraint associated with P (resp. D or IN) is satisfied, the output of the PCA is P (resp. D or IN). If none of the constraint is satisfied, the output of the PCA is NA.

Figure 2 shows how to use linear constraints to specify deny-overrides, only-one-applicable, weak-consensus, and strong-majority. Other PCAs mentioned in this paper whose evaluation results do not depend on the order of sub-policies, such as permit-override, strong-consensus, and weak-majority, can also be specified using linear constraints. However, the linear constraint approach cannot specify "first-applicable", as the evaluation of linear constraints does not take the order of policies into account. We will discuss the expressive power of the linear constraint approach in Section 3.5.

Handling evaluation errors. Similar to the PCO approach, errors are treated as uncertainties. To handle evaluation errors, instead of keeping counters for the four values P, D, NA, and IN, we keep a counter for each of the fifteen non-empty subsets of $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$. Whenever we receive a (possibly uncertain) value, one increases the counter corresponding to it.

After evaluating all the sub-policies, when no uncertain value has been encountered, only one final state is possible; otherwise we enumerate all possible final states of the four counters P, D, NA, and IN. For example, assume that the value of the counter for $\{\mathsf{P}, \mathsf{NA}\}$ is 2, while all other counters are 0. In this case, the three possible states are $\langle \#\mathsf{P} = 2\rangle$, $\langle \#\mathsf{P} = 1, \#\mathsf{NA} = 1\rangle$, and $\langle \#\mathsf{NA} = 2\rangle$. Then, we evaluate every possible state against the linear constraints in the PCA, and every decision whose constraint is satisfied is included in the combination result of the PCA. For instance, assume that the "only-one-applicable" PCA is used. For the three states in the earlier example, $\langle \#\mathsf{P} = 2\rangle$ satisfies the constraint for IN (which is $\#\mathsf{P} > 1 \vee \#\mathsf{D} > 1 \vee \#\mathsf{IN} > 0$), $\langle \#\mathsf{P} = 1, \#\mathsf{NA} = 1\rangle$ satisfies the constraint for P (which is $\#\mathsf{P} = 1 \wedge \#\mathsf{D} = 0 \wedge \#\mathsf{IN} = 0$), and $\langle \#\mathsf{NA} = 2\rangle$ does not satisfy any constraint and thus results in NA. Therefore, the combination result of the PCA is an uncertainty set $\{\mathsf{P}, \mathsf{NA}, \mathsf{IN}\}$.

3.4 Additional Details of PCL

We have introduced the foundations of PCL. While our design of PCL is motivated by XACML, it can be used in any policy language that needs to combine a possibly unbounded number of policies and/or needs to consider policy evaluation errors. Next, we will discuss some additional issues that arise mostly for compatibility considerations with XACML.

Order-preserving evaluation. XACML has ordered deny-overrides and permit-overrides. To be compatible, PCL provides a boolean flag to specify whether the evaluation must follow the order of the original rules (policies).

Pre-processing and post-processing. We use uncertainty to handle errors occurring at the time of evaluation. However, sometimes, one may want to specify PCAs that do not treat errors as uncertainties and handle errors in their own ways, such as several PCAs in XACML do. To specify those PCAs, we introduce a pre-processing boolean flag, which allows one to map all uncertain input values to IN before feeding them to the PCA. A PCA with pre-processing enabled can define its own error handling scheme by specifying how to combine IN with other input values.

Also, we introduce a post-processing boolean flag, which allows one to map any uncertain output value to IN. While the pre-processing step applies to each input result to be combined, the post-processing step applies to the final combining result. By mapping any uncertain final result to IN, post-processing prevents upper-level PCAs to handle an error returned by the current PCA as uncertainty. Note that when the pre-processing option is selected, post-processing has no effect. The reason is that a PCA with pre-processing will not have any uncertain input and thus will not produce an uncertain output.

In general, by introducing pre-processing and post-processing, one can specify three levels of uncertainty handling in PCL: (1) no uncertain input, (2) allow uncertain input, but not uncertain output, (3) allow both uncertain input and uncertain output.

3.5 The Expressive Power of PCL

A natural question that arises is: How expressive is PCL? This question has both practical and theoretical relevance. We now study the expressive power of PCL. As we will see, in PCL, the PCO-based approach and the linear-constraint-based approach have different expressive powers. While there are PCAs that cannot be expressed using either, all the ones that we consider to be *natural* can be expressed. For clarity, we assume that policy evaluation errors are always handled by uncertainty and ignore policy evaluation errors in the discussions below. To understand what PCAs can be expressed in PCL and what cannot, we need a definition of PCAs independent of PCL. This is given below.

DEFINITION 4. *A PCA is given by a tuple of four languages $\langle L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{IN}}\rangle$ over the alphabet of $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$, satisfying the conditions that they are mutually disjoint and $\Sigma^+ = L_{\mathsf{P}} \cup L_{\mathsf{D}} \cup L_{\mathsf{NA}} \cup L_{\mathsf{IN}}$.*

For example, the first-applicable PCA can be expressed using regular expression syntax as:

$$\begin{array}{lll} \langle & L_{\mathsf{P}} = \mathsf{NA}^*\,\mathsf{P}\,\Sigma^*, & L_{\mathsf{D}} = \mathsf{NA}^*\,\mathsf{D}\,\Sigma^*, \\ & L_{\mathsf{NA}} = \mathsf{NA}^*, & L_{\mathsf{IN}} = \mathsf{NA}^*\,\mathsf{IN}\,\Sigma^* \quad \rangle. \end{array}$$

DEFINITION 5. *We say a PCA $\langle L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{IN}}\rangle$ is* regular *if and only if the languages $L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{IN}}$ are regular.*

DEFINITION 6. *We say a PCA $\langle L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{IN}}\rangle$ is* order-insensitive *if and only if the languages $L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{IN}}$ are* order-insensitive. *A language L is* order-insensitive *if for every pair of*

	Deny-override	Only-one-applicable	Weak consensus	Strong majority
P	$\#\mathsf{P} > 0 \wedge \#\mathsf{D} = 0 \wedge \#\mathsf{IN} = 0$	$\#\mathsf{P} = 1 \wedge \#\mathsf{D} = 0 \wedge \#\mathsf{IN} = 0$	$\#\mathsf{P} > 0 \wedge \#\mathsf{D} = 0 \wedge \#\mathsf{IN} = 0$	$\#\mathsf{P} > \#\mathsf{D} + \#\mathsf{NA} + \#\mathsf{IN}$
D	$\#\mathsf{D} > 0$	$\#\mathsf{P} = 0 \wedge \#\mathsf{D} = 1 \wedge \#\mathsf{IN} = 0$	$\#\mathsf{P} = 0 \wedge \mathsf{D} > 0 \wedge \#\mathsf{IN} = 0$	$\#\mathsf{D} > \#\mathsf{P} + \#\mathsf{NA} + \#\mathsf{IN}$
IN	$\#\mathsf{D} = 0 \wedge \#\mathsf{IN} > 0$	$\#\mathsf{P} > 1 \vee \#\mathsf{D} > 1 \vee \mathsf{IN} > 0$	$(\#\mathsf{P} > 0 \wedge \#\mathsf{D} > 0) \vee \#\mathsf{IN} > 0$	$(\#\mathsf{P} \leq \#\mathsf{D} + \#\mathsf{NA} + \#\mathsf{IN}) \wedge$ $(\#\mathsf{D} \leq \#\mathsf{P} + \#\mathsf{NA} + \#\mathsf{IN}) \wedge$ $(\#\mathsf{P} + \#\mathsf{D} + \#\mathsf{IN} > 0)$

Figure 2: Using linear constraints to specify PCAs

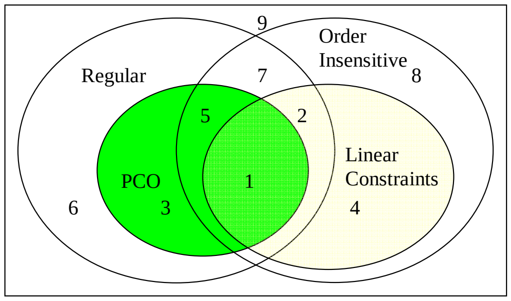


Figure 3: Expressive power of PCL using PCO or linear constraints

strings π, π' such that π and π' consists of the same multiset of letters, $\pi \in L$ if and only if $\pi' \in L$. A PCA that is not order-insensitive is order-sensitive.

The following lemmas establish upper-bounds on the kinds of PCAs that can be expressed using the PCO-based approach and the linear-constraint-based approach in PCL.

LEMMA 1. *A PCO-based PCA is regular.*

PROOF. All PCO-based PCAs specified in PCL are expressed using DFA, and the languages derived from DFAs are regular. □

LEMMA 2. *A PCA specified by linear constraints is order-insensitive.*

PROOF. We cannot use linear constraints to specify order-sensitive PCAs, because the evaluation of linear constraints only depends on aggregated values (i.e. counts) of inputs. The order of inputs is not taken into account in constraint evaluation. □

The upper-bounds given in Lemmas 1 and 2 are not tight. There exist regular PCAs that cannot be expressed using the PCO-based approach. Similarly, there exist order-insensitive PCAs that cannot be expressed in PCL using linear constraints.

Figure 3 presents a diagram that illustrates the expressive power of the two approaches in PCL. Figure 3 has 9 numbered areas. Areas 1-5 can be expressed in PCL; areas 6-9 cannot be expressed in PCL. Some observations from the diagram are:

- PCL can express non-regular PCAs (area number 4 in Figure 3). Examples include weak majority and strong majority.
- There exist order-insensitive PCAs that can be expressed using PCOs but cannot be expressed using linear constraints (area number 5). Examples include those that use periodicity constraints, e.g., permit a request if the number of sub-policies permitting the request is even.

Below we discuss areas 1-9 in Figure 3 and give examples in each area.

Area 1: These PCAs can be expressed using PCO or linear constraints. Examples include *Deny-overrides*, *Permit-overrides*, *Only-one-applicable*, *Weak-consensus*, *Strong-consensus*.

Area 2: These PCAs can be expressed using linear constraints, are regular, but cannot be expressed using PCO. An example is *At least k*, which permits a request if at least k sub-policies permit it, and denies a request if at least k deny. The PCA returns IN if both of the above two conditions hold, and returns NA if neither holds.

Specifying linear constraints to define this PCA is straightforward. The PCA is regular for any fixed k, as one can use a DFA which counts how many P's and D's have been encountered so far. We point out that "at least 1" is the same as weak-consensus, which is expressible using a PCO. However, "at least 2" cannot be expressed using PCO, because it requires differentiating between eight states

$$\{\langle i \times \mathsf{P}, j \times \mathsf{D}\rangle \mid i, j \in \{0, 1, 2+\}\},$$

whereas a PCO can distinguish only four of them. Only the four states in Σ can be used to differentiate states.

Area 3: These PCAs can be expressed using PCO, but are order-sensitive, and hence cannot be expressed using linear constraints. An example is *First-applicable*.

Area 4: These PCAs can be expressed using linear constraints, but are not regular. Examples include *weak-majority*, *strong-majority*, *super-majority-permit* (see Section 1). All these require counting to an unbounded number, and hence are not regular.

Area 5: These PCAs can be expressed using PCO, are order-insensitive, but cannot be expressed using linear constraints. An example is *odd-permit*, which permits a request if there are an odd number of sub-policies that permit it, and return NA otherwise.

Area 6: These PCAs are regular, order-sensitive, and cannot be expressed using PCOs. An example is *first-two-agree*, which permits (resp. denies) a request if during the combining process, it encounters two applicable sub-policies that permit (resp. deny) the request before encountering two applicable sub-policies that deny (resp. permit) the request. This PCA is clearly regular. It cannot be expressed using PCO, because, similar to "at least 2", it requires differentiating between eight states, whereas a PCO can distinguish only four of them.

Area 7: These PCAs are both regular and order-insensitive, but cannot be expressed in PCL. These PCAs typically involve periodicity constraints, such as permitting a request when the number of permit is even and the number of deny is odd.

Area 8: These PCAs are order-insensitive, non-regular, and cannot be expressed using linear constraints. One example is to permit a request when number of permits is even and is greater than the number of denies.

Area 9: These PCAs are non-regular and order-sensitive. An example is *ordered evaluation dominance*, which permits a request if during the combining process, the number of sub-policies that permit the request is always larger than the number of sub-policies that deny the request. This PCA is neither regular (as it requires counting) nor order-insensitive.

Expressing Other Policy Combining Behaviors. To achieve more exotic behaviors, e.g., combining the first two sub-policies using permit-overrides, and then combine with the third using deny-overrides, one can exploit the recursive nature of XACML and uses a policy-set that contains other policy-sets as sub-policies.

4. HANDLING OBLIGATIONS

In this section, we discuss how to combine the obligations associated with sub-policies in PCL. In PCL we adopt an approach that views obligations as black-boxes, that is, our approach does not rely on any ways of comparing whether one set of obligations implies another set. Conceptually, a PCA that combines the decisions and obligations of n sub-policies $P_1, P_2, \cdots, P_n$ on a request gives $\langle d, O\rangle$, where $d \in \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$ denotes the decision, and O indicates which sub-policies' obligations should be returned. O can be a set of indexes to the sub-policies, i.e, $O \subseteq \{1, 2, \cdots, n\}$, which means that obligations from sub-policies $\{P_i \mid i \in O\}$ are returned. Note that when the obligation of a policy P_i is returned, only those obligations whose `FullFillOn` attributes are the same as the decision of P_i (i.e., both are permit or both are deny) are returned. O can also take one of two special values $\emptyset$ and μ. $O = \emptyset$ means that no obligation is returned, and $O = \mu$ means that the set of sub-policies whose obligations should be returned is uncertain. Following the approach in XACML, we assume that in any combined outcome $\langle d, O\rangle$, we have "$(d = \mathsf{NA}) \Rightarrow (O = \emptyset)$" and "$(d{=}\mathsf{IN}) \Rightarrow (O{=}\mu)$".

We first describe the obligation combining scheme for the PCO-based approach, and then we discuss the case for linear constraints.

4.1 Combining Obligations with PCO

The need to combine obligations places restrictions on what matrices can be used as PCOs. Because XACML's obligation behavior implicitly assumes that a Permit outcome depends upon evaluated sub-policies that also permit the request, we require a PCO to satisfy the following principle:

PRINCIPLE 3. *A PCO matrix must satisfy: a* P *can occur only in the row or the column corresponding to* P*, and a* D *can occur only in the row or the column corresponding to* D.

We note that the only PCA we have encountered so far that does not satisfy this is XACML's deny-overrides PCA, which we have shown to be problematic in several ways. The variant that we show in Figure 1(a) satisfies the above principle.

In PCO-based specification of PCA, we need to specify when combining results from two branches, which branch's obligations should be used. To simplify the specification, we adopt the following default behavior: (1) If the combined outcome is NA, then $O = \emptyset$; (2) If the combined outcome either is IN or is an uncertain value (i.e., a non-singleton set), then $O = \mu$; (3) If the combined outcome is P, and only one branch gives P, then that branch's obligation should be returned; (4) If the combined outcome is D, and only one branch gives D, then that branch's obligation should be returned.

What remain to be specified are what happens when a P is resulted from two branches that return P (and similarly, when a D is resulted from two branches that return D). This can be specified in PCL; and PCL allows three possibilities:

1. `both`, which indicates that the combined outcome depends on both branches. This would be desirable when specifying the strong-consensus PCA, which permits a request when all sub-policies permit the request. Since all sub-policies play a role in the decision; their obligations should all be returned.
2. `first`, which indicates that the combined outcome depends on the first branch. This would be desirable when specifying the first-applicable PCA. Combining a P and the next P gives P, but only the obligations of the first branch should be used.
3. `either`, which indicates that the combined outcome depends on one of the two branches, and either branch is fine. This would be desirable when specifying the permit-overrides PCA. Combining two P's give a P, and any one of the branch is sufficient for permitting the request. Returning the obligations of either branch should be considered correct.

In PCL, a PCO-based PCA can specify which of the three approaches is used when combining two P's and which is used when combining two D's. (Refer to Figure 5 for the complete syntax.)

Figure 4 gives the pseudo-code for combining decisions and obligations. This code returns sets of obligations, rather than sets of policy indices.

Input: A sequence of policies $P_1, \ldots, P_n$, a request Q, a PCO g, $v_Y \in \{\texttt{both}, \texttt{first}, \texttt{either}\}$ indicates how to combine obligations for P and P, $v_N \in \{\texttt{both}, \texttt{first}, \texttt{either}\}$ indicates how to combine obligations for D and D.

Output: $\langle d, O\rangle$, where $d \in \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{IN}\}$ is the decision and O is a set of obligations

```
$d := \emptyset$; $O := \emptyset$; // initialization
For $i := 1$ TO $n$ Do
    $\langle d_i, o_i\rangle := D(P_i, Q)$;
    $d' := g(d, d_i)$;
    If $d' = \mathsf{NA}$ Then $O := \emptyset$;
    Else If $d' = \mathsf{IN}$ or $d'$ is a non-singleton Then $O := \mu$;
    Else If $d' \neq d$ and $d' = d_i$ Then $O := o_i$;
    Else If $d' = d$ and $d' = d_i$ Then
        If $d = \mathsf{P}$ Then $v := v_Y$ Else $v := v_N$;
        Switch $v$
            Case both: If $O \neq \mu$ and $o_i \neq \mu$ Then
                            $O := O \cup o_i$;
                       Else $O := \mu$;
            Case first: continue;
            Case either:
                       If $O = \mu$ and $o_i \neq \mu$ Then $O := o_i$;
        EndSwitch;
    EndElse;
    $d := d'$;
EndFor
Return $\langle d, O\rangle$
```

Figure 4: Description of the obligation-combining algorithm for the PCO-based approach. $D(P_i, Q)$ recursively calls the same procedure to return combined decision and obligations for P_i.

4.2 Obligations with Linear Constraints

When a PCA is specified using linear constraints, one can choose either of the following options for obligation handling.

- `all`: All sub-policies contained in the current policy must be evaluated, and the obligations of all the sub-policies whose decision is the same as the final decision of the PCA (when the final decision is P or D) are returned.
- `all-evaluated`: There is no requirement to evaluate all sub-policies, as long as the final decision can be determined

without evaluating all of them. The obligations of all sub-policies that have been evaluated and whose decision is the same as the final outcome of the PCA are returned.

5. DISCUSSIONS

Implementation. We implemented PCL and integrated it with Sun's implementation for XACML 1.1[19]. To make the integration possible, we changed several classes. In particular, we changed the Result class in order to account for errors in evaluation. Thus instead of holding only one decision element, it was extended to include a set of decisions that conform to PCL. The encoding of the response was also modified to include a set of decisions in the ⟨Decision⟩ element. Subsequently, the relevant files using this class were modified to account for changes made to the class interface.

Impacting XACML. While PCL provides a general approach for specifying PCAs, and is useful for any access control policy language that desires sophisticated policy combining behavior, the place that PCL is mostly likely to have an immediate impact is XACML. We have developed an XML encoding of PCL, which is described in Figure 5. XACML has been increasingly adopted in recent years, and is becoming the de facto standard for specifying access control policies for various applications, especially web services. Currently, version 3.0 of XACML is being developed. This new version of XACML will be addressing some of the issues with the `Indeterminate` combining result which we discussed in this paper. Furthermore, the new version will introduce two new combining algorithms: permit-unless-deny and deny-unless-permit, both of which can be expressed using our approach. We plan to submit our proposal of adding PCL to the XACML standard group.

6. RELATED WORK

Many policy languages specify some fixed policy combining strategies, such as XACML [22], XACL [10], EPAL [1], SPL [17], and firewall policies. Policy combing also appears in the access control language in Bauer et al. [4]. However, none of them provides a formal language for specifying new PCAs.

Theoretical studies have resulted in a number of algebras for combining access control policies [6, 8, 3, 11, 12, 16, 21]. Bonatti et al. [6] proposed a 2-valued algebra for composing access control policies. Backes et al. [3] introduced a 3-valued algebra for combining enterprise privacy policies. Their algebra applies to EPAL [1] policies, whose effect can be "allow", "deny", or "don't care". They defined three operators: conjunction, disjunction, and scoping, and studied their properties. Among existing algebras, this is the only one that specifies how obligations are combined. Obligations can be union'ed (corresponding to our `both`)or or'ed (corresponding to our `either`) when combining two sub-policies of the same result. Bruns el at [8] introduced a 4-valued algebra for policy combination. They defined a policy as a 4-valued predicate that maps each request to grant, deny, conflict, or unspecified, which correspond to the four elements of the Belnap bilattice [5]. None of these works deal with policy evaluation errors; nor can they specify counting-based policy combining algorithms.

Mazzoleni et al. [16] proposed a policy integration algorithm for XACML. Their approach allows an resource owner to specify how she would like her policies to be combined with third party policies, and the combination behavior can be based on analyzing the two policies to be combined. For example, one can specify that two policies are combined using some strategy only if they are similar according to some measurement of similarity. Such behavior cannot be specified in PCL, as the combining result on one request may be depending on how the policies handle other requests. This work, however, proposes only new combination algorithms, rather than a language for specifying such algorithms.

Kolovski et al. [12] presented a formalization of XACML using description logics. They map the semantics of XACML policies and combining algorithms to a set of logical rules, which allows them to perform a variety of analysis on XACML policies. The formalization of XACML combining algorithms in this work, however, is incomplete. It does not deal with policy evaluation errors, nor can it express the only-one-applicable PCA. The formalization in [12] was not intended to be a language for specifying PCAs.

Wijesekera and Jajodia [21] proposed a propositional algebra to manipulate access control policies. An interesting aspect about this work is that a policy can be a non-deterministic set of permission assignments, e.g., a policy may grant either permission A or permission B but not both. They also use algebraic approach for policy combining, and hence has the same limitations as other algebraic approaches.

Our work uses automata for policy evaluation. This is related to work on security automata [18, 13], started by Schneider [18]. We are using similar techniques (automata, in particular) to solve different problems in access control.

Other related and orthogonal work includes work on XACML policy analysis and testing [15, 9] and optimizing XACML policy evaluation [14].

7. CONCLUSION

In this paper, we have introduced an expressive policy combining language PCL. PCL advances the theory of policy combining in several ways. It is more expressive than previous algebras. It systematically treats policy evaluation errors as uncertainty over a set of possible values. It also considers combining obligations as well as decisions. Though PCL is primarily motivated by XACML, the underlying formalism and approaches can be applied to other languages. On the practice side, we have provided a detailed and principled analysis of policy combining in XACML and identified several problems in XACML. We have also shown that PCL can be implemented and integrated with XACML. One big advantage of having such a formal language in XACML is that it enables the introduction and usage of new PCAs. Anyone can create a new PCA, and all PCL-enabled policy evaluation engines will be able to evaluate policies using the PCA.

8. REFERENCES

[1] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. The enterprise privacy authorization language (EPAL). *http://www.w3.org/2003/p3p-ws/pp/ibm3.html*.

[2] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pp. 195–210, 2002.

[3] M. Backes, M. Durmuth, and R. Steinwandt. An algebra for composing enterprise privacy policies. In *ESORICS '04: Proceedings of the 2004 European Symposium on Research in Computer Security*, 2004.

[4] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: ACM Conference on Programming Language Design and Implementation*, 2005.

[5] N. D. Belnap. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, 1977.

⟨PolicyCombiningAlgorithm⟩	::=	⟨PreProcessing⟩ ⟨SpecType⟩ ⟨PostProcessing⟩	(1)
⟨SpecType⟩	::=	(⟨ConstraintSet⟩ | ⟨DecisionMatrix⟩)	(2)
⟨PreProcessing⟩	::=	"PreProcessing" "(" [*True* | *False*] ")"	(3)
⟨DecisionMatrix⟩	::=	"Matrix" "(" [⟨PermitResult⟩] [⟨DenyResult⟩] [⟨NotApplicableResult⟩] [⟨ConflictResult⟩] ⟨YY-Obligation⟩ ⟨NN-Obligation⟩ ")"	(4)
⟨PermitResult⟩	::=	"Permit" "(" ⟨DecisionPair⟩$^+$ ")"	(5)
⟨DenyResult⟩	::=	"Deny" "(" ⟨DecisionPair⟩$^+$ ")"	(6)
⟨NotApplicableResult⟩	::=	"NotApplicable" "(" ⟨DecisionPair⟩$^+$ ")"	(7)
⟨ConflictResult⟩	::=	"Conflict" "(" ⟨DecisionPair⟩$^+$ ")"	(8)
⟨DecisionPair⟩	::=	⟨tag1⟩ - ⟨tag2⟩	(9)
⟨YY-Obligation⟩	::=	"YY-Oblg" "(" ⟨obl-tag1⟩")"	(10)
⟨NN-Obligation⟩	::=	"NN-Oblg" "(" ⟨obl-tag1⟩")"	(11)
⟨ConstraintSet⟩	::=	"Constraints" "(" [⟨Permit-If⟩] [⟨Deny-If⟩] [⟨Conflict-If⟩] [⟨NotApplicable-If⟩] ⟨ConsObligation⟩ ")"	(12)
⟨Permit-If⟩	::=	"Permit" "(" ⟨ConstraintBiExp⟩$^+$ ")"	(13)
⟨Deny-If⟩	::=	"Deny" "(" ⟨ConstraintBiExp⟩$^+$ ")"	(14)
⟨Conflict-If⟩	::=	"Conflict" "(" ⟨ConstraintBiExp⟩$^+$ ")"	(15)
⟨NotApplicable-If⟩	::=	"NotApplicable" "(" ⟨ConstraintBiExp⟩$^+$ ")"	(16)
⟨ConstraintBiExp⟩	::=	⟨ConstraintTerm⟩ {⟨simpleOp⟩ | ⟨logicOp⟩} ⟨ConstraintTerm⟩	(17)
⟨ConstraintTerm⟩	::=	(⟨ConstraintTerm⟩ | ⟨C-Term⟩ | ⟨Integer⟩)	(18)
⟨ConsObligation⟩	::=	"Cons-Oblg" "(" ⟨obl-tag2⟩")"	(19)
⟨C-Term⟩	::=	(*#P* | *#D* | *#N* | *#C*)	(20)
⟨simpleOp⟩	::=	$(+ \mid - \mid \times \mid > \mid < \mid \geq \mid \leq)$	(21)
⟨logicOp⟩	::=	(*AND* | *OR*)	(22)
⟨Integer⟩	::=	⟨digit⟩$^+$	(23)
⟨PostProcessing⟩	::=	"PostProcessing" "(" [*True* | *False*] ")"	(24)
⟨tag1⟩	::=	(*Permit* | *Deny* | *NotApplicable* | *Conflict* | *Any*)	(25)
⟨tag2⟩	::=	(*Permit* | *Deny* | *NotApplicable* | *Conflict*)	(26)
⟨obl-tag1⟩	::=	(*All* | *First* | *Either*)	(27)
⟨obl-tag2⟩	::=	(*All* | *AllEvaluated*)	(28)

Figure 5: Syntax of the proposed PCL schema in BNF. Complex types are shown in ⟨⟩ and simple types are shown in italics. Elements following each other denotes sequence, | denotes choice. #P, #D, #NA and #CF denote the number of P, D, NA, and CF to be combined, respectively. Lines (4)-(11) define the XML encoding of PCOs. The DecisionMatrix element can have the property isSymmetric which denotes whether the matrix is symmertic. In case of symmetry, only half the matrix needs to be encoded. The PolicyCombiningAlgorithm element can have a policyCombiningAlgId attribute in order to uniquely identify the algorithm in an implementation. Lines (12)-(19) define the XML encoding of the linear-constraint-based approach.

[6] P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. and Sys. Sec.*, 5(1):1–35, Feb. 2002.

[7] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *TACAS*, pp. 317–333, 2005.

[8] G. Bruns, D. S. Dantas, and M. Huth. A simple and expressive semantic framework for policy composition in access control. In *FMSE*, pp. 12–21, 2007.

[9] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE*, pp. 196–205, 2005.

[10] S. Hada and M. Kudo. XML access control language: Provisional authorization for XML documents. *http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html*.

[11] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *CSFW*, 2003.

[12] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proc. 16th International Conference on World Wide Web*, pp. 677–686, 2007.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[14] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A fast and scalable XACML policy evaluation engine. In *Proc. SIGMETRICS*, pp. 265–276, June 2008.

[15] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 16th International Conference on World Wide Web*, pp. 667–676, May 2007.

[16] P. Mazzoleni, E. Bertino, B. Crispo, and S. Sivasubramanian. XACML policy integration algorithms: not to be confused with XACML policy combination algorithms! In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pp. 219–227, New York, NY, USA, 2006. ACM.

[17] C. Ribeiro, A. ZÍšquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *NDSS '01: Network and Distributed System Security Symposium*, 2001.

[18] F. B. Schneider. Enforceable security policies. *ACM Tran. Inf. and Sys. Sec.*, 3(1):30–50, 2000.

[19] Sun Microsystems. Sun's XACML implementation. *http://sunxacml.sourceforge.net/*.

[20] D. Wijesekera and S. Jajodia. Policy algebras for access control the predicate case. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pp. 171–180, 2002.

[21] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Tran. Inf. and Sys. Sec.*, 6(2):286–325, May 2003.

[22] XACML TC. OASIS eXtensible Access Control Markup Language (XACML). *http://www.oasis-open.org/committees/xacml/*.