Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation

Said Marouf Mohamed Shehab
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
{*smarouf, mshehab*}*@uncc.edu*

Anna Squicciarini Smitha Sundareswaran
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA, USA
{*acs20, sus263*}*@psu.edu*

***Abstract*—The adoption of XACML as the standard for specifying access control policies for various applications, especially web services is vastly increasing. A policy evaluation engine can easily become a bottleneck when enforcing large policies. In this paper we propose an adaptive approach for XACML policy optimization. We proposed a clustering technique that categorizes policies and rules within a policy set and policy respectively in respect to target subjects. Furthermore, we propose a usage based framework that computes access request statistics to dynamically optimize the ordering of policies within a policy set and rules within a policy. Reordering is applied to categorized policies and rules from our proposed clustering technique. To evaluate the performance of our framework, we conducted extensive experiments on XACML policies. We evaluated separately the improvement due to categorization and to reordering techniques, in order to assess the policy sets targeted by our techniques. The experimental results show that our approach is orders of magnitude more efficient than the standard Sun PDP.**

***Keywords*-Policy Evaluation; Policy Categorization; XACML;**

I. Introduction

An access control policy is a set of rules that enables resource owners and administrators to control access and dissemination of their shared resources. When a user requests access to a certain resource, the access control module evaluates the policy rules to decide whether to allow or to deny access to the requested resource. With the continuously expanding number of resources and the increasing diversity and size of online systems, policies are becoming more complex and will involve a large number of rules. Efficient policy evaluation techniques are required to ensure that policy evaluation introduces low latency without affecting the correctness of the evaluation process. Taking the widely adopted XACML (Extensible Access Control Mark-up Language) [9] policy as an example, a policy set is composed of a set of policies, where each policy is divided into a set of rules. XACML not only provides a formalism to specify authorization policies, but it also includes information useful in making authorization decisions, as well as approaches to integrate constraints specified by multiple subjects, such as the policy combination algorithm. The policy combination algorithm along with other features unique to XACML, make it a very flexible and rich language.

A policy in XACML is evaluated by an XACML engine, which is composed of two main components, the Policy Evaluation Point (PEP) and the Policy Decision Point (PDP). The PEP receives an access request, translates it into an XACML request, and then sends the XACML to the PDP. The PDP checks the request against a set of XACML policies, and determines whether the request should be permitted or denied. The evaluation process, in turn, has two main phases: first the policy to be used is selected, and second the rules among the selected policies to be evaluated. The designers of current XACML engines, however, have not taken into account performance of the policy evaluation process. For example, Sun XACML PDP [12], which is the first and most widely used evaluation engine, performs brute force searching by comparing a request with all the rules in an XACML policy. Clearly, this approach does not efficiently support a large number of users' requests, who need prompt access to the data they are entitled to. To enable an XACML policy evaluation engine to process a large number of requests in real time, especially in face of a burst volume of requests, an efficient XACML policy evaluation engine is necessary. Our work aims at providing such a policy engine. Achieving this goal is a challenging task due to the complexity of both the XACML policies, and the evaluation process. Policies need to be reorganized according to the incoming access request type, in a possibly inexpensive and adaptive manner. Additionally, in order to preserve the original intention of the policy writers, it is important that the policy reorganization process does not affect the policy evaluation results, that is, the response to access requests must not change.

Starting from the SUN policy evaluation engine [12], in this paper we present the design and implementation of a simple yet effective framework that greatly improves the performance of XACML policies evaluation. Our design draws from the following two observations: (1) users who share common properties have the same requests types, thus, the same subset of rules are evaluated, and (2) optimal rules ordering is subjective to the actual users' requests. Precisely, our problem consists of finding which policy is applicable to an incoming request and also optimizing the ordering of the rules within the policy to match the request. In order to

allow for this type of matching, we propose a technique that utilizes actual users' requests' characteristics. We categorize the users' access requests at two levels. Based on observation (1) we first categorize the request by subjects to see which policy would be applicable to it. Then, based on observation (2) we find a match between the request and the *execution vectors* for that policy. Execution vectors are the order in which the rules in a policy are applicable to a request. We build execution vectors by using different statistics to evaluate the cost of a rule and their frequency, and develop an approach to efficiently reorder policies and rules based on the specific properties of access requests. We formulate the rule optimization problem for access policy requests, and show that our usage framework solves it.

We implemented our proposed framework as an extension of the open source Sun PDP engine. We conducted extensive experiments on synthetic XACML policies of different structures and sizes, and conducted experiments using different sets of access requests. The experimental results show that our framework is orders of magnitude more efficient than Sun's PDP, and the performance difference between our and Sun's PDP grows almost linearly with the number of rules in XACML policies. We tested the categorization and reordering techniques separately, and find interesting results on how our categorization technique by itself already outperforms the Sun implementation by orders of magnitude. The reordering provides a means for adaptability to user requests to further enhance the performance of the policy evaluation subject to different request trends.

The rest of the paper is organized as follows. In the next section we present some background information on XACML and access requests. In Section III we present our usage framework, present the optimal rule ordering problem, and provide an efficient algorithm to reorder rules and policies. In Section IV, we present our categorization based optimization. Our experimental results are shown in Section V, whereas Related work is discussed in Section VI. We conclude the paper with conclusion and pointer for future research directions in Section VII.

II. Preliminaries

In this section we provide the logic formalism adopted throughout the paper to denote XACML policies and access requests. XACML policies are composed of five basic components, namely, *PolicySet*, *Policy*, *Target*, *Rule*, and *Policy and Rule Combining algorithm* for conflict resolution. The root of the XACML policy is the PolicySet element, which is defined as:

Definition 1: PolicySet is a tuple $PS = (id, t, P, PC)$, where:

- id is the PolicySet id.
- t is the PolicySet Target element, and takes values from the set {Applicable, NotApplicable, Indeterminate}.
- $P = \{p_1, \ldots, p_n\}$ is the set of policies.
- PC is the policy combining algorithm.

A *Policy* element is a set of rules and conditions that control access to protected resources which we refer to as objects. A policy contains a *target*, a set of *rules*, and a *rule combining* algorithm. A policy is defined as:

Definition 2: A policy is a tuple $P = (id, t, R, RC)$, where:

- id is the policy id.
- t is the policy target element, and takes values from the set {Applicable, NotApplicable, Indeterminate}.
- $R = \{r_1, \ldots, r_n\}$ is the set of rules.
- RC is the rule combining algorithm.

The Target element t specifies a set of predicates on the request attributes, which must be met in a PolicySet, Policy or Rule to apply to a given request. The attributes in the target element are categorized into *Subject, Resource* and *Action*. The attribute values in a request are compared with those included in the Target, if all the attributes match then the Target's PolicySet, Policy or Rule is said to be Applicable. If the request and the Target attributes do not match then the request is NotApplicable, and if the evaluation results in an error then the request is said to be Indeterminate. If a request satisfies the target of a policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examining its rules. The Target predicates can be quite complex, and can be constructed using functions and attributes. The rule combining algorithm RC allows one to specify the approach to compute the decision result of a policy when the policy contains rules evaluating to conflicting effects. The policy combining algorithm PC follows the same logic but at the PolicySet level.

A *Rule* identifies a complete and atomic authorization constraint that can exist in isolation with respect to the policy in which it has been created. We define rules as follows.

Definition 3: A Rule is a tuple $r = (id, t, e, c)$, where:

- id is the rule id.
- t is the rule target element, and takes values from the set {Applicable, NotApplicable, Indeterminate}.
- e is the rule effect, where $e \in \{Permit, Deny\}$.
- c is a boolean condition against the request attributes.

The rule target element is similar to the policy target instead it indicates the requests applicable to the rule. The condition c is a boolean function with respect to the request attributes. The rule's effect e, which can be Permit or Deny, is returned if the rule's condition c evaluates to true. The rule evaluation can also result in an error (Indeterminate) or the condition is doesn't apply to the request attributes (NotApplicable). Access requests are typically matched against a policy set. A policy set is the root of an XACML policy, it holds policy elements and, possibly, other policy sets. We denote access requests according to the following notation. Let S, O, A and X denote the of all subjects, objects, actions and context variables in an access control system respectively.

Definition 4: (Access Request) An access request q is the tuple (s, o, a, x), where $s \in S$ is the subject making the

request, $o \in O$ is the requested object, $a \in A$ is the requested action on object o, and $x \in X$ are the context attributes.

III. Policy Reordering Framework

When a web server needs to enforce an XACML policy with a large number of rules, the policy evaluation engine may easily become the performance bottleneck for the server. To enable an XACML policy evaluation engine to process simultaneous requests of large quantities in real time, especially in face of a burst volume of requests, an efficient XACML policy evaluation engine is necessary. In such environments the requests' distribution is dynamic in terms of volume, and type of requesters. Motivated by such observation, we develop an adaptive framework that dynamically determines the best ordering according to the incoming requests and the recently received history of requests and executions. In this section we present the basic notions that are relevant for our framework, define statistics extracted from policy execution logs, formulate the rule ordering problem, and finally provide an algorithm to provide the optimal rule ordering.

A. Execution Vector and Policy Permutation

In what follows for the sake of presentation we focus on policy permutation where a similar approach adopted for PolicySet permutation. We define a policy permutation as follows:

Definition 5: (Policy Permutation) Given a policy P with a rule set $P.R = \{r_1, \ldots, r_n\}$, a policy permutation π is a policy P_π generated by the following procedure: (0) $P_\pi.R = \{\}$, $P_\pi.id = P.id$, $P_\pi.t = P.t$, and $P_\pi.RC = P.RC$. (1) P' is a copy of P. (2) Select a random rule r_i from P' and append r_i to the end of P_π. (3) Repeat step 2 until P' is empty.

Policy permutation may alter the correctness of a policy, and result in different evaluations for a same set of requests. We are interested in policy permutations that do not alter the policy evaluation results for any request.

Definition 6: (Safe Policy Permutation) A safe policy permutation π of a policy P is safe iff all requests permitted (denied) by the permuted policy P_π are also permitted (denied) by P.

We assume all requests are well formed such that the policy evaluation returns PERMIT or DENY by the PDP. With such an assumption we provide the below theorem:

Theorem 1: Safe Permit (Deny) Overrides Permutation. A policy P having a rule combining algorithm $P.RC$ set to Permit-Overrides or Deny-Overrides is safe with respect to all possible policy permutations.

Proof: Assuming each rule returns either permit or deny then the policy evaluation of a policy P, with a permit overrides rule combing algorithm is the disjunction of all the rule results represented by: $E(P) = E(r_1) \vee \cdots \vee E(r_n)$. The disjunction operator is commutative where $a \vee b = b \vee a$, and associative where $(a \vee b) \vee c = a \vee (b \vee c)$, thus the evaluation of the policy P and any permutation P_π are equal $E(P) = E(P_\pi)$. The deny override follows similar semantics and follows a similar proof. ■

Using Theorem 1 policies with permit override or deny override rule combining algorithms can be permuted without affecting the policy semantics. This does not hold for other rule combining algorithms such as first applicable. We focus our discussion on permit and deny override combining algorithms for reordering optimization, while as will be discussed in the following sections policy based categorization is independent of the rule combining algorithm used.

Given a policy permutation π and a given request q, a subset of rules is of relevance. We represent an ordering of such rules as the execution vector.

Definition 7: (Execution vector) $\Gamma = [r_1, \ldots, r_n]$ is the execution vector representing the set of applicable rules, where rule r_i is executed before rule r_{i+1}. Where $\pi(i)$ refers to the position for rule r_i in execution vector.

According to Theorem 1, any policy execution vector for a policy P having permit overrides rule combining algorithm will evaluate to the same effect as P, the challenge is to evaluate the execution vector that will provide the lowest latency. Before presenting our optimal rule ordering approach, we need to define the rule weights.

B. Computation of Rule Weights

Our approach relies on statistics and metrics collected as PDP receives requests. Statistics are collected at two separate levels: *policy* and *rule* level. At the policy level, we are interested in understanding how often a policy applies, and by which class of users. At the rule level, it is important to identify the class of efficient execution vectors. In order to collect meaningful metrics, we assign to each rule (policy) weights that reflect the dominance of this rule in the requests. The weights are based on the PDP returned values, and constructed based on the 1) frequency and the 2) complexity of the rule (policy).

During a given time interval the number of times a policy P_i or a rule r_j gets evaluated is referred to as the hit frequency. We refer to the hit frequency by f and use the dot notation to refer to policy $(P_i.f)$ and rule $(r_j.f)$ hit frequency. Statistics with respect to the hit frequency are accumulated as follows:

- *Policy (Rule) Permit Ratio:* Records the ratio between the number of times a policy (rule) returns a permit with respect to the number of times a policy (rule) gets evaluated, where $P_i.p$ and $r_j.p$ represent the policy and rule permit ratios respectively.
- *Policy (Rule) Deny Ratio:* Records the ratio between the number of times a policy (rule) returns a deny with respect to the number of times a policy (rule) gets evaluated. Where $P_i.d$ and $r_j.d$ represent the policy and rule deny ratios respectively.
- *Policy (Rule) Hit Ratio:* Records the ratio between the number of times a policy (rule) is applicable with respect to the number of times a policy (rule) gets

evaluated. Where $P_i.a$ and $r_j.a$ represent the policy and rule hit ratios respectively.

Note that all the above statistics are easily derived from the XACML execution log. In addition to the rule evaluation statistics we also consider the rule computational complexity. Rules vary from simple conditions to more complicated statements that require the parsing of an XML document or querying a database. The rule complexity metric is related to the number of operations required to execute the rule, we compute it as the number of boolean atomic conditions appearing in a rule, both at target and at the condition element. Let $n(t)$ denote the number of conditions in the Target element (denoted as t according to Definition 3), and let $n(c)$ be the number of conditions in the Condition element c. XACML supports over 100 standard functions that could be used in the boolean conditions, for example the $Belong\_to$. We assign a cost m_i to each standard function std_i appearing in the rule. m_i is computed by estimating the average execution time of the function. The simple atomic boolean conditions are assigned a constant cost k. For a rule r_j the complexity metric is given by: $E_j = k * (n(r_j.t) + n(r_j.c) + \sum_{std_i \in r_j} m_i$ where std_i represents a uniquely identified standard function appearing in r_j. Using both the accumulated rule statistics and the complexity metric for a rule r_j we compute the rule cost as $c_j = \beta * E_j + \alpha * F_j$. Here, β and α are weights that allow system administrators to tune the cost computation, based on the local constraints, such as the available processing power and network bandwidth.

The rule cost is designed to represent the cost of computing a rule, the complexity metric E_j easily represents the rule cost, however the other component is based on the rule's accumulated statistics F_j. The value of F_j is based on the rule combining algorithm, for example if a rule combining algorithm is Permit-Overrides then the metric F_j is based on the decreasing function with respect to the rule permit ratio ($r_j.p$) or an increasing function with respect to the rule deny ratio ($r_j.d$). Intuitively, this implies that the rules need to be reordered such that for a policy with the permit overrides rule combining algorithm, the rule r_j with the lowest c_j to be evaluated first.

C. Optimal Rule Reordering

Using the rule cost metrics we present our optimal rule reordering problem. Given a policy (P_i), the optimal request execution problem (REP) is to find an execution sequence that requires the minimum number of rule evaluations. We assume that rules within policies are evaluated sequentially. The policy P_i, composed of n rules $\{r_1, \ldots, r_n\}$, where $\pi(j)$ refers to the position (depth) for rule r_j in the policy execution vector. The cost associated with rule r_j is referred to by c_j, as computed in Section III-B. The expected cost (average search length) for a given permutation π is given by: $\Phi_i = \sum_{j=1}^{n} c_j \pi(j)$. The main challenge is to compute the optimal policy permutation π that will generate the minimum expected policy execution cost. Additionally, among the possibly optimal π, we need to ensure the policy permutation to be safe, as defined in Definition 6. By computing Φ_i we are able to generate a cost metric for each policy P_i.

A policy set PS is composed of a set of policies $\{P_1, \ldots, P_m\}$. We assume the policies are executed sequentially. Using the minimum policy expected cost Φ_i, and the collected policy evaluation statistics, we compute the policy set execution sequence. The position of policy P_i in the policy set execution sequence is referred to by $\xi(i)$. The expected cost (average search length) for a given policy set (PS_k) permutation ξ is given by: $\Psi_k = \sum_{i=1}^{m} \Phi_i \xi(i)$. The computed costs for both the policyset and the policy reflect both the cost of the cost and statistics of execution. The costs Φ_i and Ψ_k are minimized when policies and rules are ordered in ascending order with respect to their costs [10].

Weights can be updated according to two different strategies: 1) periodically, 2) based on the last ρ received requests. In the first case, we update the weight values using the latest statistics. New execution vectors are constructed using fresh rule weights in order to boost up the hit performance close to its optimum level. The update period should be based on the predictable incoming request (e.g., certain months of the year) flow changes. In the latter case, the optimal execution vectors are constructed based on the computed rule weights. The incoming access requests are then processed according to the ordering determined. Intuitively, the maximal reduction is obtained when the incoming requests perfectly match the requests' distribution. Notice that more than one execution vector could be optimal and safe. However, since not all rules have the same complexity, different execution vectors may sensibly influence the overall evaluation time, even if a safe and efficient policy permutation is found.

IV. Categorization Based Optimization

The optimization problem minimizes the average request evaluation time. This approach is ideal if the policy requests follow a uniform statistic. If we solely rely on reordering, assuming a role based access control (RBAC) system of two roles, say *student* and *faculty*, where there are on average 100 student requests for every faculty request, the computed statistics will be dominated by the student requests. As such, the optimization problem presented above will favor the student role. Reordering rules and policies in these circumstances is not sufficient, as the computational cost will not be given by the evaluation of the rules themselves, rather it will heavily depend on the time spent on finding the applicable policies to the given request.

Hence, in order to further improve the efficiency of the rule reordering, we resort to clustering the policies. Building on execution vectors, an intuitive mechanism is to categorize the policies based on the subjects. Starting from a set of $L[S]$

clusters, where $L[S]$ is the number of subjects in S, the goal is first to reduce the number of categories in order to allow the reordering to have a considerable effect on the execution time. Second, to reduce the memory footprint needed for caching the categories.

To achieve these results, we propose adopting an algorithm based on the K-Means clustering method [13]. Generally speaking, the K-Means algorithm is used to cluster *m* objects based on attributes into *k* partitions, $k < m$. Each cluster consists of a "center" around which individual elements of the data set being clustered are grouped together. This grouping is done based on some measure of similarity to the other elements in that cluster. In our domain, the number of clusters N_c and the centers of these clusters, i.e. N_c subjects are chosen at random from the set of subjects *S*. The set of centers (or clusters) is referred to as C_s. Each subject $S_i \in S$ is considered, and its similarity $D_{i,k}$ is calculated with respect to each subject $S_k \in C_s$ in the different clusters. S_i will be added to that cluster where the similarity $D_{i,k}$ is maximum. The strength of this simple algorithm lies in the way the similarity metric $D_{i,k}$ is calculated. The similarity metric aims to cluster together the subjects that share a large number of policies which are applicable to all of them. Let $\mathbb{P}_i$ represents the set of policies applicable to a given subject S_i and let $L[\mathbb{P}_i]$ be the number of policies applicable to that subject. The number of policies shared between two subjects, S_i and S_k is given by $L[\mathbb{P}_i \cap \mathbb{P}_k]$. The fraction of the number of policies shared between the two subjects that are a part of $L[\mathbb{P}_i]$ is given by $\Theta_{i,k}$, where: $\Theta_{i,k} = \frac{L[\mathbb{P}_i \cap \mathbb{P}_k]}{L[\mathbb{P}_i]}$. The similarity metric $D_{i,k}$ between subject S_i and S_k is calculated as follows $D_{i,k} = \Theta_{i,k} + \Theta_{k,i}$. The subject S_i is grouped with the cluster centering on S_k where $D_{i,k}$ is maximum. This ensures that only those subjects which have a large number of policies in common are grouped together. In general, the clustering is more effective when the number of shared policies is large, i.e. when $L[\mathbb{P}_i \cap \mathbb{P}_k]$ is large.

This algorithm allows us to tune our optimization approach such that we can either maximize the improvement due to clustering or due to reordering, or both, based on the specific context. In general, the improvement due to clustering and categorization is most apparent when there are very large policies to process. On the other hand, for extremely simple policies with only one or two subjects, reordering is more helpful. In this scenario, reordering saves valuable execution time because by reordering, we can ensure that the policy does not do a brute force search to evaluate all the rules.

V. Experimental Results

Our experiments were performed on a MacBook Pro running Mac OS X 10.5.5 with 4GB of RAM and a 2.4GHz Dual Core Intel processor. We conducted two set of tests. The first test suite deals with XACML policy sets where subjects have a small number of applicable rules. The second suite investigates policy sets where subjects have a large number of applicable rules. All tests were conducted using 100,000 randomly generated XACML requests. All requests have a single value for the subject, resource, and action. Using both test suites we performed extensive experiments to investigate the performance enhancements yielded by our proposed categorization and reordering techniques. We also compared our results with Sun's PDP engine results.

Our experimental process includes two main stages; first, the setup stage and second, the actual request evaluations. The setup stage includes three sub-stages:

S1. Categorization of the experimental policy sets. Categorization is performed as explained in Section IV. The number of categories used for each policy set ranges from N to $N/10$, where N is the number of unique subjects within a policy set,

S2. Training stage that collects the results of request evaluations (permit, deny, not-applicable, indeterminate) subsequently used for the reordering stage,

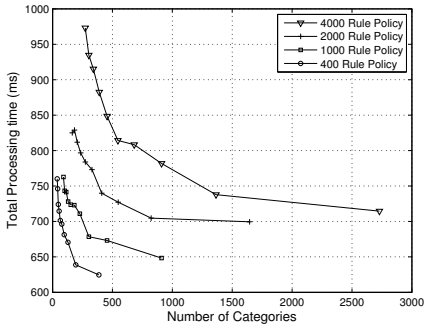
S3. Reordering policies within the policy set and all rules within each policy according to the statistics we gathered during the training stage.

The setup stage needs to be executed only once, however the sub-stages (S2) and (S3) could be executed repeatedly to retrain and reorder the policies and rules to achieve better performance. For our tests, we chose not to repeat the sub-stages, and thus measure the performance in the worst case scenario. The results of categorization and reordering are cached in memory. During the second stage the access requests are actually evaluated, using the ordering and categories set up in the previous stage. The processing time is the time needed to evaluate a request against a policy subjected to our setup stage plus the time to make a decision on that request. The preprocessing time is the time needed to complete the setup stage.

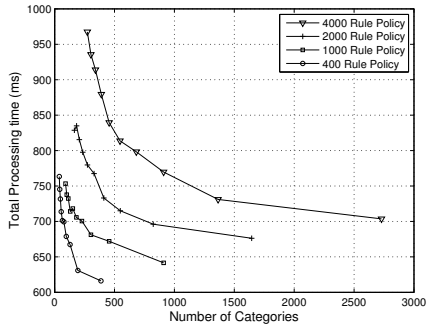
The experimental results show that our framework is orders of magnitude more efficient than Sun's PDP, and the performance difference between our framework and Sun's PDP grows linearly with the number of requests and number of rules within a policy set. We discuss the test results in details in the following subsections.

A. Test Suite I Results

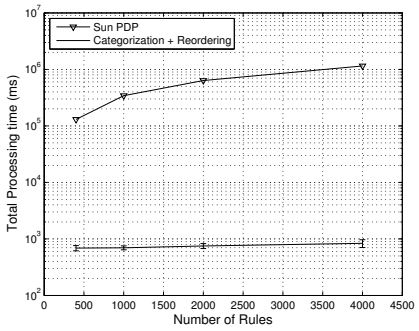
This test suite deals with policy sets where each subject has a few number of applicable rules. This test case is used to emphasize the effect of our categorization technique, whereas our reordering technique may have a minor effect. This test suite uses policy sets of 4000, 2000, 1000, and 400 rules. For each policy set, rules are divided evenly among 100 policies. For the sake of testing the *Permit Overrides* combining algorithm is used for all the test policy sets and policies. Using this test suite our approach is 1638 times faster than the Sun PDP.



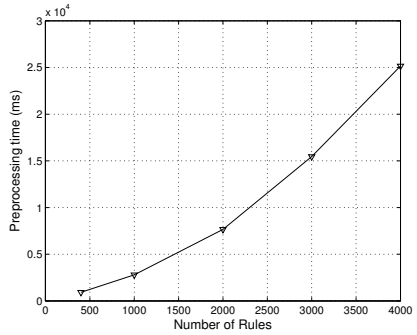
(a) Effect of categorization on evaluation time w.r.t # of categories used with no reordering.



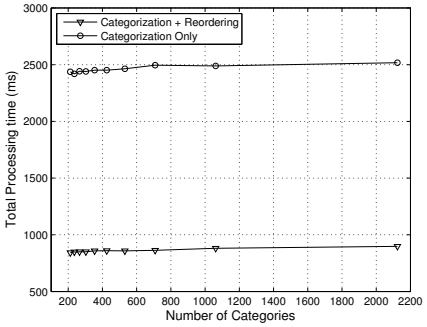
(b) Effect of categorization and reordering on evaluation time w.r.t # categories used.



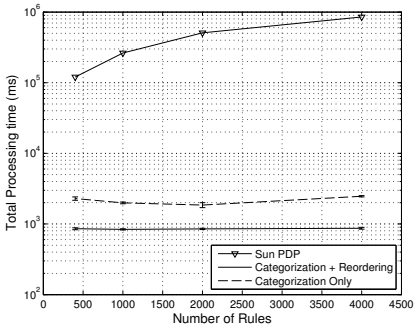
(c) Evaluation times comparison between our approach and Sun PDP.



(d) Preprocessing times including categorization and reordering.



(e) Performance boost from reordering w.r.t the # of categories using a 4000-rule policy.



(f) Sun PDP evaluation times compared to categorization only and categorization + reordering.

Figure 1. Experimental Results

1) Results with Categorization Only: We carried out a first set of tests only applying the categorization technique with no reordering. The number of categories used for each policy set was varied from N to $N/10$, where N is the number of unique subjects within a policy set. The preprocessing time for this approach is the time needed for categorizing a policy set (sub-stage S1.). When using N categories, results show that preprocessing a policy set of 100 policies and 4000 rules takes about 25138 ms and a policy set of 100 policies and 400 rules takes about 913 ms. When $N/10$ categories are used, preprocessing times are 23464 ms and 487 ms for the 4000-rule and 400-rule policy sets respectively.

The experimental results shows that the total processing times for our approach is at least 172 times faster than Sun's PDP. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 973.1 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 1152460 ms. A policy set with 400 rules takes 760.2 ms and Sun's PDP takes about 130421.3 ms. When N categories are used, total processing times are 714.6 ms and 624.6 ms for the 4000-rule and 400-rule policy sets respectively. Figure 1(a) shows the complete results when using categorization alone with respect to the number of categories used, which range from 0 to 3000.

2) Results with Categorization plus Reordering: For this set of tests, we applied the categorization technique, followed by our reordering technique. The number of categories used also range from N to $N/10$. We make use of all substages within the setup stage. Preprocessing time in this case is the time for both categorization and reordering of rules. Our results show that preprocessing time is proportional to the number of rules, as reported in Figure 1(d). Preprocessing a policy set of 100 policies and 4000 rules while using N categories takes about 25158 ms, and a policy set with 100 policies and 400 rules takes about 925 ms. When $N/10$ categories are used, preprocessing times are lower, 23472 ms and 491 ms for the 4000-rule and 400-rule policy sets respectively. The results for this set of tests are reported in Figure 1(b). The experimental results shows that the total processing times for our approach is at least 171 times faster than Sun's PDP. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 967.5 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 1152460 ms. A policy set with 400 rules takes 763 ms and Sun's PDP takes about 130421.3 ms. When N categories are used, total processing times are 703.7 ms and 616.2 ms for the 4000-rule and 400-rule policy sets respectively. Figure 1(b) shows our complete results when using categorization plus reordering with respect to the number of categories used. Figure 1(c) is a comparison between our approach with categorization plus reordering and Sun's PDP. The plot representing our approach is an average of the best and worst case we obtained from using different numbers of categories. The results obtained by this

set of tests report a very slight performance improvement due to the reordering. Reordering rules is not a significant factor to performance. because of the low number of rules applicable to each subject. Reordering's effect can be better appreciated for policy sets with many rules applicable to each subject.

B. Test Suite II Results

Our first test suite did not give us any indications about the reordering effect. We decided to generate a second test suite that could allow us to observe the effect of reordering on performance. This suite simulates a scenario where each subject within a policy set is guaranteed to have a significant number of applicable rules. When reordering happens in such a scenario, there will be no need to go over all rules within a subject's category. As expected, this test suite showed a significant performance advantage for the categorization plus reordering approach over the categorization only approach. We used policy sets of 4000, 2000, 1000, and 400 rules (different from the ones used in first test suite). For each policy set, rules are divided evenly among 100 policies. Overall, our results for this test suite show that our approach is 949 times faster than Sun's PDP engine. Similar to the first test suite, we conducted experiments using categorization only and categorization with reordering.

1) Results with Categorization Only: The preprocessing times for this case are inline with the times for the analogous set of tests (Section V-A) of the first test suite. Precisely, when using N categories, preprocessing a policy set of 100 policies and 4000 rules takes about 25397 ms and a policy set of 100 policies and 400 rules takes about 978 ms. When $N/10$ categories are used, preprocessing times are 28633 ms and 1075 ms for the 4000-rule and 400-rule policy sets respectively.

As in the previous test case, the results for total processing times show a very significant improvement in performance over Sun's PDP. Our results indicate that our mechanism provides at least 48 times faster evaluation. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 2437.2 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 851477 ms. A policy set with 400 rules takes 2272.2 ms and Sun's PDP takes about 120230.3 ms. For N categories, total processing times are 2517.6 ms and 2242.5 ms for the 4000-rule and 400-rule policy sets respectively.

2) Results with Categorization plus Reordering: Although the policies are different, we notice that the gathered times are very similar to the times recorded for preprocessing the set of policies used for the first test suite (reported in Figure 1(d)). This observation leads to the conclusion that the preprocessing time is not influenced by the type of policies used. The preprocessing times are almost negligible when compared to the highly significant performance improvement in total processing times over Sun's PDP, not to mention that preprocessing times correspond to the setup stage of our framework which only occurs once within a policy set's lifetime or upon a client's request.

Figure 1(f) compares Sun's PDP total evaluation times with our results from the second test suite. The total processing time of our approach is at least 139 times faster than Sun's PDP. As shown, for a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 842.3 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 851477 ms. A policy set with 400 rules takes 867.5 ms and Sun's PDP takes about 120230.3 ms. When N categories are used, total processing times are 897.6 ms and 830 ms for the 4000-rule and 400-rule policy sets respectively.
For the 4000-rule policy set used in this test suite, results indicate that categorization plus reordering has a 65.4% performance improvement over using categorization alone. Figure 1(e) shows the performance boost reordering provides with respect to the number of categorizations used. We notice a slight improvement in performance when the number of categories is reduced. This result is explained by the fact that the policy set we used has many rules that are applicable to all subjects, which means the resulting categories are not much different from the original categories.

Space Complexity. Concerning space complexity, our framework is relatively efficient. After sub-stage *a*, the categorized policy set will be cached in memory using a Hashtable (H_1). H_1 will be of size $N_c * L[\mathbb{P}_c] * L[\mathbb{R}_c]$, where N_c is the number of categories used, $L[\mathbb{P}_c]$ the number of policies within a category, and $L[\mathbb{R}_c]$ the number of rules within $\mathbb{P}_c$.

VI. Related work

XACML optimization and analysis has recently emerged as a new research topic in the area of XACML [6][8][2][11]. In [6], Liu et al. present the most recent proposal for the optimization of XACML policies. Liu et al, focus on improving the performance of the PDP by numericalization and normalization of the XACML Policies. The authors posit that since numerical comparison is more efficient, an improvement in performance is achieved by numericalization itself. The normalized policies are converted into tree data structures for efficient processing. We identify two main differences between our work and [6]. First, we rely on statistics, which help us defining the best ordering process based on *actual* users requests. Secondly, the authors unify all the rule combining algorithms into only the First-Applicable, while we do not require such cumbersome preprocessing stage. Finally, although no complexity evaluation is given by Liu et al., we believe our approach is more efficient in terms of space complexity, since it does not rely on storing complex data structures such as tables and trees. Another work related is by Miseldine [8]. Miseldine proposes to achieve policy optimization by minimizing the average cost of finding a match at the rule level the target level and the

policy level. The work assumes no changes to the XACML specification, in that the Sun's XACML implementation is not altered. The improvement is achieved by applying optimization techniques to the policies themselves. Therefore, anyone who consumes XACML remains structurally unaffected but anyone who generates XACML policies can generate an optimized output by applying the optimization techniques outlined. The main differences between [8] and our work arise in the way we try to meet this premise. While we focus on the reordering of rules and further on the categorization of the policies based on both the policies and the rules, Miseldine approaches this problem considering *policy configurations*. Although interesting, the improvements are drastically worse than ours. For example, their optimized method takes around 200ms for evaluation 4000 rules, where with our techniques, it only takes 1 ms. Kolovski et al [11], formalizes XACML policies using description logics (DL), and using the DL verifier they are able to detect and remove redundant XACML rules. The idea of removing redundant policies is interesting and may be useful to boost the evaluation time. However, it is yet to be validated whether the improvement will be worth the time needed to remove redundant policies, and how significant the overall improvement would be.

One related area where similar optimization techniques are often explored is Firewall Filtering [3][4]. The major differences between firewall optimization and XACML policy optimization arise because in the case of firewalls, a major portion of the traffic packets match a small subset of the firewall rules, and the same distribution of traffic is maintained over a significant period of time. This skewness is not experienced in the incoming requests for an XACML policy. Besides, firewall rules, which have dependencies on each other, have an order of precedence defined, while rules in an XACML policies are not related. Despite these differences between firewall filtering optimization and optimization of XACML policies, we can still draw from the body of work on firewalls. We employ metrics similar to the ones used by the authors for evaluating which rules would be most applicable to our policies. For example, frequency is useful in predicting the best match for a new incoming request which does not match any existing categories. The packet matching is a simple, single level problem as the only requirement is to match the packet's header against the rule list and performing the corresponding filtering. Our goal is more ambitious, since not only we try to find which policy is applicable to an incoming request but also we optimize the ordering of the rules within the policy to match the request.

VII. Conclusions

XACML policies and their evaluation play a critical role in many access control systems, where numerous requests are received by large set of subjects. This calls for high performance XACML policy evaluation engines. In this paper, we introduced a novel optimization framework based on statistics and policy set categorization. Our categorization technique, which is based on the K-means algorithm, provides fast access to applicable policies and rules for a certain subject. Reordering policies and rules within a policy set ensures that request evaluations are done on policies and rules that are most likely to return a positive effect; hence, avoid examining all policies and rules which are not likely to be significant for the access request being evaluated. We showed through experimental analysis the enhancement obtained for different set of policies of varying size and structures. Our results show that our techniques outperform the policy evaluation of the SUN PDP engine by orders of magnitude.

Acknowledgment

This work was partially funded by the National Science Foundation (NSF-CNS-0831360) and National Security Agency (NSA H98230-07-1-0231).

References

[1] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary cams can be smaller. *SIGMETRICS Perform. Eval. Rev.*, 34(1):311–322, 2006.

[2] D. el Diehn I. Abou-Tair, S. Berlik, and U. Kelter. Enforcing privacy by means of an ontology driven xacml framework. In *IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security*, pages 279–284, Washington, DC, USA, 2007. IEEE Computer Society.

[3] H. Hamed and E. Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 332–342, New York, NY, USA, 2006. ACM.

[4] H. Hamed, A. El-Atawy, and E. Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. In *INFOCOM 2006: Proceedings of the 25th IEEE International Conference on Computer Communications*, pages 1–12, April 2006.

[5] D. Lin, P. Rao, E. Bertino, and J. Lobo. An approach to evaluate policy similarity. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 1–10, New York, NY, USA, 2007. ACM.

[6] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: a fast and scalable xacml policy evaluation engine. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 265–276, New York, NY, USA, 2008. ACM.

[7] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino. Xacml policy integration algorithms. *ACM Trans. Inf. Syst. Secur.*, 11(1), 2008.

[8] P. L. Miseldine. Automated xacml policy reconfiguration for evaluation optimisation. In *Proceedings of the 4th International Workshop on Software Engineering for Secure Systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[9] T. Moses. Extensible access control markup language (XACML). *Technical Report, OASIS*, 2003.

[10] R. Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, 1976.

[11] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the International Conference on Management of Data*, pages 551–562, New York, NY, USA, 2004. ACM.

[12] Sun Microsystems Inc, Sun XACML Policy Engine. http://sunxacml.sourceforge.net/guide.html.

[13] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 2 edition, 2005.