Towards Composing Access Control Policies

Muhammad Shahzad
Department of Computer Science
North Carolina State University
Raleigh, NC 27606
Email: mshahza@ncsu.edu

Abstract—Existing access control languages assume a single monolithic specification of the entire access control policy. This assumption does not fit many real-world situations, where access control might need to combine independently stated restrictions in separate policies that should be enforced as one. Furthermore, due to the rapidly increasing number, size, and complexity of the access control policies in the enterprise and IoT communication networks, efficient evaluation of access requests against large number of policies is challenging. Thus, there is an imminent need to develop a framework that composes distributed access control policies into a single policy. In this paper, we propose ACE, an access control policy composition engine, which uses a novel algebraic framework to combine any pair of access control policies into a single resultant policy. Through repetitive application, ACE can generate a resultant policy from any arbitrary number of individual policies. ACE can perform three binary composition operations on access control policies, namely addition, conjunction, and subtraction. The decision to any arbitrary access control request by the composed policy is the same as the decision for that request obtained by combining the independently obtained decisions from all policies. We implemented ACE, integrated it with SUN’s implementation of XACML (a well-known access control language), and extensively evaluated it on a large number of policies. Our results show that the decisions obtained using the policy composed with ACE match the decision obtained from SUN’s implementation for 100% of the access requests. Our results also show that with ACE, the evaluation time of access requests reduces by at least an order of magnitude compared to when ACE is not used.

I. INTRODUCTION

Researchers have proposed several approaches to increase the expressiveness and flexibility of access control languages by supporting multiple policies within a single framework [3], [7]. Although all these approaches are based on powerful languages able to express different policies, they still assume a single monolithic specification of the entire policy. This assumption does not fit many real-world situations, where access control might need to combine independently stated restrictions that should be enforced as one. An example of such a real-world scenario is the smart city IoT infrastructure, where the infrastructure administrators and the infrastructure users impose their own independently formulated access restrictions on the data collected by the IoT infrastructure. In such a scenario, the infrastructure access control policy has to combine the access control policies from each of the administrators and users. Another example is represented by “dynamic coalition” scenarios where different parties, coming together for a common goal for a limited time, need to merge their access control policies for shared resources, such as the wireless network, in a controlled way while retaining their autonomy. It may also be desirable to make an aggregate access control policy by combining several small independently conceived policies. This situation calls for a policy composition framework, using which, different component policies can be integrated into a single policy while retaining their independence.

Another motivation to develop a policy composition framework is the rapidly increasing number, size, and complexity of the access control policies due to the explosive growth of resources and applications deployed in the enterprise and IoT communication networks. Typically, the access control policies are constructed independently by different entities in both enterprise and IoT communication networks. Efficient evaluation of access requests against even a single access control policy is already challenging [9]. Evaluation against multiple policies only exacerbates this challenge because a policy evaluation engine needs to retrieve all the access control policies, evaluate a request against each policy, and combine all decisions into a final decision based on some predefined mechanism for resolving the decision conflict. Therefore, it is very important to develop a framework that composes distributed access control policies into a single policy such that the policy evaluation engine only needs to evaluate a request against that single policy, which is much more efficient than evaluating a request against all distributed policies.

In this paper, we propose ACE, an access control policy composition engine, which uses a novel algebraic framework to combine any two arbitrary access control policies into a single aggregate access control policy. More specifically, given two access control policy specifications \(A\) and \(B\), ACE generates an aggregate composed policy specification \(R\) such that the access control decisions resulting from \(R\) are the same as the combined decisions resulting from using \(A\) and \(B\) separately. Through repetitive application, ACE can generate an aggregate access control policy from any arbitrary number of individual access control policies. It can perform three composition operations on access control policies, namely addition, conjunction, and subtraction. These are binary operations that take two policies and perform the desired composition operation to generate a resultant policy.

The objective of composing multiple component policies into a single aggregate composed policy is very challenging due to two main reasons. First, the aggregate composed policy may lead to unforeseen decisions to certain access requests due to interference between the component policies. Second,
the composed policy may lead to the loss of control on the individual component policies and their autonomous maintainability. In our design of ACE, we have addressed both these challenges. Our approach ensures that in combining any pair of policies, all theoretically possible outcomes are identified and accounted for. It also allows us to update composed policies if the component policy from which the composed policy has been derived is changed.

In its current implementation, ACE composes policies specified in XACML (eXtensible Access Control Markup Language) [6]. XACML is a rich language proposed by OASIS [1] that can be used to implement access control policies for various applications, such as networking [5], web services [10], smart homes [8], and several more. In XACML, whenever a subject has to access a resource (e.g., a user application probing an IoT sensor), it sends the request to a policy enforcement point (PEP). PEP manages the access to the protected resources. PEP forwards this request to a policy decision point (PDP) to find out whether or not the subject has the privilege to access the resource. PDP checks the request against the XACML policy and determines whether the request should be permitted or denied. PDP sends its decision back to PEP which enforces it. Our ACE resides in PDP, where it takes all individually specified XACML policies as input and generates a single XACML policy, which the PDP checks the requests against. While the current implementation of ACE works with XACML, the algebraic framework that ACE employs uses generic primitives that can be seamlessly extended to most other access control languages.

II. RELATED WORK

We first describe prior work on optimizing PDP of XACML. After that, we introduce some existing algebras proposed for access control. While the functionalities provided by these algebras are relevant to XACML policy compression, these all come with a set of fundamental limitations which make them impossible to be used for policy composition.

A. PDP Optimization

Sun provides an open-source implementation of XACML PDP [2]. This implementation performs a brute force search by comparing any given access request against all the rules specified in an XACML policy set. This, however, is not an efficient approach, and causes severe bottlenecks in large-scale systems at runtime. Liu et al. proposed XEngine, which introduces a new representative form of access control policies that lets PDP make the decisions only based on the first applicable rule, instead of a brute force search on the entire set of policies [9]. To achieve this, XEngine converts the values in string format in XACML access rules to a numerical format using an operation called policy numericalization. The numericalized XACML policy has a hierarchical structure with multiple matching rules. Next, it applies another operation, called policy normalization, on this numericalized XACML policy to convert it into a flat numerical structure with only a single matching rule. Last, it converts the numericalized and normalized policy to a tree structure, which is used to efficiently process access requests in PDP. While XEngine significantly improves the performance of PDP, it still cannot compose multiple policies into a single one.

B. Policy Algebras for Access Control

Ni et al. introduced an algebra, namely “\(\mathcal{D}\)-algebra”, where \(\mathcal{D}\) here stands for “decision”. \(\mathcal{D}\)-algebra is functionally complete, i.e., any possible decision matrix can be represented in this algebra. The primary design objective of \(\mathcal{D}\)-algebra is to avoid unintended results in standard policy algorithms that are due to the lack of formal semantics in the decision model. The powerset interpretation of \(\mathcal{D}\)-algebra highlights the existing drawbacks in XACML rule/policy evaluation truth tables and policy combining algorithms, and at the same time suggests a set of solutions to overcome these problems. Unfortunately, this algebra is not directly usable for our problem because it cannot perform operations such as addition and subtraction on individual policies. Bonatti et al. proposed another algebra of security policies along with the associated formal semantics [4]. The framework therein formulates complex policies as expressions of the algebra and is flexible enough to keep the composition process by organizing compound specifications into different levels of abstraction. More specifically, the authors analyze the problem of composing security policies in a modular and incremental fashion for a policy composition framework. Furthermore, they proposed an algebra of security policies as a composition language.

III. XACML OVERVIEW

In this section, we present a brief overview of XACML. The detailed description of XACML can be found in [6]. The fundamental entity in an XACML policy is a rule. A rule is made up of a target, an effect and optionally, a condition. The target is a predicate over subject(s) (such as an IoT infrastructure user), resource(s) (such as sensor data), and action(s) (such as delete sensor data) of the access requests. The effect of a rule is the decision made by that rule, which can either be permit or deny. The condition is used to further refine the applicability of the rule beyond the predicate specified by its target. The effect of a rule is returned in response to a request if and only if the request matches both the target and the condition of that rule.

On top of a set of rules exists a policy. A policy consists of a target, a set of rules, and a rule combining algorithm. PDP checks a request against the rules of a policy only if the rule satisfies the target of that policy. On top of a set of policies exists a policy-set. A policy-set consists of a sequence of policies or policy-sets, a policy combining algorithm, and a target. PDP matches the targets of a policy and a policy-set in the same way as it matches the target of a rule.

The rule/policy combining algorithms resolve an access decision in the case of a conflict or redundancy within a policy or policy set. XACML supports four combining algorithms: (1) first-applicable, (2) only-one-applicable,
(3) permit-overrides, and (4) deny-overrides. If using first-applicable, PDP returns the effect of the first rule (policy) that matches the access request. If using only-one-applicable, PDP returns the effect of the only applicable rule (policy). It returns indeterminate if more than one rule (policy) match a request. If using permit-overrides, PDP returns permit if at least one rule (policy) that matches the request has the effect of permit. If using deny-overrides, PDP returns deny if at least one rule (policy) that matches the request has the effect of deny. If a request does not match against any rule (policy), then PDP returns not-applicable.

IV. ACCESS CONTROL POLICY COMPOSITION ENGINE

Next, we describe how ACE composes any two independent policies into a single policy. We use the term compose and not combine because in addition to combining any two given policies, ACE can also subtract the permissions of one policy from the other. Specifically, we present three fundamental policy composition operations: (1) addition, (2) conjunction, and (3) subtraction. Next, we first give an overview of ACE, followed by the description of each step that ACE performs.

A. ACE Overview

To compose any two given policies, ACE first needs to identify the decision of both policies for any valid request without actually having to enumerate all possible valid requests. To do so, it performs four steps. First, it represents each policy through a tree-like structure called policy decision diagrams (PDDs). Second, it makes the tree structure of the two policies identical by applying a process that we call PDD shaping. Third, it applies the required composition operation (i.e., addition, conjunction, or subtraction) on the corresponding leaves of the two identical trees, and obtains a resultant PDD. Last, it regenerates rules from the resultant PDD according to the semantics of the language being used (in our case XACML). The policy evaluation engine uses the rules from this single resultant policy and arrives at the same decisions for any given request that it would arrive at had it used the two policies separately and then combined their results. Note that the four steps that ACE uses to compose the policies are generic and not tied to any particular access control language. Figure 1 shows the block diagram showing the four steps involved in composing any two given policies into a single resultant policy. Next, we describe these four steps.

B. PDD Conversion

To convert an XACML policy to a PDD, we first numericalize all attributes in its rules, where we map each attribute to a distinct consecutive integer starting from 0. After numericalization, each rule can be represented as a range of integers. This step is followed by the conversion of these numericalized rules into a PDD. The basic idea of the PDD is to generate a directed tree such that any two overlapping rules in the given policy are split into more than two non-overlapping rules represented by distinct directed paths from root to leaf. Each directed path from the root to the leaf represents a distinct non-overlapping rule. The leaf contains the effect of the rule and can either be permit, deny, or not-applicable. Two rules are considered overlapping if there is at least one attribute common in the subject, resource, and/or action parts of the targets of the rules. Formally, a PDD for any given XACML policy with \(d\) attributes \(A_1, A_2, \ldots, A_d\) has the following 5 properties:

1) There is exactly one vertex with no incoming edges. It is called the root. The vertices with no outgoing edges are called terminal nodes.

2) Each vertex \(v\) has a label \(L(v)\). If \(v\) is a nonterminal node, then \(L(v) \in \{A_1, A_2, \cdots, A_d\}\). If \(v\) is a terminal node, then \(L(v) \in \{\text{permit}, \text{deny}, \text{not-applicable}\}\).

3) Each edge \(e: u \rightarrow v\) is labeled with a nonempty set of integers, denoted \(I(e)\), where \(I(e)\) is a subset of the domain of \(u\)'s label (i.e., \(I(e) \subseteq D(A(u))\)).

4) A directed path from the root to a terminal node is called a decision path. No two nodes on a decision path have the same label.

5) The set of all outgoing edges of a node \(v\), denoted \(E(v)\), satisfies the following two conditions: (a) consistency: \(I(e) \cap I(e') = \emptyset\) for any two distinct edges \(e\) and \(e'\) in \(E(v)\); (b) completeness: \(\bigcup_{e \in E(v)} I(e) = D(A(v))\).

Figures 2(a) and 2(b) show two example PDDs. \(S\) represents the subject, \(R\) represents the resource, and \(A\) represents the action. Note from the figures that the range for subject is from 0 to 3, which means that in the original XACML policy, subject takes on 4 distinct string values. Similarly, there are two distinct values each for resource and action. Each terminal node gives the decision of the rule represented by the directed path from root to that terminal node.

Fig. 1: Architecture of our access control policy composition engine

Fig. 2: Examples of policy decision diagrams
C. PDD Shaping

Next, we shape these PDDs such that both PDDs become semi-isomorphic and are functionally equivalent to their respective original PDDs. These semi-isomorphic PDDs, represented as SPDDs, have the following three properties: (1) if the labels are ignored, two SPDDs are identical in structure, (2) the labels of the root nodes, all the internal nodes, and the directed arcs are the same across the two SPDDs, and (3) the labels of the corresponding terminal nodes across the two SPDDs can be different. To convert any pair of PDDs into two isomorphic PDDs, ACE applies the following three operations:

1) Node Insertion: If along all the decision paths containing a node $v$, there is no node that is labeled with the field $A_i$, then insert a node $v'$ labeled $A_i$ above $v$ and make all the incoming edges of $v$ point to $v'$, add an arc from $v'$ to $v$, and label this arc with the domain of $A_i$.

2) Arc Splitting: For an arc $e$ from $v_1$ to $v_2$, if $I(e) = S_1 \cup S_2$, where neither $S_1$ nor $S_2$ is empty, split $e$ into two arcs by replacing $e$ by two edges from $v_1$ to $v_2$, and labeling one arc with $S_1$ and the other with $S_2$.

3) Subgraph Replication: If a node $v$ has $m$ ($m \geq 2$) incoming edges, make $m$ copies of the subgraph rooted at $v$ and make each incoming edge of $v$ point to the root of one distinct copy.

Figures 3(a) and 3(b) show the SPDDs obtained after the shaping of the PDDs shown in Figure 2. We can see that the structures of both SPDDs are exactly the same. The only difference lies at the decision level in the greyed terminal nodes.

D. Policy Composition

Next, we describe how we apply the three fundamental composition operations of addition, conjunction, and subtraction on the SPDDs to obtain the resultant PDD.

1) Addition (+): Addition operation is useful for any scenario where access requests can be authorized if allowed by any of the two policies. Consider an example where all the engineering departments share a resource such as the financial documents of the engineering division of the university. An access to this kind of resource may be granted only if all the authorities that have a stake in such a resource agree on it. This means that in conjunction operation, $dec^{i_1}$ is equal to 1 if and only if both $dec^{i_1}$ and $dec^{i_2}$ are equal to 1. While addition enforces maximum privilege, conjunction enforces minimum privilege, i.e., for a given decision path in the SPDDs, the decision of the resultant policy for that decision path can be permitted if and only if none of the decisions in the two SPDDs on that path are deny. The decision of a terminal node $i$ of PDD $R$ is given by equation 2.

\[ dec^{i_R} = dec^{i_1} \land dec^{i_2} \]  

(2)

where $\land$ represents standard AND operation. As discussed in the previous section, $dec^{i_1}$ can also be equal to $n$, so we need to define $\land$ operation for the case when either one or both of the operands are $n$. Following the same argument as for $\lor$ operator i.e., if an operand $dec^{i_1}$ is $n$, it means that there is no rule available for any request that matches the decision path
towards the terminal node $i$ in SPDD$_j$ and the decision of the other policy (SPDD$_k$) should be used. Thus, $dec^k_i \land n = n \land dec^k_i = dec^j_i$. Figure 5 shows the resultant PDD after performing the conjunction operation on SPDD$_1$ and SPDD$_2$.

3) Subtraction ($-$): This operation is used in the scenarios when a policy has to be restricted by eliminating all the accesses in a second policy. For example, when we have to remove a graduated student’s access to the wireless network, this subtraction operation is needed. For any given decision path, if the terminal node of SPDD$_1$ is deny, then the corresponding terminal node of the resultant policy has to be deny because our task is to remove any permits of SPDD$_2$ from SPDD$_1$, therefore, we can never convert a deny of SPDD$_1$ to permit. For any given decision path, if the terminal node of SPDD$_2$ is permit, then the corresponding terminal node of the resultant policy is permit if and only if the corresponding terminal node of SPDD$_2$ is not permit. If the corresponding terminal node of SPDD$_2$ is permit, then we have to remove this permission from SPDD$_1$. Therefore, we convert this permit to deny in the resultant policy as per the goal of the subtraction operation. If the terminal node of a decision path in SPDD$_1$ is not-applicable, then the corresponding terminal node of the resultant policy has to be deny if the terminal node of SPDD$_2$ is either permit or deny. Table I shows the required truth table for the subtraction operation.

We implemented ACE using Java and integrated it with Sun’s implementation of XACML PDP [2]. The Java program takes any arbitrary number of policies in the XACML format as input along with a composition expression defining how the results of the policies should be composed. It applies the binary composition operations on the policies according to the composition expression to obtain the final composed policy. An example of a composition expression when there are four independent policies is $(P_1 + (P_2 \cdot P_3)) - P_4$. This composition expression implies that the decisions of policies $P_2$ and $P_3$ should first be composed using Eq. (2). The resultant should next be composed with the decision of policy $P_1$ using Eq. (1). The new resultant should finally be composed with the decision of policy $P_4$ using Eq. (3). Given a set of policies, our java program applies the three steps described in Sections IV-B, IV-C, and IV-D to compose pairs of policies in the sequence specified in the composition expression. The final composed policy is still in the form of a PDD. Therefore, it applies the step described in Section IV-E on this final composed PDD to enumerate all rules, and converts them into XACML format, which the Sun’s PDP uses to evaluate requests. To keep the implementation just as flexible as the algebra of ACE itself is, in composing any pair of policies, our java program generates the PDDs of the two policies independently. This can lead to assignment of different numerical values to subjects, resources, and actions across the two policies during the numericalization of attributes in the step described in Section IV-B. To solve this problem, we need to make the numerical labels of the PDDs consistent. For this, our program performs an additional step of standardization between the two steps described in Sections IV-B and IV-C. During the standardization step, ACE first splits all the edges in PDD of policy B in such a way that each edge has a numerical label with a single value instead of a range. Next, ACE replaces the numerical labels of the attributes in PDD of B with the numerical labels of those attributes in the PDD of A. For any attributes in PDD of B for which there are no numerical labels in the PDD of A, i.e., those attributes never appeared in the policy A, ACE sequentially assigns them numerical labels that are bigger than the maximum value of numerical label in the PDD of A. After this, it sorts all edges according to the values of their numerical labels and adds the not-applicable branches to the PDD. We need to add the not-applicable branches to the PDD to make it complete because the PDD must have a path from node S to the leaf node for any arbitrary request. By this time, the PDD of B becomes significantly large, which can negatively effect the performance in evaluating the access requests. Thus, ACE traverses the PDD of B in reverse breadth first order and combines adjacent branches that have same decisions at nodes on the leaf level. Figure 7 visualizes the standardization step.
VI. Evaluation

We carried out our experiments on a desktop PC running Windows 10 with 16GB memory and Intel i7-6700 processor. We evaluated ACE from two perspectives: correctness and efficiency. For correctness, we compared the decisions for access requests by the standard implementation of Sun’s PDP and by the Sun’s PDP that uses the composed XACML policies generated by ACE. Our results show that the decisions for access requests by both standard PDP implementation and the PDP implementation appended with ACE matched 100% of the times, which demonstrates that the final composed XACML policy generated by ACE is functionally the same as the individual policies evaluated separately first and then their decisions combined. For efficiency, we compared the time it takes for both implementations to evaluate an access request. Our results show that when using the composed policy generated by ACE, the evaluation time of PDP is an order of magnitude smaller than the evaluation time of standard PDP implementation. Furthermore, the difference in the evaluation time of PDP with ACE and PDP without ACE grows almost linearly with the number of policies. We also measured the preprocessing time that ACE takes to compose policies. The preprocessing time includes the time for performing the four steps described in Sections IV-B through IV-E. Our results show that the preprocessing time increases linearly with the number of policies. For 1000 independently generated policies, each with 1000 rules, the processing time is just one minute.

Figure 8 plots the ratio of the access request processing time of standard PDP implementation and the PDP implementation augmented with ACE for different number and sizes of policies. We observe from this figure that the access request processing time of PDP, when augmented with ACE, is an order of magnitude smaller than without ACE. Furthermore, the difference in the processing times increases both with the increase in the number of policies, as well as the number of rules per policy. Each data point in this figure is obtained by averaging the request processing times of 1000 randomly generated requests. The vertical lines on each data point show the standard deviation in the values of the ratios.

Figure 9 plots the average time taken by ACE to generate a composed policy from the given set of policies. The figure also plots standard deviation in the preprocessing times at each data point. We observe from this figure that as the number of policies and the number of rules per policy increase, the preprocessing time increases linearly. However, the preprocessing time is very small. Even for 1000 independently generated policies with 1000 rules each, ACE takes just one minute to generate the composed policy. Note that, in practice, the preprocessing step is required only when component policy changes, which is a very infrequent process. Consequently, the 1 minute time of ACE does not become any bottleneck in implementing new policies in real-world settings.

VII. Conclusion

In this paper, we presented ACE, which can compose any arbitrary number of policies into a single resultant policy, which is functionally equivalent to the individual policies. The key technical depth of this paper is in developing the method to convert policies specified in XACML language into PDDs, shaping them to make them equivalent, applying composition operation on them, and regenerating XACML policy from the resultant PDDs. We implemented ACE, integrated it with SUN’s implementation of XACML, and extensively evaluated it on a large number of policies. Our results show that the decisions obtained using the policy composed with ACE are the same as the decision obtained from SUN’s implementation 100% of the times. Our results also show that with ACE, the evaluation time of access requests reduces by at least an order of magnitude compared to when ACE is not used to compose the policies.

REFERENCES