Challenges of Composing XACML Policies

Bernard Stepien[1], Amy Felty[1], Stan Matwin[2, 3]
[1]School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada
[2]Faculty of Computer Science, Dalhousie University, Halifax, Canada
[3]Institute for Computer Science, Polish Academy of Sciences, Warsaw, Poland
(bernard | afelty)@eecs.uottawa.ca, stan@cs.dal.ca

***Abstract*— XACML (eXtensible Access Control Markup Language) is a declarative access control policy language that has unique language constructs for factoring out access control logic. These constructs make the specification of access control requirements more compact than decision trees, which can be considered the most natural way to specify access control logic. However, many publications report that performance of XACML policy decision point (PDP) engines is greatly affected by the structure of policy sets. In this paper we first explore the causes of potential inefficiencies of XACML policies, and then propose a procedure to re-structure policy sets vertically by modifying the distribution of access control logic among different configurations of structural elements, in order to remove much of this inefficiency. This is in contrast to horizontal re-ordering of constant structural elements. Our procedure can be applied regardless of the complexity and structure of the original policy set. We also compare the performance of policy sets that take advantage of the expressive power of XACML targets to decision trees.**

Keywords: access control; policy restructuring; XACML.

I. INTRODUCTION

While early access control (AC) policy specification languages were based on the principle of specifying simple combinations of attribute values to govern an effect (permit or deny), more recent languages started to incorporate policy language constructs that take advantage of the benefits resulting from allowing more complex combinations, such as the aggregation of common attribute values. This extra expressive power allows the policy writer to reduce the number of policies required in order to comply with a given set of AC requirements. It is a well-known fact that there is great benefit in reducing the number of policies because it makes the management of policies easier and also reduces the risk of errors. The XML based XACML language [1][2] has a number of such powerful constructs. While in general, an AC policy can be abstracted to a Boolean expression, XACML structures such Boolean expressions into hierarchical groupings using the concept of targets, which allow the specification of alternate conditions of combinations of attribute values. XACML targets are not pure Boolean expressions. Instead they structure logic along specific language constructs that represent only a subset of the capabilities of Boolean expressions. As a result, when structuring a given policy, the AC policy designer must make critical design choices that can impact performance of PDPs and management of the policies in general.

These features are very efficient for specifying the most recent AC models, especially those that are particularly efficient for expressing fine grained AC, such as the ABAC [3] and RBAC [4][5] models, as well as many other derived models.

In this paper, we first review XACML structuring mechanisms, exploring the factors that specifically govern such structuring, in order to motivate restructuring. We then explore the possibilities of improvements. In particular, we present a specific procedure for restructuring policies that has the potential to greatly increase the efficiency of policy evaluation.

II. STRUCTURAL MODELS FOR POLICY LOGIC SPECIFICATION

A. XACML Hierarchical Structuring Elements

XACML has two basic levels of constructs to express structuring:

- Hierarchical partitioning blocks: policy sets that contain other policy sets or policies that further contain rules.
- Access control logic blocks that contain fine-grained logic on attributes.

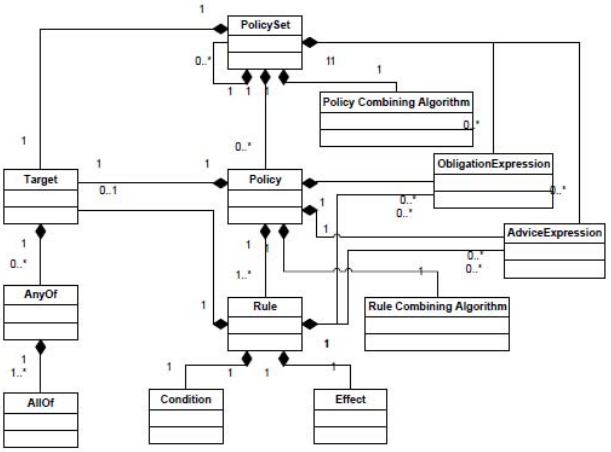

Figure 1: The XACML 3.0 policy set model

978-1-4799-4223-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ARES.2014.38

However, one of the characteristics of XACML AC logic blocks is that its grammar, as shown in Figure 1, allows only specific configurations of conjunctions and disjunctions for expressing logic in the form of targets and conditions. Targets are used as the primary logic description technique in all three levels of partitioning blocks: policy sets, policies and rules. Rule conditions have an additional level of logic expressed by pure Boolean expressions, which are used mostly for continuous domains such as numeric types including date and time.

1) Partition Blocks as Decision Trees

The partitioning block structural elements are organized as decision trees where alternate edges for a given node (a structural element) represent disjunctions only, while sequences of edges represent conjunctions.

Access control logic is expressed using an instance of the policy set model such as shown on Figure 2. This figure presents the hierarchy of the partitioning blocks. At this level, we have the structure of a decision tree. However, the evaluation of the logic is located inside these blocks which, instead of trees, contain graphs that each has a starting and a terminal node. A terminal node of one block is the starting node of a subsequent block. For example, the terminal node of a policy target is the starting node of each child rule targets.

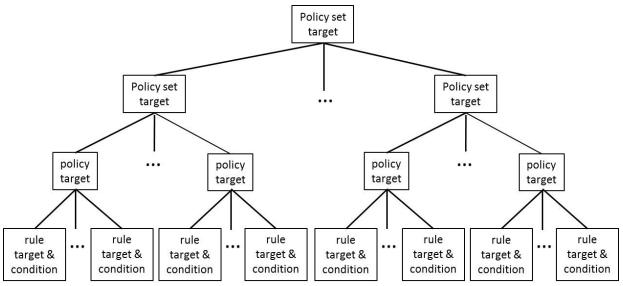

Figure 2: A XACML policy logic partitioning

2) XACML Parse Tree Oriented Target Logic

In XACML 3.0, access control logic is represented inside the target construct of each structural element (policy sets, policies and rules). The XACML target is expressed more like a parse tree where alternative edges represent both disjunction and conjunction, which is radically different from decision trees. Furthermore, there are limitations in expressive power due to the kind of construct that is allowed at each depth of the parse tree. For example the possible logic expressed in a target using *AnyOf* and *AllOf* language constructs results in specific logical patterns as shown on Figure 3.

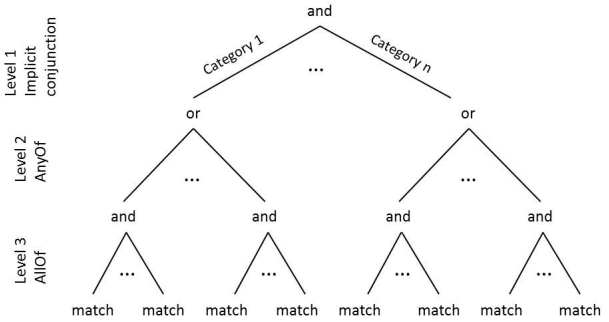

Figure 3: XACML 3.0 target parse tree Structure

In order to appraise the use of XACML targets in practice, it is interesting to instead represent target logic as a directional graph as shown on Figure 4. Such a graph can be used as a decision graph, i.e. a top-down walk through the nodes and edges. This graph has a very unique shape. In such a graph, as in a decision tree, a sequence of edges determines conjunction while alternate edges from a node represent disjunction. In the case of XACML targets, the *AnyOf* constructs are concatenated as a sequence, while the *AllOf* constructs form alternate edges from a given *AnyOf* node. Inside an *AllOf* node, the arguments are themselves represented as sequences of matches. The most important fact is that all the *AllOfs* of an *AnyOf* merge together to a common terminal node that itself is either the starting node of the next *AnyOf* construct, or the last node of the current target construct as shown on Figure 4.

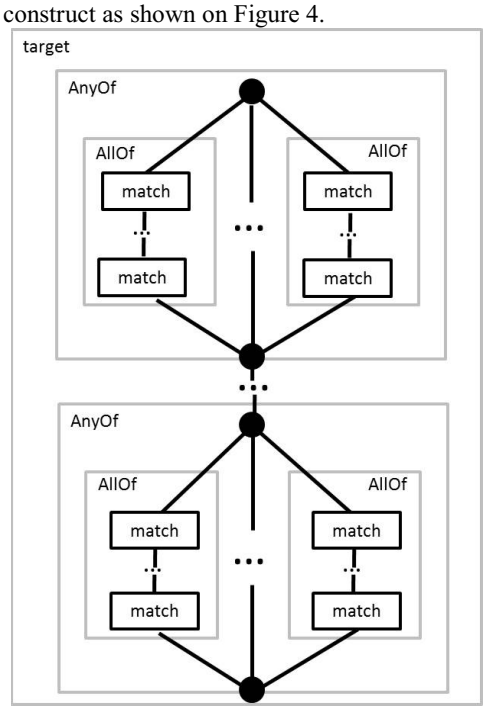

Figure 4: XACML 3.0 target represented as a graph

B. The Benefits of Structuring

Normally, the primary goal of the structuring mechanism of XACML policies is to provide a more efficient evaluation of requests by a Policy Decision Point (PDP). These evaluations are performed by a top down search of the decision tree as shown in Figure 2. A request is evaluated first against the top level policy set target. If its logic satisfies the request for the attributes it contains, it then explores its children's policies target logic. Then, once a policy target is satisfied, it further evaluates the targets and conditions of this policy's child rules. If an attribute is not used in the policy logic, this is considered as satisfying any value that this attribute can match. If any of these structural elements is not satisfied, it prevents the evaluation of elements that are further down in the hierarchy. A description of this process and its resulting savings in terms of computation costs is provided in [7].

While the structuring mechanisms in XACML are available, they do not prevent bad specifications, especially when policies evolve over time. This often occurs when modifications are performed by different engineers with different backgrounds both in experience and programming styles. The difficulties include the awareness of the existing policy logic implemented at different intervals of time, as well as and especially the knowledge of how to modify an existing policy so as to satisfy new needs.

While these structuring mechanisms exist and have already demonstrated their benefits, implementation inefficiencies are not always avoided. As a result, there has been extensive research on finding new ways to optimize policies so as to further reduce processing time and avoid PDP bottlenecks.

As has already been mentioned in [7], there are cases where the principle of hierarchical structures of decision trees does not avoid searching all the rules of a policy or at least some of its subtrees. Effectively, the efficiency of this process can depend on the distribution of logic expressed as match expressions on attributes among these structural elements. For example, let's consider an AC specification where we have structured the logic in three policies P_1, P_2 and P_3 and further in corresponding rules. The logic in our examples uses two attributes and their related matching values, i.e. a resource R $\{r_1, r_2, r_3\}$ and an action A $\{a_1, a_2, a_3\}$. Each policy has a different number of rules depending on the way we distribute the matches for each attribute among policy and rule targets. In the ABAC model and its implementation language XACML 3.0, the order of attributes in the parse tree representing a XACML target is not prescribed. Thus, several different policy writers could use different orders. This will naturally lead to redundancies for a specific effect, or conflicts in the case of the use of opposite effects (e.g., Permit/Deny). Here we use a single match in each partition level. Figure 5 shows two different structuring strategies.

- Policy set 1 has an inversion in attribute distribution between policies and rules for describing logic. For example policy P_1 and P_2 targets handle attribute R in the policy target and attribute A in the rule target while policy P_3 does the reverse.
- Policy set 2 follows the principle of using only one homogeneous kind of attribute in the respective levels of policy and rule target. Policy targets contain only attribute R expressions, while rule targets contain only attribute A expressions.

For these particular configurations, policy set 2 is more efficient since for example it requires only four comparisons to find the matches for attribute A value a_3 and attribute R value r_3 while it requires six comparisons to achieve the same result for policy set 1. In more detail, for policy set 1, the six targets of policies P_1, P_2, P_3 and rules R_3, R_4 and R_5 are evaluated while for policy set 2, only the four targets for policies P_1, P_2, P_3 and rule R_5 are evaluated.

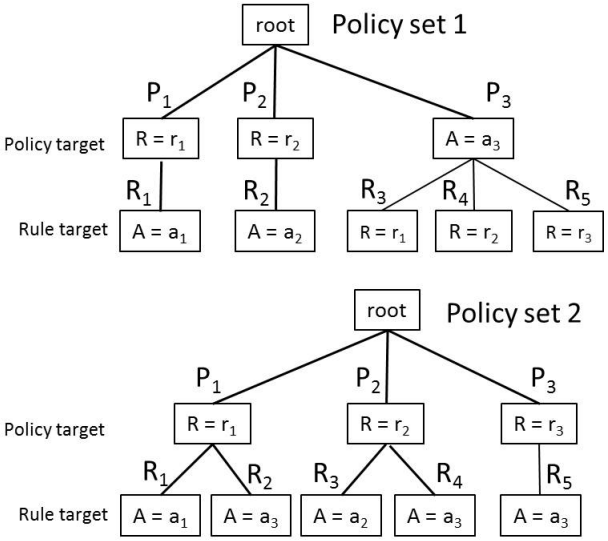

Figure 5: Equivalent policy set structures

Table 1 shows the PDP request processing costs in terms of comparisons for each combination of attribute values that is covered by the policy sets along with the total cost for evaluating requests for all combinations of values.

Requests/policy sets	PS1	PS2
R: r_1 and A: a_1	7	5
R: r_2 and A: a_2	7	5
R: r_1 and A: a_3	6	5
R: r_2 and A: a_3	6	5
R: r_3 and A: a_3	6	4
Total comparisons costs	**32**	**24**

Table 1: Request processing costs

In the above example, we have considered only the five requests that will return an effect of permit or deny. All remaining cases would return not applicable. Normally, with the cardinality of this example there are nine possible combinations of requests $|\{r_1, r_2, r_3\}| \times |\{a_1, a_2, a_3\}| = 9$. The total comparison costs shown in Table 1 suggest a plain average cost indicator where each request has the same probability of occurrence as others. Operational realities would be more along a weighed cost configuration. Effectively, resources are not used in an equal manner by the same subjects for the same actions. Some resources are used more than others and even changes of usage pattern can occur depending on external events as reported in [6].

C. *Expressing Access Control Requirements As Decision Trees*

Decision trees [12] are well-known for expressing access control logic [14][16]. They are in fact the most efficient representation from the point of view of request processing by a PDP, since a subtree will be explored only if its parent edge satisfies the request. Boolean expressions are decision trees. XACML rules have conditions that are Boolean expressions. Structuring capabilities of Boolean expressions have been explored in [10]. However, from an access control logic specification point of view decision trees require considerable redundancy of definitions. For example, given the alphabets for three attributes representing subject, resource and action $S = \{s_1, s_2\}$, $R = \{r_1, r_2, r_3\}$, $A = \{a_1, a_2\}$, we obtain one possible decision tree for the complete state space regardless of the effect (permit or deny) as shown on Figure 6.

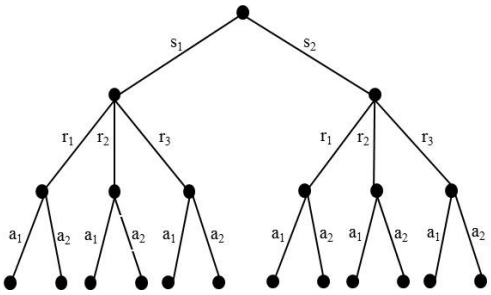

Figure 6: XACML policy set decision tree

Also, we can observe that Figure 6 shows what could be specification redundancies. Effectively, the subtrees of the edges s1 and s2 that consist of logic for attributes R and A are identical. In reality, this might not be the case since each leaf is associated with potentially different effects (permit or deny or not-applicable). It is the distribution of these effects that will determine which subtrees are either fully or partially redundant.

Also, decision trees can be structured differently in ordering the attributes among each level of the tree. For our above example, the three attributes can produce six possible different hierarchical structures for the decision tree representing their combinations as shown on Figure 7.

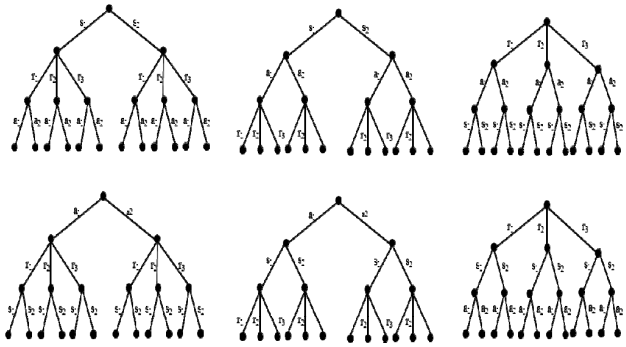

Figure 7: Decision tree equivalence

However, it is interesting to note that these equivalent variants of the same decision tree are also equivalent from a PDP request evaluation performance point of view. For example, the worst case represented by a request using the last value in the set of each attribute s_2, r_3 and a_2 requires exactly 7 match attempts before finding a full match satisfying all attributes. Thus, re-ordering full state space decision trees is of no value from a request evaluation performance point of view. However, we have determined that partial trees for a given effect could gain from re-ordering. However decision trees are considerably inefficient from an administration point of view because they produce redundant subtrees.

Finally, other methodologies, some graphical, can be used to design access control policy sets. [15] proposes a procedure based on business process models. [17] proposes a similar process as role mining in RBAC for ABAC.

III. THE ART OF RESTRUCTURING XACML POLICY SETS

A. *A Review of Existing Restructuring Algorithms and Procedures*

XACML restructuring algorithms can be classified into two broad categories:

- Horizontal re-ordering, consisting of optimizing the order of alternate children of policy sets (policy sets or policies) and/or optimizing the order of child rules of policies, as described in [6]. This also includes weighing requests according to statistical usage. In this approach, the overall vertical structure remains constant, i.e. the content of policy sets, policies and rules and their corresponding targets expressions remain constant.
- Vertical re-ordering, consisting of redistributing logic among hierarchical partitioning blocks including changing the configuration of these

blocks entirely [7] [9]. For example interchanging the logic contained in a rule target with one of its parent policies, or even further upstream targets of policy set parents. Consequently, vertical restructuring produces completely different contents of policy sets, policies or rules.

B. A Procedure for Vertical Restructuring

1) Limitations of the Subsumption Algorithm

In previous work, we have defined a subsumption algorithm for compressing policies [9]. This algorithm cannot solve the problem of restructuring the three policies of policy set 1 in Figure 5. This is because this subsumption algorithm works only in the case where Boolean expressions resulting from the sequences of elements in policy set, policy and rule targets have n-1 attribute expressions in common, where n is the total number of attributes used in such a Boolean expression. In fact, this algorithm works only for specific structures that consist of a conjunction, where each element is either an atomic operation or a disjunction of operations on the same attribute.

For example the two following Boolean expressions:

```
A1 == v1 /\ A2 == v2 /\ A3 == v3
A1 == v1 /\ A2 == v2 /\ A3 == v4
```

can be collapsed into the following single expression:

```
A1 == v1 /\ A2 == v2 /\ (A3 == v3 \/ A3 == v4)
```

In the example of Figure 5, the n - 1 common elements restriction is not satisfied since all operations have different attribute values as shown in Table 2.

Attribute	Policy 1	Policy 2	Policy 3
A	$A == a_1$	$A == a_2$	$A == a_3$
R	$R == r_1$	$R == r_2$	$R == r_1 \vee R == r_2 \vee R == r_3$

Table 2: Heterogeneity of attribute operations

However, we can show that policy set 1 can still be used to derive policy set 2 which is not compressed in the sense of [9]. In particular, it does not produce a compressed policy set that contains a reduced number of policies and/or rules, but its overall structure is more efficient. For example, in our case, policy set 1 and 2 contain exactly the same number of policies or rules but the access control logic is distributed differently among them. We have found that the subsumption algorithm can still be used by first deriving the traces from policy set 1 and applying the algorithm on the traces instead of the original policy set. These traces can have n - 1 common elements and thus be compressed into policies with more complex expression content:

Trace 1: $R == r_1 \ /\backslash\ A == a_1$
Trace 2: $R == r_2 \ /\backslash\ A == a_2$
Trace 3: $A == a_3 \ /\backslash\ R == r_1$
Trace 4: $A == a_3 \ /\backslash\ R == r_2$
Trace 5: $A == a_3 \ /\backslash\ R == r_3$

2) Using Decision Trees

Decision trees can be implemented in XACML using the recursive nature of the policy set language construct. However, they can be fastidious to compose due to the verbosity of XACML. Here, we will discuss the transformation of a decision tree into a more compact XACML structure that uses the aggregation capabilities of the XACML target. The purpose of this exercise is to determine which of the structures—decision trees or compact XACML policy sets—is the more efficient.

As we have mentioned above, a XACML policy set is a kind of decision tree and the restructuring process that uses the subsumption algorithm can be performed only on one subset of the decision tree at a time corresponding to a specific effect (permit or deny separately). For example in the three-attribute decision tree depicted in Figure 8, the black edges correspond to the effect permit and the grayed edges correspond either to the effect deny or to the effect not-applicable.

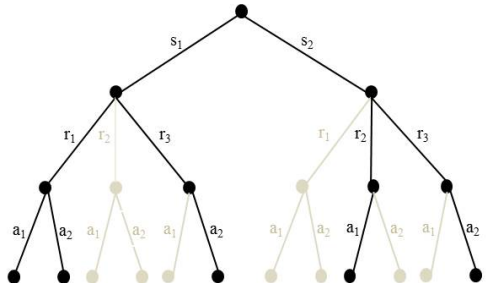

Figure 8: XACML decision subtree for effect permit

From that decision tree or from any corresponding XACML decision tree/graph hybrid, we can easily derive the five following traces merely by walking the tree once and collecting the traces of edges back to the root:

```
Trace 1: S == s1 /\ R == r1 /\ A == a1
Trace 2: S == s1 /\ R == r1 /\ A == a2
Trace 3: S == s1 /\ R == r3 /\ A == a2
Trace 4: S == s2 /\ R == r2 /\ A == a1
Trace 5: S == s2 /\ R == r3 /\ A == a2
```

These traces can also represent policies expressed in simple logic, consisting of a list of single conjunctions, similar to Access Control Lists (ACL) policies. This logic can itself be distributed among XACML structural components (policy set, policy and rule) producing equivalent logic. These five traces/policies can be compressed using the subsumption algorithm described in [9], giving one possible policy set shown as a hybrid decision tree in Figure 9.

These examples demonstrate the challenges facing the policy designer when composing XACML policy sets. A manual optimization could be complex. An illustration of these challenges can be derived from [7]. Effectively they propose an algorithm for computing performance of various randomly or manually restructured policy sets. They do not show any restructuring algorithm per se. Thus, the novelty of our approach is in the automation of this process.

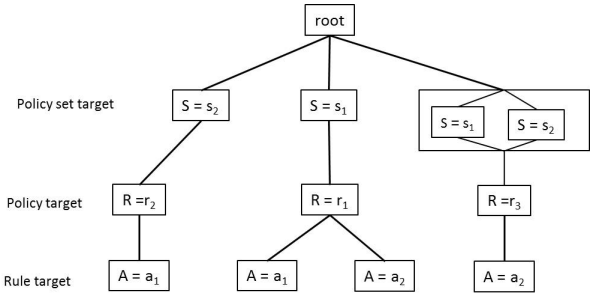

Figure 9: Policy set compression result (S-R-A)

Interesting is the fact that when this procedure is applied to policy set 1 in Figure 5, we indeed obtain policy set 2 which is the more efficient policy set configuration. Also, if we change the order of attributes in the decision tree of Figure 8 we obtain always the same identical tree shown in Figure 9, with the only difference being that the attributes change levels, with each level retaining the same structure (*AnyOfs*), like for example the tree of Figure 10. However, these trees produce different average number of computation costs.

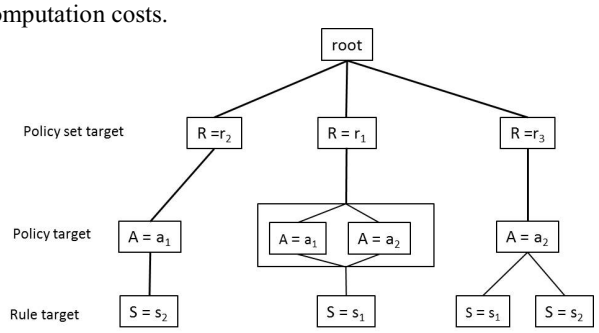

Figure 10: Compression result on resource, action, subject ordering (R-A-S)

In Table 3Table 3, we have summarized the computation costs for the original decision tree and two possible vertical orderings of attribute expressions. As expected, the decision tree has the best performance while the XACML hybrid decision/graph trees have equal costs. Normally one would expect that the XACML code restructuring produces code savings. This particular example does not. The duplication of expressions exists in both cases. However, this is mainly due to the nature of the example, the subset of the decision tree chosen and the limited number of attributes.

Requests	Decision tree	S-R-A ordering	R-A-S ordering
s_1-r_1-a_1	3	4	4
s_1-r_1-a_2	4	5	5
s_1-r_3-a_2	3	5	5
s_2-r_2-a_1	3	3	3
s_2-r_3-a_2	4	6	6
Total costs	17	23	23

Table 3: Cost comparison on order of attributes

Also, it is interesting to note that the resulting restructured policy set of Figure 9 shows a different kind of redundancy on the first level composed of policy sets, where the targets for attribute S show values s_1 and s_2 first separately and then combined into an *AnyOf*. This in fact is the result of the restrictive grammar of the XACML target language construct, and is not an oddity resulting from a faulty restructuring algorithm. Also, this does not mean that the XACML language is bad but instead shows the limitation of factoring out common behavior in general.

3) Enhancement of the Subsumption Algorithm

The results of the subsumption algorithm shown in Figure 9 are however suggesting a possible enhncement.of this algorithm. Effectively, the presence of duplicate operations on the same attribute s_1 and s_2 could be factored out by connecting the subtree $r_3 - a_2$ to both of the other subtrees starting with operations on attributes s_1 and s_2 individually rather than as a disjunction. The result of this additional step shown in Figure 11 is rather surprising because it produces the decision tree from which we started with in the first place with an insignificant difference in the horizontal ordering of the subtrees. However, this same result is not achievable by attempting to refine Figure 10 because there are no duplicate operations, neither on the first level nor any subsequent level. This, in itself is an interesting property that shows that the vertical ordering of attributes is in fact important. Thus, at first glance, the algorithm needs to be applied several times in order to find the more efficient configuration. However, the problem is that the number of orderings can be non-polynomial. This can be easily avoided altogether by instead inspecting each level of the tree individually to detect attribute operation duplications. When a level reveals such a duplication, the entire level can be pushed to the top where the factoring out can then be performed.

From a practical point of view, this exercise produced some interesting findings. It shows that any policy set structure can be transformed into a decision tree using the enhanced subsumption algorithm. Also, the policy set of Figure 10 could have been the starting point because this policy set could have been manually encoded by some policy engineer. In this case, generating traces from this policy set

and applying the enhanced subsumption algorithm on them would have resulted in the more performant decision tree.

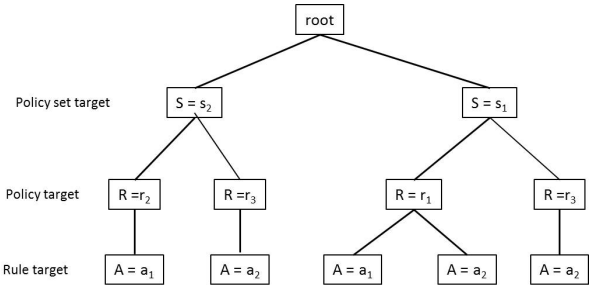

Figure 11: Enhanced compressed policy

4) Transforming Boolean Expressions into XACML Structural Elements

In figures 9, 10 and 11 we have suggested the XACML structural elements of policy sets, policies and rules for each level of the decision trees for the purpose of better understanding. This was for convenience purposes only. The subsumption algorithm for vertical restructuring does not produce this naturally. However, this structure can be easily derived. First of all, the bottom level always needs to be a rule. Thus, working upward, the level above the rule level naturally becomes a policy while any levels above the policy level all become policy sets because policy sets are the only recursive constructs in XACML. Also, it is important to note this procedure produces a flattening of Boolean expressions. Effectively, after generating the traces, the relationship between elements of the trace and XACML structural elements disappears. This means that it is no longer possible to distinguish that a conjunction is the result of a target AllOf element or the natural conjunction linking policy sets to their child policies or policies to their child rules.

IV. Summary of the Procedure for Restructuring XACML Policies

A. General Procedure Summary

In light of these findings, we propose the following procedure for performing vertical and horizontal restructuring of XACML policy sets.

- Step 1: transform a XACML policy set into a graph which is a hybrid between a decision tree and sub-graphs.
- Step 2: generate all the traces of that graph. The traces represent pure conjunctions between attributes match expressions and thus each trace can be used as a policy.
- Step 3: perform the subsumption policy compression algorithm [9] on these traces.
- Step 4: factor out duplicate operations on the same decision tree level.
- Step 5: perform horizontal restructuring [6] based on probabilities of requests.
- Step 6: compute the total cost of request processing of the compressed policies using the algorithm proposed in [7].
- Step 7: repeat step 2 to 5 using a different order in the sequence of attribute conditions of traces as a best-first search heuristic.
- Step 8: compare costs of the various orderings of attributes and choose the best one.

B. Handling Absent Attributes

In a XACML policy, when an attribute is not used in a match expression, this is equivalent to specifying all values that a given attribute can have, i.e. its full alphabet. This is indeed a very powerful feature that does not exist in many other AC languages. However, the absence of an attribute has very interesting benefits from a computation cost point of view since by definition it eliminates a number of computations altogether. Currently the policy compression algorithm from [9] cannot handle such cases. Solutions to include absent attributes are proposed in [16] where they are called missing attributes. However, the use of such an approach is for further study.

C. Proof of Equivalence

In previous work [9] on the subsumption algorithm, we have already demonstrated the benefits of using theorem proving techniques to provide correctness guarantees for our algorithm. In particular, in that work, we used the Coq Proof Assistant [19] to prove that a compressed policy set is equivalent to an original collection of simple policies. In this work, we can also provide such formal correctness guarantees, which in this case show that the results of restructuring are equivalent to the original complex policy sets. For example, we have verified the equivalence of the policy sets of Figure 5. XACML target match expressions can be easily proved equivalent with a single Coq command "tauto" that is able to automatically prove tautologies in propositional logic, which is our case here. For the interested reader, we give the full Coq script for this example. Both original and compressed policy sets are expressed as Boolean expressions as follows:

```
Section figures_5.

Inductive resources: Set := R1 | R2 | R3.
Inductive actions: Set := A1 | A2 | A3.

Variable R:resources.
Variable A:actions.

Definition PS1 := (R=R1 /\ A=A1)
       \/ (R=R2 /\ A=A2)
        \/ (A=A3 /\ (R=R1 \/ R=R2 \/ R=R3)) : Prop.

Definition PS2 := (R=R1 /\ (A=A1 \/ A=A3))
                  \/ (R=R2 /\ (A=A2 \/ A=A3))
```

```
\/ (R=R3 /\ A=A3): Prop.

Lemma fig5a :PS1   -> PS2.
Proof.
unfold PS1, PS2. tauto.
Qed.

Lemma fig5b : PS2   -> PS1.
Proof.
unfold PS1, PS2. tauto.
Qed.

Theorem fig5 : PS1 <-> PS2.

Proof.
split. exact fig5a. exact fig5b.
Qed.

End figures_5.
```

V. Future Work

The redundancies of specification caused by the XML hierarchical model of monolithic trees, both in XACML and decision trees could be avoided. Other tree-based specification languages in a variety of other fields different from access control use the concept of decision tree. This is the case, for example, in the Tree and Tabular Combined Notation (TTCN) [18] that is used for specifying tests in the domain of telecommunication protocols. TTCN uses the concept of collections of trees, where each individual tree can be attached to the leaves of any other tree. This technique allows unlimited re-usability of subtrees. This feature actually already exists in XML with the reference mechanism for schemas and there are already two similar concepts in XACML, namely, policy and policy set references and variables. However these references are used exclusively as an inheritance mechanism and apply to an entire policy, while XACML variables are snippets of Boolean expressions that can be used only in rule conditions and are not re-usable in the more restrictive XACML target. Thus, we propose to introduce the concept of independent sub-tree definitions and the attachment mechanism. Also, complex models such as the RBAC profile [11] may not be amenable to such a restructuring procedure, mainly because they are highly dependent on policy and rule combining algorithms, which operate on sets of intentional opposite effects (permit/deny).

VI. Conclusion

In this paper, we have examined some key factors in understanding problems of adequately structuring XACML policy sets and shown the limitations of re-structuring procedures when taken individually. We have proposed a procedure that combines these diverse procedures into a single procedure with the goal of increasing efficiency of the evaluation of requests against a policy.

Acknowledgements

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada.

References

[1] XACML 2.0 OASIS standard DOI= http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[2] XACML 3.0 OASIS standard DOI= http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf

[3] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, R. Miller, K. Scarfone, Guide to attribute based access control (ABAC) definition and considerations, in NIST Special Publication 800-162, January 2014, http://nvlpubs.nist.gov/nistpubs/-specialpublications/NIST.sp.800-162.pdf

[4] D. F. Ferraiolo, D. R. Kuhn, Role-based access control, in *Fifteenth National Computer Security Conference*, pp. 554-563, 1992.

[5] S. Verma, S. Kumar, M. Singh, Comparative Analysis of role base and attribute base access control model in semantic web, in *International Journal of Computer Applications*, vol. 46, no. 18, pp. 1-6, May 2012.

[6] S. Marouf, M. Shehab, A.Squicciarini, S. Sundareswaran, Adaptive reordering and clustering-based framework for efficient XACML policy evaluation, in *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 300-313, Oct.-Dec. 2011.

[7] P. L. Miseldine, Automated XACML policy reconfiguration for evaluation optimisation, in *Fourth International Workshop on Software Engineering for Secure Systems*, pp. 1-8, 2008.

[8] V. Kolovski, J. Hendler, B. Parsia, Formalizing XACML using defeasible description logics, in U. of Maryland technical report, 2007, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.-3914&rep=rep1&type=pdf .

[9] B.Stepien, S. Matwin, A. Felty, An algorithm for compression of XACML access control policy set by recursive subsumption, in *Seventh International Conference on Availability, Reliability, and Security*, pp. 161-167, 2012.

[10] B. Stepien, S. Matwin, A. Felty, Strategies for reducing risks of inconsistencies in access control policies, in *Fifth International Conference on Availability, Reliability, and Security*, pp. 140-147, 2010.

[11] OASIS, XACML 3.0 Core and Hierarchical Role Base Access Control (RBAC) Profile version 1.0, in http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.html

[12] K.-M. Leung, "Decision trees and decision rules," presentation slides, http://cis.poly.edu/~mleung/FRE7851/f07/decisionTrees.pdf, December 2007. Accessed: 2014-01-07.

[13] Y. Yuan and M. J. Shaw, Induction of fuzzy decision trees, in *Fuzzy Sets and systems*, vol. 69, no. 2, pp. 125–139, 1995.

[14] V. Zaliva, Firewall policy modelling, analysis and simulation: a survey, 2008, http://www.crocodile.org/lord/fwpolicy.pdf.

[15] C. Wolter, A. Schaad, C. Meinel, Deriving XACML policies from business process models, in *Web Information Systems Engineering—WISE 2007 Workshops*, pp 142-153, 2007.

[16] R. A. Shaik, K. Adi, L. Logrippo, S. Mankovski, Inconsistency detection method for access control policies in *Sixth International Conference on Information Assurance and Security*, pp. 204-209, 2010.

[17] L. Krautsevitch, A. Lazouski, F. Martinelli, A. Yautsiukhin, Towards policy engineering for attribute-based access control, in *Trusted Systems,* Springer Verlag Lecture Notes in Computer Science, vol. 8292, pp 85-102, 2013.

[18] ITU X.292 standard, Tree and Tabular Combined Notation (TTCN).

[19] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development, Springer Verlag, 2004.