Graph-Based XACML Evaluation

Santiago Pina Ros[*]
University of Murcia
Murcia, Spain
santiago.pina1@um.es

Mario Lischka[†]
AGT Group (R&D) GmbH
Darmstadt, Germany
mlischka@agtgermany.com

Félix Gómez Mármol
NEC Laboratories Europe
NEC Europe Ltd.
Heidelberg, Germany
felix.gomez-marmol@neclab.eu

ABSTRACT

The amount of private information in the Internet is constantly increasing with the explosive growth of cloud computing and social networks. XACML is one of the most important standards for specifying access control policies for web services. The number of XACML policies grows really fast and evaluation processing time becomes longer. The XEngine approach proposes to rearrange the matching tree according to the attributes used in the target sections, but for speed reasons they only support equality of attribute values. For a fast termination the combining algorithms are transformed into a first applicable policy, which does not support obligations correctly.

In our approach all comparison functions defined in XACML as well as obligations are supported. In this paper we propose an optimization for XACML policies evaluation based on two tree structures. The first one, called Matching Tree, is created for a fast searching of applicable rules. The second one, called Combining Tree, is used for the evaluation of the applicable rules. Finally, we propose an exploring method for the Matching Tree based on the binary search algorithm. The experimental results show that our approach is orders of magnitude better than Sun PDP.

Categories and Subject Descriptors

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

General Terms

Performance

Keywords

XACML, Evaluation

[*]This paper is based on the authors work during an internship at NEC Laboratories Europe

[†]This paper is based on the authors work while being employed at NEC Laboratories Europe

1. INTRODUCTION AND MOTIVATION

OASIS eXtensible Access Control Language (XACML) [11, 12] is a widely deployed standard language for access control policy specifications. It follows an attribute based model to identify atomic authorization elements (e.g subject and resource). The rules encoded in XACML are evaluated against a given request by a component called Policy Decision Point (PDP). Once a policy is specified, it evolves by and by to address new requirements and becomes large both in content and number of cases covered. This causes issues in policy evaluation run-time. Due to this known problem, there is a line of research to effectively handle large number of rules and policies with varying number of attributes.

Among the various efforts on this field, XEngine [5, 6] represents the state-of-the-art for XACML based authorization decision. XEngine has performance gains in orders of magnitude compared to SUN XACML engine (Corp.). It employs efficient data structures and special pre-processing techniques to improve the evaluation. The key idea is to preprocess the target section of the hierarchy formed by policy sets, policies and rules, and take into account that the conditions of a rule are only evaluated in case all the attributes in the target sections along the path are matching with a given request.

It is important to note that these attributes may be evaluated in any order, as long as the correct condition is reached in the end, or, in case of XEngine, just the final decision of accept or deny. In contrast to the evaluation of the target sections of policy sets, policies and rules, whose numbers could increase dramatically in large organizations, the run-time of this approach is limited by the number of attributes Ids (a), and their values (v_a) used in the target sections. Thus , XEngine has an average runtime of $O(a * log(max(v_a)))$ but as some limitations and drawbacks:

- The normalization process described in XEngine does not support XACML obligations although they are a fundamental construct in XACML.
- XEngine supports equal functions, while complex functions and comparison functions are not supported.
- The internal data structure (i.e. multi-valued decision diagram) causes extreme memory consumption.

Motivated by these issues, an enhanced concept for evaluation is presented in this paper. While we focused on the first two points of the above list, the multi-value requests and policies are currently not supported.

2. RELATED WORK

Since XACML was standardized by OASIS in 2003, a striking number of research works have been done on XACML so far. Most of the research work has been focused on verification, modelling, analysis and testing of XACML policies [1,4,8,13].

Some recently researches are focused in XACML optimization [6,7,10]. The proposals presented in [5,6] and [7] constitute the current state of the art in XACML optimization. [7] is based on statistics of the past requests and its matched rules, which are categorized according to a clustering-based technique, and tries to reorder policies and rules according to the statistics results. This approach, however, has two main flaws: i) reordering of policies does not support obligations, as we will see in section 4, and ii) the statistic method does not improve the processing time when the access requests are not uniform.

XEngine [5,6] focuses on improving the performance of the PDP by numericalization and normalization of the XACML Policies. In the implementation[1] *numericalization* is a hash function that converts every attribute type into integers and stored in a hash table with the objective of an efficient comparison. By doing like this, they achieve an improvement in performance, but are unable to handle any comparison functions. The *normalization*, in turn, converts every combining algorithm into a first applicable combining algorithm in order to build a flat policy structure from the original policies tree structure. Finally they build a tree with the numericalized and normalized policies for efficient processing of requests.

Our proposal has some similarities and differences with XEngine. The approaches are similar since both works try to achieve an improvement in the performance of the PDP using a tree data structure. Essentially both approaches build a decision path and store the matched rules at the end of the paths, but the differences are in the details. Since numericalization does not support comparison functions and our goal is to try to support most XACML specifications, we decided not to use it. For the same reason we also refused to use normalization, since reordering rules and policies does not support obligations, as mentioned before.

In our approach we build two types of trees, namely: Matching Tree and Combining Tree. The Matching Tree is similar to the tree built in XEngine without the numericalization process, and with different end nodes. As each edge in the tree is representing a concrete value[2], thus our approach does not support multi-valued requests or target sections.

The leaf nodes in XEngine are the normalized rules in a flat structure, while leaf nodes of a Matching Tree are actually Combining Trees. The latter consist basically of a structure that stores policies and rules preserving the original tree structure of the policy set. We will explain deeply the concepts of Matching Tree and Combining Tree in section 4.

3. PROBLEM STATEMENT

XEngine [5,6] is based on the idea of using decision diagrams for a faster evaluation, and experimental results have shown that the approach provides a number of magnitude faster evaluation compared to the SUN Reference Implementation (SUN RI).

The key idea is to transform the target matching of the policy sets, policies and rules into a decision diagram (DD). Decision Diagrams are widely used in verification tools, in particular, software systems used for hardware design. A DD is a directed acyclic graph, $G = (V, E)$, where the nodes V represent attributes (used in the target sections) and E represent the equality of the attributes in the request with a particular value. The edges are leading to the next attribute present in the target sections.

Binary decision diagrams (BDDs) have been studied extensively in the literature [2], and many variants [3], [9] have been introduced. A decision based on decision diagrams is obtained by combinations of variable assignments that lead to a terminal node in G.

The reason for the performance advantage of this approach is the limitation of the evaluation time by the length of the evaluation paths in the decision diagrams. In turn, the maximum length is limited by the number of attributes used in the various target sections and is not bound to the number of policy sets, policies and rules. In XEngine a technique called forwarding tables is used to find the correct edges in constant time. If the transformation is not used, some fast search algorithms are required to find the edge to the next node. While the general approach of XEngine is quite convincing, their decision diagrams lack support for several concepts:

- Incomplete list of attributes. In general, a subset of the attributes could lead to a decision. Yet, XEngine needs all the attributes to be able to reach a decision.
- Support of comparison functions. The domain approach especially causes problems if the value area is countable, but theoretically infinite (e.g. time of the request has to be larger than a specific date).
- Support of regular expressions match functions. As the algorithm for matching could not be transformed to a simple comparison, this functionality is not supported.
- Correct handling of DenyOverrides and PermitOverrides. In case the dominant effect does not determine the result, the correct set of obligations is not returned.

As one of our requirements has been to fully support the OASIS XACML 2.0 standard (having a clear view on how to support XACML 3.0 as well), we have carefully considered the aforementioned features in our solution.

4. IMPROVED CONCEPT OF GRAPH EVALUATION

In this section we will present an improved concept for XACML evaluation making use of a graph based on the attribute IDs of a policies tree. In Figure 1 a sample policy set, policies and rules are shown, which we will use to illustrate our concept. The hierarchical structure of this graph entails the inclusion of rules, policies and policy set (button up).

In our approach we are distinguishing between the search of the applicable rules for a request and the search of the correct effect in the applicable rules. Furthermore, we will show why this approach solves the limitations presented in

[1]In [5,6] all occuring attribute values are just enumerated.
[2] [5,6] are using the combination of multi value rules.

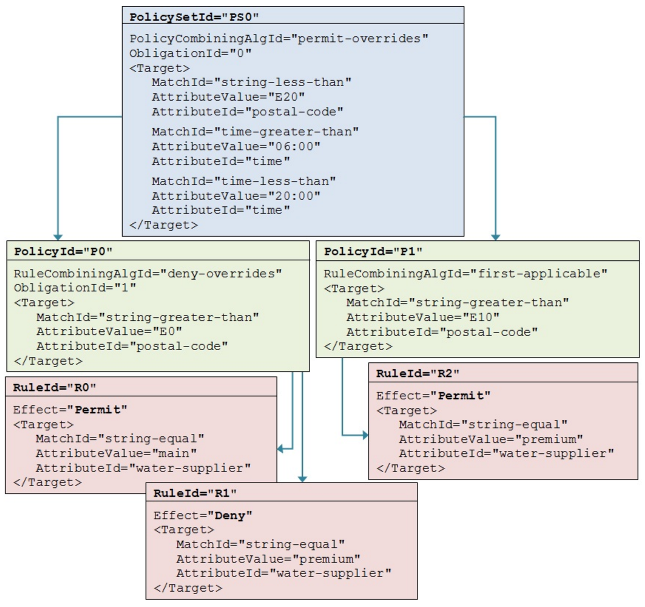

Figure 1: Example Policies Tree

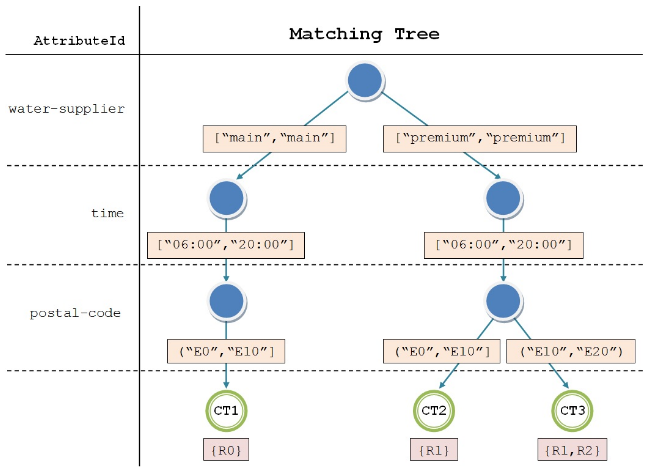

Figure 2: Corresponding Matching Tree

previous Section 3, in particular the correct support of obligations and all types of target matching functions.

In order to develop this separation we have created two data structures, namely: Matching Tree and Combining Tree.

The Matching Tree is used to find the applicable rules based on the values of the attributes in the various target sections. This tree has one level per different attribute Id, and the edges are represented by disjoint intervals of different comparable elements (e.g., ["SE10","SE20"], [5.3,7.8], [14:00, 15:30]). The evaluation of a request begins from the root node, it takes from the request the attribute value that corresponds to the attribute id of the current level, it searches the interval that contains such attribute value, and repeats the process with the node pointed by the selected interval. The evaluation is successful when a terminal node is reached, indicating that there is at least one rule whose target attributes are matching the request. When an attribute value taken from the request is not contained in any interval, there is no matching rule.

In Figure 2 we can see the Matching Tree built with the policies described in Figure 1. While the structure as such has been used in XEngine, we will discuss some optimizations regarding the sorting of the attributes in section 5.2. In the next sections the path (sequence of nodes) from the root node to a terminal node in the Matching Tree will be called the *MatchingPath* of this terminal node. In contrast to XEngine, each terminal node of the tree contains, in turn, a Combining Tree built with the rules that match with the path followed to reach this terminal node.

A Combining Tree is a structure based on the policies tree containing a subset of all the rules, where the targets of all the elements are empty, since a Combining Tree is built only with those rules which are actually applicable. This approach allows us to find the result of the request in a shorter time. In addition this structure preserves the original hierarchy of combining algorithms, which have to be applied to the obligations also included in the combining tree.

Regarding the support of incomplete list of attributes and complex functions (regular expressions, etc), we have to create special edges that need a separate evaluation. In that case the evaluation can take multiple paths in the Matching Tree, and we have to merge all the Combining Trees belonging to each path. There is a similar problem with requests with multi-valued attributes, and the solution is to make a separate evaluation for each attribute value and merge all the resulting Combining Trees again.

4.1 Combining algorithms

For XEngine, the policy combining algorithms DenyOverrides and PermitOverides are transformed into a FirstApplicable algorithm. This approach is only valid in the case where obligations are not used. In case the policy sets and their underlying policies contain obligations, then such solution is not longer valid, as the obligations of all evaluated policies or policy sets which have the same effect as the final decision have to be provided to the PEP [11, sec 7.14]. Let's assume a policy set applies DenyOverrides: if the resulting effect to the request is permit, then all underlying policies have to be evaluated and the evaluations have to check if there is an applicable rule with the effect deny. According to the standard, all obligations with the effect of the decision (i.e. Permit or Deny) have to be attached to the result, not only those of the first policy with the effect permit.

In case of incomplete attribute lists, usage comparison or regular expression functions, the evaluation is forked. Hence, multiple terminating nodes could be reached. This represents the case where multiple rules provide a result during the evaluation and policy combining has to be applied.

During the creation of the decision diagram, the original tree of policy sets and policies has to be examined and a Combining Tree is built with the rules of each terminating node. A Combining Tree preserves the tree structure of policy sets and policies, keeping the obligations and the combining algorithm of each node. In a Combining Tree the corresponding terminating nodes are rules, which can be reordered, assigning priorities to the rules with empty targets, in order to achieve a faster evaluation. If multiple terminat-

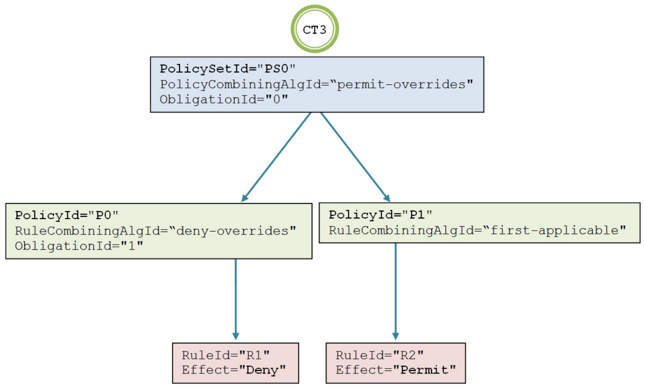

Figure 3: Example Combining Tree

ing nodes are reached during the evaluation, we will have to merge all Combining Trees.

4.1.1 Combining Tree

A Combining Tree (see Figure 3) is an auxiliary data structure based on the policies tree. It is essentially a subset of the tree without any target section. Initially one could think of reordering the policies or even changing the tree structure to a list with a special order of the rules, but if we want to preserve obligations, we have to keep the tree structure and the order of the policies. The policies have to be evaluated in the same order because otherwise the result could get a different obligation, and therefore the returned effect could be wrong.

Our goal is to keep all the applicable rules of a Matching Path in the corresponding leaf node. In order to preserve obligations we will create a Combining Tree with those applicable rules. At this point we should highlight that, as we can see in Figure 2, the same rule could be at the end of different Matching Paths, and subsequently, in different Combining Trees. As an example Figure 3 shows the Combining Tree CT3 of the Matching Tree shown in Figure 2

4.1.2 Rule Combining Algorithms

Since rules does not have obligations (at least in XACML 2.0; this is introduced in XACML 3.0 [12]), we can reorder them in the Combining Tree. The new order of the rules will depend on the combining algorithm of their parents in the Combining Tree. We will assign priorities to the rules in the original policies trees, to make it easier to build the Combining Tree with the sorted rules. When building a Combining Node referencing to a policy, its children rules will be added in descending order starting with the highest priority rule. The priorities are assigned as follows, depending on the rule combining algorithm:

- In case of a first-applicable combining algorithm, the priority is assigned to the rules in reverse order of the occurrence in the policy and increased each time, starting with priority 1.

- In case of a deny-overrides combining algorithm, all policies with the effect permit get the priority 1. The remainder rules with effect deny are assigned a priority in the reverse order of their occurrence in the policy.

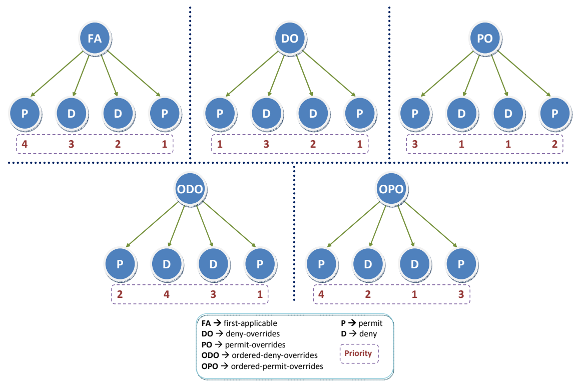

Figure 4: Priorities example

- In case it is ordered-deny-overrides combining algorithm, the priority to those rules with effect permit is again assigned in reverse order and increased by one each time starting with 1. Then those rules with effect deny are assigned in reverse order with a priority starting from the highest priority of the last permit rule plus one.

- In case of permit-overrides the assignment is done symmetrically (first rules with effect deny, then those with effect permit).

- In case of a ordered-permit-overrides combining algorithm, the priority to those rules with effect deny is again assigned in reverse order and increased by one each time starting with 1. Then those rules with effect permit are assigned in reverse order with a priority starting from the highest priority of the last deny rule plus one.

An example is shown in Figure 4 where the order of the leaf nodes correspond to the order in the containing policy, and the numbers are indicating the priority assigned based on the approach presented above.

4.2 XEngine comparison

XEngine represents the state-of-the-art for fast XACML based authorization decision, being currently, as far as we know, the fastest way to evaluate XACML policies. However, it also has some limitations. The main differences in the structures presented in this paper and the ones presented in XEngine are twofold:

- XEngine maps every attribute value to a numerical value and the PDD based in numerical intervals, while we build a Matching Tree with only one level per attribute value and the edges are represented with generic intervals.

- XEngine transforms the policies tree structure to a flat structure with the first-applicable combining algorithm. On the other hand we build Combining Trees, that preserve the tree structure and the combining algorithms.

In the following we present a qualitative comparison between XEngine and our approach.

4.2.1 *Correctness*

XEngine presents a formal proof of the correctness of Normalization, but since we do not modify the tree structure of the policy sets and policies it is easy to see that our approach find the correct solution. We separate the evaluation in two parts, first we search the applicable rules with the Matching Tree; once we have the applicable rules we create a Combining Tree preserving the original structure of the overall policy set, and we use the original algorithms to evaluate the request. So, we use the original structure, containing the applicable rules and we evaluate the response using the original algorithms, then the response must be the same. In this case both approaches are correct.

4.2.2 *Support of comparison functions*

Decision diagrams are optimized to support the check of equality of attribute values. In XEngine a range check is used for String values, but these ranges are determined by the values parsed in the policies. However, the request may contain additional values not yet known. Thus, the mapping to integer technique does not seem to be adequate.

The reason is simple, you can always find an string between any two given strings, but you can not find an integer between any two given integers. For example, you can not find any integer between 4 and 5, but given the string “x” and the string “y”, with “x” < “y”, (“x” +“a”) is between “x” and “y”. For that reason we can not find any bijective function from strings to integers.

A first approach is to find a bijective function from strings to a real number, but it present two problems:

- String comparison seems to be faster than most mapping functions, since you have to explore every char in the string.
- The precision of the mapping could be as much between 7 and 10 chars, due to the precision of doubles.

In order to solve this problem we use data-intervals. A data-interval is a structure similar to a numerical interval but composed by a different comparable data type. Intervals of dates, char, etc., are “natural”, but we need to add two new values to the value set of each data type: left infinity, and right infinity.

They are ordered according to the attributes given in the target section and used for comparison. During the evaluation these lists are checked and whenever the attribute value given in the request is valid for a particular comparison, the corresponding edge has to be evaluated has well. As in the aforementioned case of missing attributes, it has to be determined whether this should be done in a concurrent or sequential fashion. As the lists are ordered according to the respective comparison operator, they are traversed until the first comparison fails.

In addition, some early experiments have indicated that computing the hash value of a string takes significantly more time than a string comparison.

Mapping String to Integer is a good idea, since Integer comparison is fast than String comparison, but as explained before we lost the String comparison functions support. So if String comparison functions are needed it is recommended to choose the MatchingTree solution rather than the Decision Diagram solution.

4.2.3 *Incomplete lists of attributes*

Assuming a variety of different attributes used in the target sections, it is more than likely that not all known attribute Ids are used in a decision path. The order in which the attributes are checked, i.e., on which level they are present in the decision diagram, is arbitrary or subject to further optimization strategies. At some point (due to the presence of the attribute in one target section and the absence in another target section) a special handling is necessary. In this case two paths have to be evaluated: the one which contains the attribute (assuming that the attribute is present in the request) and the one which does not introduce such attribute.

Whether this forked evaluation is done in a concurrent fashion or the open ends are stored for a sequential analysis has to be determined at a later stage. Both approaches introduce some coordination overhead. Obviously, empty target categories for subjects, resources or actions are handled in the same way.

Special edges built specifically for empty attributes IDs are needed to support empty targets sections in a decision diagram solution, but XEngine does not mention this case and cannot supports it.

4.2.4 *Regular expression support*

The use of regular expression in the target sections has to be handled in a similar way than comparisons. As there is no simple replacement for this function, the matching has to be evaluated. In case of a match, the respective edge has to be evaluated (concurrently or sequentially). Since the regular expression match operations are quite expensive computations, the attributes which are checked with this function should be moved to the end of the potential decision paths. For this reason a special edge has to be created for each match function. This way, an earlier missing or mismatching attribute might already terminate the evaluation before these nodes are reached.

Regular expression can not be mapped to a single integer value, so the numericalizarion method can not be applied to this kind of function and the regular expression has to be evaluated. On the other hand the Matching Tree creates a special edge for this kind of functions and fork the evaluation. This issue should be studied deeply in future work since both approaches seem to be computational expensive.

4.2.5 *Indeterminate response support*

In XACML specification is described the policy combining algorithms and the response of the evaluation of the algorithm can be deny or permit if the policy combining algorithm get an Indeterminate response evaluation a rule. This issue is not contemplated in flat structures of rules like the one used in XEngine, so it is needed preserve the tree structure if we want to give an adequate response in this cases.

4.2.6 *Obligations handling*

In the traditional evaluation like SUN PDP, a decision is reached on rule level and the result is sent up again. During this backtracking, the obligations specified in the traversed policy sets and policies have to be collected. This collection is then sent to the PEP as part of the result. During the creation of the decision diagram, the obligations specified in the policy sets and policies have to be tracked, and it has to be ensured that all obligations which are along a specific path are stored in the final node which also contains the condition to be evaluated. This way whenever a terminating node is reached, the related obligations just have to be sent with the result. In our solution we propose Combining Trees for the final evaluation of a request and, according to this approach, we can handle obligations with the algorithm of the XACML specification.

A solution with obligation support also needs to preserve the tree structure of the applicable rules, in order to collect the obligations once the result is found. The flat structure of rules and the reordering into first-applicable combining algorithms used in XEngine does not take into account the special cases where the none dominant result (i.e. deny in case of permit-overrides) is determined. Thus XEngine only provides the set of obligations related to the first applicable policy (set), instead of all the obligations of all policies evaluated according to [11, 12] and done in our approach.

4.2.7 *Multi-valued policies and requests*

As we have previously mentioned, our approach supports requests with multi-valued attributes, but the Matching Tree does not support policies (or more precisely predicate path lists) with multi-valued attributes. This is due to the current concept in which each level of the Matching Tree represents a different attribute Id and the edges of the tree are disjoint intervals. Based on this, it is not possible to build an edge with an interval that match only with two given values. As part of our future working we are evaluating the idea of adding dedicated edges or sets of edges to support policies with multi-valued attributes.

On the other hand, XEngine support this characteristic with the use of several levels per attribute Id. This point could be important in some environments, so in this case the use of XEngine could be more interesting.

4.2.8 *Processing time*

An important goal of this paper is to reduce the processing time of evaluation and, as we will see in section 6, our approach is several times faster than SUN implementation. XEngine is also several time faster than SUN RI and two points lead us to think that XEngine could be faster than our approach:

- The use of Integer comparison is faster than the use of String comparison. and the numericalization process convert every String to Integer. This numericalization is done in XEngine by a mapping. Initial experiments indicated that hashing the attribute values of the request will void the advantage of integer comparison. Thus, we preserve the string values, also to support comparison and regular expression functions.
- XEngine transform every combining algorithm to first-applicable combining algorithm, and the response is found faster with this algorithm. Since obligations are not supported with this kind of transformation, we decided to preserve the correct combining algorithms in the Combining Tree.

So in environments were the speed is valued over functionality XEngine is more suitable, but in environments were functionality is valued over speed our approach is more appropriate.

5. SOLUTION DETAILS

In this section we will describe the needed algorithms for building the Matching Tree and the Combining Tree, as well as the algorithms for a correct evaluation of these trees.

5.1 Preliminary concepts

Here we present some concepts and terms needed to better understand the aforementioned algorithms:

- Predicate ($P = (att, f, v)$): It is a tuple of the three elements, namely attribute id, function and attribute value, where the function could be any of the ones presented in the XACML specification. The function specifies if a request matches with the Predicates.
- PredicatePath ($PP = (P_1, \dots, P_n)$): It is a list of Predicates. It is used to represent all the Predicates that a request has to satisfy for matching the target of a series of elements (policy sets, policies or rules). A request matches one PP iff the request matches $P_i \in PP \, \forall \, i \in [1, n]$.
- PredicatePathList ($PPL = (PP_1, \dots, PP_n)$): It is a list of PredicatePaths. A request matches one PPL iff $\exists \, i \in [1, n]$ such that the request matches PP_i.
- IntervalPath ($IP = (I_1, \dots, I_n)$): It is a list of Intervals of different data-types. This list contains exactly one interval for each different attribute Id.
- IntervalPathList ($IPL = (IP_1, \dots, IP_n)$): It is a list of IntervalPaths.
- Element (E): One Element can represent a Rule, a Policy or a PolicySet and it conserves their properties (i.e. obligations, effect, etc).
- CombiningNode (CN): A CombiningNode is a node of the Combining Tree and it is defined by an Element E.
- CombiningNodeList ($CNL = (CN_1, \dots, CN_n)$): It is a list of CombiningNodes. It is used to collect a complete branch where CN_1 is the root node, and CN_n is one leaf.
- Builder ($B = (CNL, IPL)$): It is an auxiliary structure used to collect information and to build both trees (Matching Tree and Combining Tree).
- BuilderList ($BL = (B_1, \dots, B_n)$): A BuilderList is a list of Builders, and it is the starting point for building the Matching Tree.
- Node ($N = (leaf, CT, edges)$): N represents a node of the Matching Tree composed by three elements. $leaf$ is a boolean value that indicates whether the node is a leaf or not. CT is the Combining Tree created with the rules that match with the path, and it is empty if the node is not a leaf. Finally $edges$ represent a list with the edges of the node.
- Edge ($e = (I, N)$): The edges of the Matching Tree, e, represent the pair (I, N), where I is an interval and N is a node.

5.2 Transformation algorithm

Once we know the concepts used in the algorithms, we will describe some functions used that will help us to better understand the process of building both the Matching and the Combining Trees.

- `extractPredicatePaths(E)`: this function, used in the line 6 of the algorithm 1, extracts a PPL from the target of the element E such that a request matches with the target iff the request matches with one of the $PP \in PPL$.

- `addRestriction(I,P)`: this function, applied in the line 9 of the algorithm 2, adds the restriction extracted from the predicate P to the interval I (i.e. function = (x < "E10") ⇒ return $I \cap (-\infty,$ "E10")). If the function is a "complex function" (i.e. regular expression) it is attached to the interval and it has to be executed each time that you check if $x \in I$.

- `sort(IP)`: as we know, each interval I in the intervalPath IP depends on one attribute id. Then we can find a bijective function $\mathcal{F} : IP \to AT$, where AT is the set of every attributesIds. If we have an order in AT then we can define an order in IP such that $I_1 < I_2 \Leftrightarrow \mathcal{F}(I_1) < \mathcal{F}(I_2)$; $I_1 > I_2 \Leftrightarrow \mathcal{F}(I_1) > \mathcal{F}(I_2)$; and $I_1 = I_2 \Leftrightarrow \mathcal{F}(I_1) = \mathcal{F}(I_2)$. The order in AT has to be always the same regardless of the IP of the argument. How to order AT is a difficult question and we will present one possible order at the end of this subsection.

In the following lines we will describe 4 algorithms used for the construction of the Matching Tree and the Combining Tree. We can separate the algorithms in two principal functions, one collect the information used to build the trees and the second builds the tree using the collected information, So first we need an algorithm to extract the important information from the policies tree and store it in new structure. We want to use this structure to build the trees, so we will extract the relevant information of each rule and later we will add the rules one by one to the Matching Tree. Then we will store two things for each rule of the PolicySet in the structure BL:

- The list of IntervalPaths that a request has to satisfy for being applicable to that rule.

- The list of CombiningNodes built with the parents' policies.

We extract them using the recursive function `extractPaths()` described in Algorithm 1, that explores the policies tree and return the mentioned information. Once have all the information needed to build the Matching Tree and the Combining Tree we need an algorithm to add the rules into a given Matching Tree (starting empty) using this information. This algorithm is described in Algorithm 3, and for each node of the existing Matching Tree the function takes the interval information of the given rule and builds the correspondent edge and node. When the algorithm reach a terminal node the Combining Tree of the given rule is added. So once we have the information in the structure BL we have to add the rules one by one with this algorithm.

So the first step is to obtain the BL using the function `extractPaths()` described in Algorithm 1. The function calculates the BL starting from E, and using the information of CNL and PPL_0 that is relative to the parents of E. The algorithm explores the policies tree, it takes for each element the CNL and the PPL of its parents and joins them with its own CN and PPL. When the current element is a rule, we transform the PPL into an IPL with the function `predicatesToInterval()`, described in Algorithm 2, and return a new BL containing its B. When the current element is not a rule the algorithm calculate the BL of its children, and returns the union of them.

Algorithm 1: `extractPaths()` function

```
Input: $PPL_0 = (PP_1, \ldots, PP_n)$;
       $CNL = (CN_1, \ldots, CN_m)$;
       $E$, where $E \in \{'Rule', 'Policy', 'PolicySet'\}$
Output: $\{BL_0 = (B_1, \ldots, B_n)\}$

 $PPL_1 \leftarrow extractPredicatePaths(E)$;

 if ($PPL_0 = \emptyset$)
   $PPL_2 \leftarrow PPL_1$;
 if ($PPL_1 = \emptyset$)
   $PPL_2 \leftarrow PPL_0$;

 for each $PP_i \in PPL_1$
   for each $PP_j \in PPL_0$
     $PPL_2.PP_{n+1} \leftarrow (PP_i \cup PP_j)$;

 $CN.E \leftarrow E$;
 if ($E = 'Rule'$) {
   $CNL_1 \leftarrow CNL_0 \cup (CN)$;
   $B.CNL \leftarrow CNL_1$;
   $B.IPL \leftarrow predicatesToInterval(PPL_2)$;
   $BL_0 \leftarrow (B)$;
   return $BL_0$;
 } else {
   $CNL_1 \leftarrow CNL_0 \cup (CN)$;
   for each ($child \in E.children$) {
     $BL_1 \leftarrow extractPaths(PPL_2, CNL_1, child)$;
     $BL_0 \leftarrow BL_0 \cup BL_1$;
   }
   return $BL_0$;
 }
```

The function `predicatesToInterval()`, described in the Algorithm 2, transforms a PPL to an IPL. For each predicate P of each PP, the algorithm adds the restriction established by the function of the predicate, to the interval associated to the attributeId of the predicate. Then it forms an IP with the intervals extracted form the PP. The result is the union of this IP in an IPL.

Algorithm 2: `predicatesToInterval()` function

```
Input: $PPL = (PP_1, \ldots, PP_n)$
Output: $IPL = (IP_1, \ldots, IP_n)$

 for each $PP_i \in PPL$ {
   for each ($attributeId\ k$)
     $I_k \leftarrow (-\infty, +\infty)$;

   for each $P_j \in PP_i$
     $I_{P_j.attId} \leftarrow addRestriction(I_{P_j.attId}, P_j)$;

   for each ($attributeId\ k$)
     $IP_i \leftarrow IP_i \cup (I_k)$;

```

```
        $IPL \leftarrow IPL \cup (IP_i)$;
    }
    return $IPL$;
```

The function `addNodes()` (Algorithm 3) is a recursive algorithm that adds to the Matching Tree the corresponding nodes and edges, extracting the information from a sorted IntervalPath IP. If the current node is not terminal, the algorithm add a new edge to N built with the interval I_L; but since the intersection of the edge's intervals must be empty, we need to modify the current edges of N and add some news. For each $e \in N$.edges, such that e.I $\cap I_L \neq \emptyset$, we need to add the edge e'.I $\leftarrow$ e.I $\cap I_L$, and then e'.N will be the combination of e.N and I_{L+1} using the function `addNodes()`. Once we have added the interval e.I $\cap I_L$ to a new edge, we need to modify the old edge e.I $\leftarrow$ e.I - (e.I $\cap I_L$). Finally we need to subtract to I_L every e.I such that $e \in N$.edges, and add a new edge with the resulting interval I_L. When we reach a terminal node we have to merge the existing Combining Tree with the given CNL.

Algorithm 3: `addNodes()` function

```
Input: $N$, a node of the Matching Tree;
        $IP = (I_1, \ldots, I_n)$;
        $CNL = (CN_1, \ldots, CN_m)$;
        $L$, depth level of $N$
Output: $N'$

 if ($L = n + 1$) {
   $N'.leaf \leftarrow$ true;
   $N'.CT \leftarrow N.CT \cup CNL$;
 } else {
   $N'.leaf \leftarrow$ false;

   for each (edge $e \in N.edges$) {
     $e'.I \leftarrow e.I \cap IP.I_L$;
     $e'.N \leftarrow addNodes\ (e.N,\ IP,\ CNL,\ L + 1)$;
     $N'.edges \leftarrow N'.edges \cup (e')$;
     $e.I \leftarrow e.I - e'.I$;
     $N'.edges \leftarrow N'.edges \cup (e)$;
   }

  for each (edge $e \in N'.edges$)
    $IP.I_L \leftarrow IP.I_L - e.I$;

  $e'.I \leftarrow IP.I_L$;
  $e'.N \leftarrow addNodes(\emptyset,\ IP,\ CNL,\ L + 1)$;
  $N'.edges \leftarrow N'.edges \cup (e')$;
 }

 return $N'$;
```

Finally, the Algorithm 4 builds the Matching Tree based on the above functions. The algorithm extracts the PredicatePathLists PPL and CombiningNodeList CNL of each rule starting from the root of the policies tree. Then it sorts every IntervalPath IP, as we will explain next, and adds each one with its corresponding CombiningNodeList CNL to the tree with the function `addNodes()` (algorithm 3).

Algorithm 4: Obtaining the Matching Tree

```
Input: $RootPolicy$
Output: $MT$, the Matching Tree

 $BL \leftarrow extractPaths(RootPolicy)$;

 for each ($B_i \in BL$)
   for each ($IP_j \in B_i.IPL$) {
     $sort(IP_j)$;
     $MT \leftarrow addNodes(MT,\ IP_j,\ B_i.CNL,\ 0)$;
   }

 return $MT$;
```

As we commented before, we need to order the attributeIds. We propose to sort the attributeIds in increasing order by the complexity of the attributeIds. Thus, the algorithm should reach less "complex functions" during the search when a path is truncated. For calculating the complexity of an attributeId in the variable C=0, we take all the predicates that contain this attribute id an depending on the complexity of the functions we add a different value (equal$\rightarrow$ C+= 1, compare$\rightarrow$ C+= 10, regexp$\rightarrow$ C+= 100). By this way the evaluation will do less unnecessary complex functions. This is just a first approach and it is a good point for optimize in future researches.

5.3 Evaluation Algorithm

Algorithm 5 shows how to obtain the appropriate Combining Tree corresponding to a given Request. It explores the Matching Tree obtained from algorithm 4 using a "depth-first search" and merging all the reached Combining Trees.

Finally, the resulting merged Combining Tree is evaluated against the Request according to the combining algorithms described in the XACML standard.

Algorithm 5: Obtaining the Combining Tree (`getCombiningTree()` function)

```
Input: $Request$; $N$, a node of the Matching Tree
Output: $CT$, the Combining Tree

 if ($N.leaf$ = true)
   return $N.CT$;

 for each (attribute $att \in Request.Attributes$)
   for each (edge $e_i \in N.edges$)
     if ($att \in e_i.I$) {
       $CT' \leftarrow getCombiningTree(Request,\ e_i.N)$;
       $CT \leftarrow CT' \cup CT$;
     }

 return $CT$;
```

5.4 Binary Search Evaluation Algorithm

With the aim of enhancing even more the evaluation processing time, it is possible to make some modifications in the data structures and in the evaluation algorithm. The first modification is to sort the list of edges of each node in the Matching Tree. To this end, we should have two lists: one with the special edges, and a sorted list with the interval edges. The second modification is to use a binary search algorithm in the evaluation algorithm for finding the correct edge in the interval edges. The main reasons for using this algorithm are:

- Intervals are disjoint, so an attribute can only be contained in one interval.
- Intervals have comparable data-type, so it is easy to sort the edges.

The new evaluation algorithm uses the method `binary-Search()`, as shown in algorithm 6. This method returns:

- The edge, extracted from `Edges`, whose interval contains the attribute.
- An empty edge if the attribute is not contained in any interval from `Edges`.

Algorithm 6: Obtaining the Combining Tree using binary search (`getCombiningTreeBS()` function)

```
Input: Request; N
Output: CT, the Combining Tree

 if (N.leaf = true)
    return N.CT;

 for each (attribute att ∈ Request.Attributes) {
    edge e ← binarySearch(att ∈ N.intervalEdges);
    if (e ≠ ∅) {
       CT′ ← getCombiningTreeBS(Request, e.N);
       CT ← CT′ ∪ CT;
    }
    for each (edge e′ ∈ N.specialEdges)
       if (att ∈ e′.I) {
          CT′ ← getCombiningTreeBS(Request, e′.N);
          CT ← CT′ ∪ CT;
       }
 }

 return CT;
```

6. EXPERIMENTAL COMPARISON

For an experimental examination of our approach we created several policy sets with a different amount of attribute IDs (ranging from 6 to 12) and different total number of policies (ranging from 20 to 400) each policy with 10 rules. The number of levels of each policy set depends on the number of policies contained, then the overall policy sets formed with 20 to 60 policies has 2 levels of depth, the policy sets composed with 80 to 200 policies has 3 levels of depth, and the ones with 220 to 400 policies has 4 levels of depth. A random mixture of combining algorithm has been used based on [14].

The targets of each policy set are composed by "equal functions" and "complex functions" (string-greater-than, string-greater-or-equal, etc), the 70% are "equal functions" and the rest are "complex functions". Every combining algorithm is presented in each test set the same number of times, so each combining algorithm represents the 25% of the total number of combining algorithms. We did not include obligations in our experiments since the processing time is not affected by the obligations.

These testing sets have been generated in a similar way than those presented in [14]. The target attributes of synthetic policies used in the experiments of [5, 6] only differentiate in their attribute values i.e. one attribute ID for subject, one for resource and one for action. Thus, the main effect of a faster evaluation is gained from the quick lookup of the matching attribute value.

The algorithms has been implemented in Java, using some auxiliar data structures (List, Hashtable, etc) available in Java framework. The PDP developed has been integrated in NEC XACML implementation, so during the experiments we have compared just evaluation time of the PDP in both implementations.

Each setup has been tested with the reference implementation (SUN), our simple solution (Tree), and our solution based on Binary Search (BsTree). As mentioned in section 4.2 our goal is to improve XEngine functions support but keeping faster than SUN PDP, as we do not targeted to improve XEngine speed an experimental comparison is not necessary at this point. For each experiment we measured the processing time of the given policy sets, required to evaluate 100 requests.

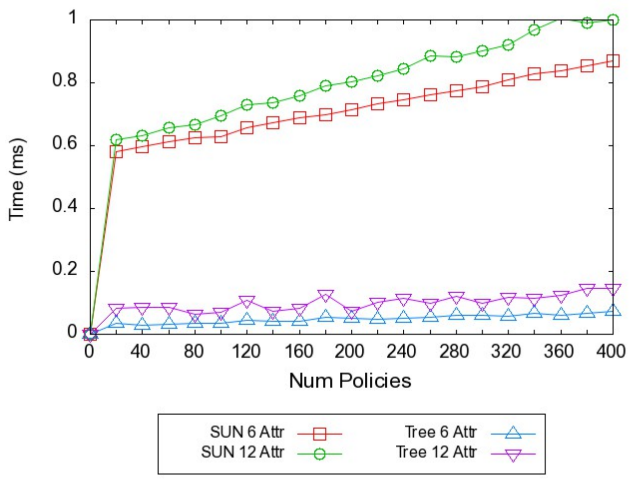

Figure 5: Comparison with SUN RI

The experimental results show that our first approach is one order of magnitude faster than Sun PDP, as shown in Figure 5. The problem is that the processing time of SUN and Tree growth linearly with the number of policies-rules. It is worth noticing that the growth rate of Tree is lower. this fact motives for our second approach: `Binary Search Evaluation Algorithm`, explained in the subsection 5.4.

The Figure 5 shows the comparison of the processing time of a request evaluation with the SUN implementation and the one of our first tree based approach for 6 and 12 different attribute IDs. The processing time with our approach is one order of magnitude faster than the SUN PDP. Due to the scale an increase of the processing time in our approach with a rising number of policies is barely notable. The evaluation time roughly doubles when we quadruple the number of policies. As discussed in Section 4 the number of levels of the Matching Tree is the number of different attribute IDs used in the policies represented by the tree; thus the processing time should increase significantly with the number of levels of the tree. In addition the Matching Tree becomes wider with the number of policies, because each policy adds new functions to the tree; then processing time should increase with the number of policies.

In Figure 6 we compare the processing time of a request evaluation of our different approaches.The processing time of the `Binary Search Evaluation Algorithm` is about two times faster than the simple approach. We assume that the fluctuation in the results is based on the preciseness of the measurement in combination with the tree structure we are utilizing to store the attribute values. Overall the processing time of the `Binary Search Evaluation Algorithm` grows logarithmically with the number of policies, while the result

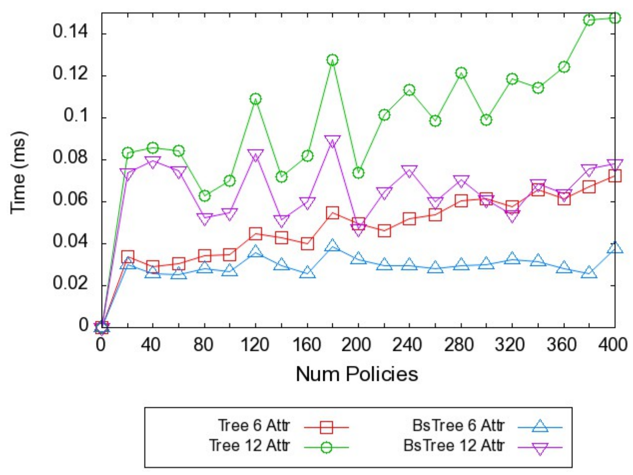

Figure 6: Comparing Tree and Binary Search

of the simple `Evaluation Algorithm` indicate a linear with the number of policies.

For real systems which concurrently receive a huge number of request, reducing the linear dependency on the number of policies to be evaluated to a logarithmical one has a large impact. Although the improvement of a binary search tree over the initial tree result shows quite some fluctuations, we can detect that the evaluation only took half the time. In real systems with a large number of requests, this is a notable improvement.

7. CONCLUSIONS AND FUTURE WORK

Nowadays XACML policies are used in a large number of applications and some of them receive a big amount of requests. These application could range from a web service, to an identity prover, or an application. Thus there is a general need for an high performance XACML PDP, independent of specific use case characteristics. Therefore instead of focusing of one particular scenario we utilized the generic set of test policies as presented in [14].

In this paper we presented an optimization of XACML policies evaluation which support most of the XACML specifications, while future work will include the support of multi-valued requests. We explained two new concepts: Matching Tree, based in XEngine tree, and Combining Tree. We built a new data structure with these concepts and presented two different evaluation algorithms over this structure. The first was a simple evaluation algorithm, while the second was an enhanced version based on a binary search. Finally we showed that our approach is orders of magnitudes better than SUN PDP but still being completely aligned with all features required by the XACML standard. Since our approach supports multi-valued requests, but not multi-valued policies, the future work is focusing on the support of multi-valued policies without increasing the memory utilization.

8. REFERENCES

[1] Dhiah Diehn I Abou-Tair, Stefan Berlik, and Udo Kelter. Enforcing Privacy by Means of an Ontology Driven XACML Framework. In *Proceedings of the Third International Symposium on Information Assurance and Security*, pages 279–284, Manchester, United Kingdom, 2007. IEEE Computer Society.

[2] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, April 1997.

[4] Dan Lin, Prathima Rao, Elisa Bertino, and Jorge Lobo. An approach to evaluate policy similarity. *Proceedings of the 12th ACM symposium on Access control models and technologies SACMAT 07*, page 1, 2007.

[5] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Designing Fast and Scalable XACML Policy Evaluation Engines. *IEEE Transactions on Computers*, 60(12):1802–1817, December 2011.

[6] A.X. Liu, F. Chen, J.H. Hwang, and T. Xie. XEngine: A fast and scalable xacml policy evaluation engine. In *ACM SIGMETRICS Performance Evaluation Review*, volume 36, pages 265–276. ACM, 2008.

[7] Said Marouf, Mohamed Shehab, Anna Squicciarini, and Smitha Sundareswaran. Adaptive Reordering & Clustering Based Framework for Efficient XACML Policy Evaluation. *IEEE Transactions on Services Computing*, 4(4):300–313, October 2011.

[8] Pietro Mazzoleni, Bruno Crispo, Swaminathan Sivasubramanian, and Elisa Bertino. XACML Policy Integration Algorithms. *ACM Transactions on Information and System Security*, 11(1):1–29, 2008.

[9] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM.

[10] Philip L Miseldine. Automated xacml policy reconfiguration for evaluation optimisation. *Proceedings of the fourth international workshop on Software engineering for secure systems SESS 08*, pages 1–8, 2008.

[11] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, February 2005.

[12] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*, April 2009. Comittee Draft 1.

[13] Shariq Rizvi, Alberto Mendelzon, S Sudarshan, and Roy Pollock. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the International Conference on Management of Data*, pages 551–562, 2004.

[14] Fatih Turkmen and Bruno Crispo. Performance evaluation of XACML PDP implementations. *Proceedings of the 2008 ACM workshop on Secure Web Services*, pages 37–44, 2008.