2012 Seventh International Conference on Availability, Reliability and Security

An Algorithm for Compression of XACML Access Control Policy Sets by Recursive Subsumption

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

***Abstract*— Policy administrators increasingly face the challenge of managing large policy bases, and this need becomes more acute with the growing importance of fine-grained access control models, e.g. ABAC. We have shown in previous work that simple policies mostly based on conjunctions of single attribute conditions, can be merged into more complex conditions composed of combinations of conjunctions and disjunctions of attribute/value pairs. Here, we propose an algorithm that uses a recursive process of subsumption applied on the original set of policies that results in a complex and short policy, often significantly compressing the original policy. We present this algorithm, and discuss the advantages of this approach, i.e. its performance when working on the policy structures encountered in real-life policy sets, its scalability, and its ability to deal with large alphabet sets.

Keywords: access control, subsumption algorithm, XACML.

I. INTRODUCTION

Access control (AC) policy specification languages have gone through a long evolution over time [3]. More recently, the increasingly important requirement of interoperability has led to standardization efforts that produced the XML-based XACML access control policy specification language [14]. In addition to the benefits of standardization, this language has the additional advantage that it allows the specification of complex conditions, which are ideal for fine grained AC systems. However, a combination of two factors has resulted in very little use of these powerful capabilities. The first factor is related to legacy. Administrators have to migrate legacy AC systems that don't allow complex conditions.

They usually do a straightforward translation that perpetuates a specification style of using only simple logical expressions. Such expressions are usually simple conjunctions of individual conditions on given attributes. The second factor ironically resides in the XML nature of XACML itself. While XACML is an ideal machine readable language, its extreme verbosity, due to the combination of long XML tags and long domain names for operators, makes the use of complex conditions difficult for a human to read, prone to errors during editing and thus impractical. In [15] we have shown that the verbosity of XACML can be eliminated easily using a non-technical notation combined with an attribute data model for the purpose of displaying and editing of XACML policies. Because of its coupling with an attribute data model, the transformation into the non-technical notation preserves the semantics of XACML. The notation is not a language in itself; it is merely a simplification of the XACML language that retains the overall XACML structure. We use this notation throughout this paper for examples. Once the verbosity of XACML is eliminated, it is possible to take advantage of the benefits of using complex conditions, which we have shown in [16,17]. These benefits arise mostly from the ability to make policy conditions more understandable and thus more manageable.

II. BACKGROUND

A. *Reasons for Large Policy Sets*

It is a well-known fact that AC policy sets are traditionally large for two reasons: first of all, in early AC systems, permissions to access resources were assigned directly to individuals. Later, new AC models eliminated the direct assignment of permissions to individuals. Instead, permissions were assigned to various attributes such as the roles an individual plays, as in the RBAC model [7] or even to an unlimited number of attributes as in the ABAC [8] model. These models allow for fine grained AC, which is the central feature of the XACML language itself. This is a major improvement that has resulted in a reduction of the number of policies required to specify the AC requirements of an organization. However, there still remain reasons for having large numbers of policies: one of them is the number of attributes, especially in fine grained AC systems. In [10], it has been established that for RBAC models, the size of a policy set can be potentially 2^n combinations of n roles, while ABAC has a potential number of 2^n combinations of rules for n attributes. However, the prime reason for large numbers of rules resides mostly in specification styles. For example, in XACML, the target part of a rule or policy uses mostly simple logical expressions. In [16] we showed that it is possible to reduce considerably the number of policies by instead using complex conditions. Our findings were the result of an exercise in scalability where policies were generated systematically on sets of alphabets for attributes. The advantages of reducing the number of policies are twofold:

- Reducing the risk of conflicts;
- Improving performance of access granting tools.

B. *State of Research in Reducing Policy Sets*

Both of the advantages above have been the focus of extensive research. A summary of research in conflict

978-0-7695-4775-6/12 $26.00 © 2012 IEEE
DOI 10.1109/ARES.2012.38

detection is given in [16]. The topic of improving performance has been the subject of more recent research [11,12,13].

In [11], an approach to reducing the number of policies by eliminating policies that are in conflict is proposed. The elimination of conflicts results in a reduced number of policies since one of the two conflicting policies must be eliminated. This paper also describes an important policy writing style that separates policies into two groups: safety policies that consist of specifying what users should not have access to and utility policies that consist of specifying what users have access to (a generalization of availability policies). However the authors point out that these two stylistic groups are naturally conflicting. More important is the fact that they consider these types of policies difficult to resolve at run time. Thus there is still a need to find a method for detecting these conflicts at compile time. They use static pruning and minimal inconsistency cover set.

In [12], the emphasis is not on reduction of the size of the policy sets, but instead on re-ordering the policies to improve the search time required to evaluate AC requests and grant or deny permission to access a resource. The originality of this work is that the reordering algorithm is based on a statistical analysis not only of the policies of a policy set but also on the dynamic flow of requests that can structurally vary with time due to seasonal factors or market conditions.

In [13], the hierarchical structuring of XACML policy elements (policy sets, policies, rules, targets and conditions) is used for the purpose of optimization and improved performance. Their approach consists of shifting the location of these elements along the hierarchy of a policy in order to minimize the number of comparisons between attribute values of a request and a policy to achieve a reduction in evaluation costs. Experiments show that a 60% reduction in evaluation costs can be achieved. Again this is not by reducing the number of policies but merely restructuring them.

All of the above research shows that there is a widespread awareness that there is a performance problem with XACML policy sets. In summary, most of the existing research focuses on specific application domains [9] or on reasoning about policy conditions [6].

C. Distribution of Logic in XACML

XACML policy logic is described in several structural levels that already make some effort to reduce the amount of computation when evaluating AC requests. It separates policies and rules, each of them having logic located in a target, and the rule having a separate condition. In order to focus on the logic instead of computational efficiency, and to counter the scattering of logic that makes understanding and testing of policies difficult, we collapse all these levels into a single logical expression. This is possible because XACML's structural breakdown amounts in fact to an implicit conjunction. The approach of representing a XACML policy using a single condition has already been explored although not published. It is described in a U.S. patent [1]. Note that by collapsing rules and policies in this way, there is no longer any distinction between a rule and a policy.

D. Performance Measurement

In [17] we have proposed a metric to measure the impact of policy specification styles that compared two opposing styles: the use of simple conjunctions of attribute conditions and the use of complex conditions using combinations of conjunctions and disjunctions of attribute conditions. If a policy set has n attributes $a_{1, \ldots,}\ a_n$ with corresponding n_{a1}, …, n_{an} possible values v for each attribute (size of alphabet), the number of policies np required to cover the entire permission space is the number of combinations between attributes and their values:

$$np = n_{a1} \times n_{a2} \times \ldots \times n_{an}$$

If we are performing an evaluation request by linearly traversing a policy set in this representation, the worst case number of comparisons nc, also called request evaluation cost, between the values of an AC request and the values of policy attributes is the product of the sizes of each of the attribute alphabets (number of combinations np above) multiplied by the number of attributes itself:

$$nc = (n_{a1} \times n_{a2} \times \ldots \times n_{an}) \times n$$

This is due to the fact that each attribute must be re-evaluated for each policy until one matches.

Instead, we found that a policy set that has the same coverage can be represented by a single policy that is composed of a conjunction between conditions for each attribute. These conditions correspond to a disjunction between the values of the alphabet for each attribute. This single policy, let us call it a generalized policy GP, is a *subsumption* of the original set of policies $P_1, \ldots, P_n$, in that there exists, for each combination of specific values of attributes of each P_i, an assignment of values of attributes in GP such that for this assignment of values $GP = P_i$.. When a policy P subsumes a policy Q, P is in fact a generalization of Q. For example, if P gives specific values to attributes $a_1, \ldots, a_n$, then GP is:

```
   $a_1$ is one of  $v_{a11}$, …, $v_{a1n1}$
and
   …
and
   $a_n$ is one of  $v_{an1}$, …, $v_{annm}$
```

We have previously determined that the worst case number of comparisons between the attribute values of a request against the policy required to reach a match is additive in a generalized policy rather than multiplicative as in the first case. In this case it is the sum of the individual sizes of each attributes alphabets:

$$ncp_s = n_{a1} + n_{a2} + ... + n_{an}$$

While so far we had determined that it is possible to reduce the size of policy bases and that there is a clear advantage to it, we had not yet determined a procedure to systematically achieve this goal. This paper describes an algorithm to reduce policy sets based on simple policies (conjunctions of conditions on individual attributes) and discusses the benefits of such an approach for realistic AC policy sets. Our contribution is therefore the algorithm for policy compression of policy sets described in this paper. This algorithm has the capability to substantially reduce the size of policy sets by merging (by subsumption) individual policy condition logic into single conditions, thus making them both more manageable and allowing a better performance for PDPs.

III. Policy compression algorithm

In [17] we used a concrete small example of 18 policies that were automatically generated using the combinations of values of 3 attributes X, Y, Z that each have alphabets of {10, 20, 30}, {"a", "b", "c"}, {true, false}, respectively, as follows:

```
P1: X is 10 ∧ Y is "a" ∧ Z is true
P2: X is 10 ∧ Y is "a" ∧ Z is false
P3: X is 10 ∧ Y is "b" ∧ Z is true
P4: X is 10 ∧ Y is "b" ∧ Z is false
P5: X is 10 ∧ Y is "c" ∧ Z is true
P6: X is 10 ∧ Y is "c" ∧ Z is false
P7: X is 20 ∧ Y is "a" ∧ Z is true
P8: X is 20 ∧ Y is "a" ∧ Z is false
P9: X is 20 ∧ Y is "b" ∧ Z is true
P10: X is 20 ∧ Y is "b" ∧ Z is false
P11: X is 20 ∧ Y is "c" ∧ Z is true
P12: X is 20 ∧ Y is "c" ∧ Z is false
P13: X is 30 ∧ Y is "a" ∧ Z is true
P14: X is 30 ∧ Y is "a" ∧ Z is false
P15: X is 30 ∧ Y is "b" ∧ Z is true
P16: X is 30 ∧ Y is "b" ∧ Z is false
P17: X is 30 ∧ Y is "c" ∧ Z is true
P18: X is 30 ∧ Y is "c" ∧ Z is false
```

Note that in this example, there is a policy for every combination of values. Furthermore, we assume that all of these policies are assigned the same affect (accept or deny). We start with this simple example for illustration purposes, so that we can show the steps our algorithm takes to collapse all 18 policies to a single one. We will then generalize to more realistic policies. As the above policy set is "full", i.e. it represents all the possible values of all attributes, it can—as we will show below—be reduced by recursive application of subsumptions to a single policy *Ps* using a shallow complex condition as follows:

```
   X is one of 10, 20, 30
and
   Y is one of "a", "b", "c"
and
   Z is one of true, false
```

In this example, the cost of processing a request for the worst case scenario would be equal to 3×3×2×3 = 54 comparisons for the original set of 18 policies while the cost for the single policy would be only 3+3+2 = 8 comparisons. A realistic example would have considerably more attributes and larger alphabets and would show even more dramatic differences in the computation cost. For example if in this example with three attributes we, increase the sizes of the alphabets to {5, 10, 8}, we would have 5×10×8×3 = 1200 comparisons rather than 5+10+8 = 23. This is a ratio of 52 instead of a ratio of 6.5 for the number of comparisons.

Reducing the number of policies in a policy set has always been a goal among policy administrators. However, the most common technique to achieve this compression consists of using "blanket" policies that specify the same effect against exception policies that specify the opposite effect. For example, a rule that permits access for a few values of a given attribute might be placed before a blanket rule that denies access for all values. The main problem with this approach is that it creates a *de facto* intentional conflict that is difficult to distinguish from conflicts resulting from mere errors. The problem is aggravated in the case where there are several attributes, as inevitably the policy administrator will have to juggle with combinatorics, with the potential for further increasing the risk of conflicts.

A. *Recursive Subsumption Algorithm*

The algorithm consists of comparing each attribute of a pair of policies. In [16], we had already shown that it is possible to subsume a pair of simple policies, transforming them into a single synthesized policy in the case when all attributes have the same values except one. The values of the attribute that are different can be expressed as a disjunction in the synthesized policy. This is the case for example of the two first policies P_1 and P_2:

```
P1: X is 10 ∧ Y is "a" ∧ Z is true
P2: X is 10 ∧ Y is "a" ∧ Z is false
```

that can be subsumed into a single policy Pm_1 where attribute Z is now a disjunction:

Policy Pm_1 condition:

```
   X is 10
and
   Y is "a"
and
   Z is one of true, false
```

The same principle can be applied to other pairs of policies such as P_3 and P_4 and all other pairs in the sequence. This is of course because for all of these pairs the conditions of attributes X and Y are always identical. Thus, if we apply this algorithm to all the policies in the above policy base example, we will end up with nine policies $Pm_1..Pm_9$ subsuming the original 18 policies $P_1,…,P_{18}$. This corresponds to the total number of simple policies divided by two.

The second and any subsequent step in this algorithm consists of attempting to further subsume these nine intermediary policies. For example in the first pass, policies P_3 and P_4 can also be subsumed into policy Pm_2 as follows:

Policy Pm_2 condition:

```
   X is 10
and
   Y is "b"
and
   Z is one of true, false
```

We can easily observe that subsumed policies Pm_1 and Pm_2 have now attributes X and Z that are common but attribute Y that is different. Thus we can subsume Pm_1 and Pm_2 into policy Pm_{12} by creating a disjunction for attribute Y as follows:

Policy Pm_{12} condition:

```
   X is 10
and
   Y is one of "a", "b"
and
   Z is one of true, false
```

This procedure can be repeated for all pairs of subsumed policies of iteration 1 and all subsequent merged policies in the subsequent iterations resulting from increasingly complex subsumed policies until we reach the single policy Ps. In this algorithm, two policies whether simple or complex must have first, an identical effect (permit/deny) and second, n – 1 attributes in common in order to be merged. If this first condition is satisfied, the non-common attribute values can be subsumed into a disjunction regardless of the number of elements they contain. The various iterations required to merge the above policies are summarized in Figure 1 where policies were compared from left to right. Figure 1 shows the tree of values for each attribute (X, Y, Z) that are implicitly linked via conjunctions.

The set of policies shown in Figure 1 have all the same effect, either permit or deny. This allows us to show that the result of this algorithm is a single policy. This is an exceptional situation, due to the fact that the set of 18 policies $P_1,…,P_{18}$ represents exhaustively all the combinations of the values of the three attributes, and therefore it can be compressed by building a perfect binary-ternary tree shown in Figure 1. More realistic cases are addressed in subsequent sections.

We have implemented this algorithm in Prolog, which has already been successfully used in access control verification [2].

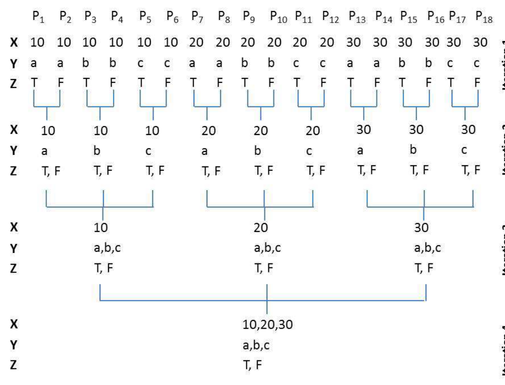
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18
X 10 10 10 10 10 10 20 20 20 20 20 20 30 30 30 30 30 30
Y a a b b c c a a b b c c a a b b c c
Z T F T F T F T F T F T F T F T F T F
Iteration 1
X 10 10 10 20 20 20 30 30 30
Y a b c a b c a b c
Z T, F T, F T, F T, F T, F T, F T, F T, F T, F
Iteration 2
X 10 20 30
Y a,b,c a,b,c a,b,c
Z T, F T, F T, F
Iteration 3
X 10,20,30
Y a,b,c
Z T, F
Iteration 4

Figure 1: Policy merging iterations summary

B. *Alternate Policy Comparison Strategies*

Since the above 18 policies $P_1,…,P_{18}$ were generated automatically according to a simple combinatorial algorithm, they are naturally sorted according to the value of their attributes from left to right. In this order, the policy compression process requires 21 comparisons between either original or subsumed policies. These comparisons consist of 9 comparisons between the original simple policies where subsumptions occur immediately after the first comparison and 12 comparisons between various degrees of generalization of already subsumed policies. The obtained policies require several comparisons before finding a match. For example, Policy Pm_3 (subsumption of P_5 and P_6), can only be matched with policy Pm_6 (subsumption of P_{11} and P_{12}), thus failing to match with policies Pm_4 and Pm_5 (not shown here but resulting from the subsumption of original adjacent simple policies) in the list of generalized policies. We have observed that the maximum number of comparison attempts was 3 with also a fair amount of cases succeeding after 1 or 2 attempts. In this experiment we have attempted several other approaches in order to optimize the number of comparisons. One of them consisted of comparing an original simple policy in priority to the list of already generalized policies before comparing it to another simple policy. This approach resulted in a large increase in the number of comparisons because the criteria to have n-1 attributes with equal values can be achieved only for a subset of policies. In fact, a simple policy can never be subsumed by an already generalized policy because there will always be more than one attribute value that is different. For example, if we attempted to merge a simple policy P_5 with a generalized policy $Pm_{12,}$ we would at least temporarily create an error:

```
P5: X is 10 ∧ Y is "c" ∧ Z is true
```

against the generalized merged policy Pm_{12}:

```
   X is 10
and
   Y is one of "a", "b"
and
   Z is one of true, false
```

Policy P_5 has only a common value of 10 for attribute X, an entirely different value "c" for attribute Y but a value of true for attribute Z that is included in the corresponding attribute set of values (implicit disjunction) of another generalized policy Pm_{12}. This is due to the fact that the original policies and the result of their merging are not equivalent in the sense that the original policies are satisfied by the merged policy, but the reverse is not true as the merged policy has combinations of values that cannot be satisfied by the original policies. We have actually verified this fact using theorem proving.

C. *Performance and Policy Base Order*

The experimental policy set being naturally sorted as a result of automated generation is highly unrealistic when compared to what will be encountered in real-life access control policies where usually policies are implemented randomly according to the unpredictable movement of personnel and their assignments to departments, work teams, etc. In order to simulate this natural chaos we have experimented by unsorting the policies, making their order more random. Random orders resulted in a wide variety of comparison costs represented by the number of comparisons to achieve the full single policy merge. Thus, sorting policies by attribute values increases performance. This is similar to the findings made in [12] although not for the same purpose. However, since our policies are simple, an ordinary sorting algorithm would be sufficient since those are log-linear or even linear (radix sort can be used for the kind of structured format data we are dealing with in policies).

IV. USING THE ALGORITHM WITH REALISTIC CASES

Our experimental example has one additional unrealistic feature. It is the result of the combinations of values of the exhaustive value sets (alphabets) of each attribute. This example is unrealistic, first of all because it will allow everyone to be either exclusively granted or exclusively denied access to resources, which is usually not the primary goal of access control policies. For one thing, since there can be only two antagonistic effects, permit or deny, a share of those combinations will permit access while the complement will deny access. It is a well-known fact that policy administrator use two strategies to deny access:

- Selective denial of access
- Default denial of access

A. *Selective Denial of Access*

The selective denial of access approach will result in a policy base that will consist of exactly the number of combinations of all values of attribute alphabets, a portion of it being permits and the complement being denies. However, from a subsumption algorithm point of view, this is actually favourable, resulting in an efficient generalized result. Each of these two portions will result in efficient compression by subsumption. Sometimes the result is a compression down to only two policies, one for permit and one for deny. The reality, however, is that permits and denies will be encountered in uneven distributions. In order to measure the resulting reduction for various proportions of effects, we have conducted further experiments by changing the effect permit or deny of some policies. In our case, we have determined that the worst result consists of a 50% reduction in the number of policies for each group of effects.

B. *Default Denial of Access*

The second policy implementation style consists of using only one of the two effects in the specification of individual policies and using a default catch-all policy (a "blanket policy") for the other effect. Handling this case is relatively simple and consists of handling only the specific effect simple policies and ignoring the default policy. To simulate this approach, we have removed a number of policies from our experimental generated policy set. Two techniques were used for this simulation:

- Removing policies with identical attribute values;
- Removing policies with non-identical attribute values.

For example, with the first technique we could remove all policies for which the value of attribute Z is false. Not surprisingly, this case results in a single generalized policy merely because this action resulted in reducing the alphabet of attribute Z to one element instead of two.

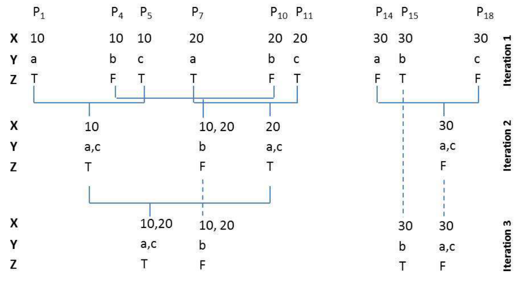

Figure 2: incomplete alphabet iterations

The second case can be simulated by alternatively removing policies for the opposite value of attribute Z as shown in Figure 2. For example, for the first four policies of our experimental policy set, we can achieve this pattern by removing policy P_2 and P_3. Now the remaining policies P_1 and P_4 will no longer satisfy the condition of having n – 1 attributes with common conditions. Instead, we now have only value 10 for attribute X in common as follows:

```
P₁: X is 10 ∧ Y is "a" ∧ Z is true
P₄: X is 10 ∧ Y is "c" ∧ Z is false
```

Thus, P_1 and P_4, now adjacent, cannot be merged along this principle (n-1 common attributes conditions) and will remain potentially single separate policies. The stable and irreducible set of policies is reached after only three iterations and one policy, P_{15}, could not be generalized at all. However, the results show only four policies instead of the initial nine which represents a reduction of 55%.

However, policies P_1 and P_4 could have been subsumed immediately using deeper conditions such as:

```
   X is 10
and
   Y is one of
      a provided Z is true
      b provided Z is false.
```

Further subsuming such complex conditions cannot be handled using the n – 1 common attribute principle of our algorithm and could potentially be achieved only at greater costs. This approach is further work.

C. *Attributes with Large Alphabets*

Discussions with industry have raised an interesting case where one or several attributes have a large alphabet while the remaining attributes have small alphabets. It was believed that our algorithm would not be useful in this case. This is, among others, the case of controlled telephones where users are restricted in the usage of such telephones, for example for long distance calls by area code or even at the exchange level, international, etc. In this case, one of the attributes is inevitably the phone number which by definition can be large for large organizations because it is at least equal to the number of employees which can be typically in the tens of thousands range. Our algorithm still works in this special case for the following reason: policies are generalized according to the attributes with smaller alphabets. For example, if we take our experimental generated example of figure 1 and we change the alphabet of attribute X to have 50,000 different values, the result of this algorithm will again be a single policy of the form:

```
      X is one of v1 .. v50000
   and
      Y is one of "a", "b", "c"
   and
      Z is one of true, false
```

This is due to the natural fact that each value of the attribute of the smaller alphabet will apply to large chunks of the attribute with the very large alphabet.

V. Merging complex and deeper conditions policies

The XACML language allows the specification of complex conditions with any combination of conjunction and disjunction operations at any depth. These may have sub-constraints attached to attribute values conditions. For example a policy that specifies the condition of access to medical documents where the function may restrict access depending on the day of the week.

Policy Pcx condition:

```
   PatientConsent is true
or
   Emergency is true
and
   Actor is one of
      Doctor,
      Nurse
         provided that DayOfTheWeek is
                  one of Saturday, Sunday
```

We now would like to merge the following policy *Prd* with the complex policy *Pcx* above.

Policy *Prd* condition:

```
   Emergency is true
and
   Actor is Radiologist
      provided that Location is hospital
```

A close inspection of these two policies shows that they cannot be merged. The reason is in the disjunction between the Patient consent and emergency attributes in policy *Pcx*. The disjunction for these two attributes would allow, in error, the radiologist to access the medical documents when the patient consent is true which is not specified in policy *Prd* that we are attempting to merge. This example illustrates that deep complex conditions can be subsumed as long as their complex conditions for each attribute are common.

VI. Scalability

One significant advantage of this algorithm is that it does not need all policies of a policy set to be in memory at once. Each pair can be retrieved from some policy data base individually, compared, eventually merged into a single policy and removed from the data base. The merged policy can be stored into the data base as well. However the number of retrievals to achieve a merging depends, as we have shown in Section 4, on the order of the attribute values. This is usually resolved naturally using data base indexing. We have run our algorithm on large policy sets of 5000 policies that produced a single merged policy within 1000 ms.

VII. Correctness of the compression

Once the transformation has been performed, it is necessary to demonstrate the equivalence of the two policy sets. The algorithm relies on a simple Boolean algebra transformation of a formula to an equivalent one. Policies in a policy set or rules in a XACML policy are mostly considered as lists, as the XML schema suggests. However, from a policy decision point of view, individual policies or rules are linked via implicit disjunction. For example, if a policy has the condition: $(C1 \wedge C2 \wedge C3)$ and another policy has the condition: $(C1 \wedge C2 \wedge C4)$, the disjunction can be constructed as follows:

$$(C1 \wedge C2 \wedge C3) \vee (C1 \wedge C2 \wedge C4)$$

This can be rewritten, by using distributivity of the disjunction operator, as:

$$(C1 \vee C1) \wedge (C2 \vee C2) \wedge (C3 \vee C4)$$

Thus, the two first terms above can use Boolean truth tables to be reduced as follows:

`(C1 ∨ C1) = C1` and the same for `(C2 ∨ C2) = C2`. Consequently, after this rewrite and reduction has been performed we obtain the desired merged conditions:

```
C1 ∧ C2 ∧ (C3 ∨ C4)
```

In addition to the above reasoning, there are also three additional approaches:

- Testing
- Redundancy check
- Theorem proving

The first approach uses the fact that the original policy set is composed only of simple policies, consisting only of conjunctions of conditions on attributes that operate on a single value of each attribute, as in our generated experimental example. Each of these rules can be viewed as a request. Thus, each policy condition of the original policy set can be sent to a Policy Decision Point (PDP) program that evaluates the policies in the subsumed policy set to verify that it is satisfied.

The second approach applies to cases where the original policy set already contains various degrees of complexities within a policy condition, usually meaning a mix of conjunctions and disjunctions of attribute conditions. This case is closer to real-life systems. In this case, the policies can no longer be used as requests and thus, the first testing approach is no longer usable. We have already shown in [16] that conflict detection algorithms using constraint logic programming can be applied in order to verify that a policy's attributes verify also the condition of another policy. This is the case of a redundancy check.

Finally, we have verified the equivalence between the set of original policies and the resulting policies of the subsumption using the tautology checker in the Coq theorem prover [4]. This verification has also been successfully performed for each intermediary result shown in figure 1 and 2. Coq was also used in our work on verifying a conflict detection algorithm for firewalls [5].

VIII. Conclusion

We have presented an algorithm for reducing the size of access control policies in XACML. As we have argued, reduced policy sets decrease the risk of conflicts and improve PDP performance. We have fully illustrated our algorithm on the best case scenario which reduces a policy set to a single policy, and discussed a variety of other realistic cases in which our experiments showed that we generally can achieve 50% reduction of a policy set size. Future work includes further improving the algorithm for more complex policies and proving its correctness.

Acknowledgements

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada and of Devera Logic Inc. for this research.

References

[1] A. Anderson, S. Proctor, Method for analysing an XACML policy, U.S. patent 20100042973.

[2] S. Barker, P. J. Stuckey, Flexible access control policy specification with constraint logic programming, in *ACM Transactions on Information and System Security*, 6(4):501-546, 2003.

[3] R.Batouba and I. Aib, "Policy-based management: A historical perspective," IEEE Transactions on Network and Service Management, Vol. 4, 2007.

[4] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development, , ISBN 3-540-20854-2, Springer Verlag

[5] V. Capretta, , B. Stepien, A. Felty, S. Matwin, Formal correctness of conflict detection for firewalls, in FMSE'07 proceedings p 22-30.

[6] D. Dougherty, K. Fisler, S. Krishnamurti, Specifying and resaonning about dynamic access-control policies, in IJCAR 2006 proceedings, pages 632-646

[7] D. F. Ferraiolo, D. R. Kuhn, Role-based access control, in *Proc. of the 15th National Computer Security Conference*, pages 554-563,1992.

[8] A. Karp, H. Haury, M.H. Davis, From ABAC to ZBAC: the evolution of access aontrol models, Technical Report HPL-2009-30, http://www.hpl.hp.com/techreports/2009/HPL-2009-30.pdf, 2009.

[9] V Kolovski, J. Hendler, B. Parsia, Analyzing web access control policies, in WWW 2007 proceedings, pages 677-686

[10] R. Kuhn, E. J. Coyne, T. R. Weil, Adding attributes to role-based access control, in *IEEE Computer*, 43(6):79-81, 2010.

[11] J. Lu, R. Li, J. Hu, D. Xu, Inconsistency resolving of safety and utility in access control, in EURASIP journal on Wireless Communications and Networking, 2011.

[12] 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, Oct.-Dec. 2011, pages 300-313

[13] P.L.Miseldine, Automated XACML policy reconfiguration for evaluation optimisation, in proceedings SESS'08, pages 1-8

[14] OASIS, eXtensible Access Control Markup Language (XACML). [Online]. Available: http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf.

[15] B. Stepien, A. Felty, and S. Matwin, "A non-technical user-oriented display notation for XACML conditions," E-Technologies: Innovation in an Open World, Proc. of the 4th International MCeTech Conference, Springer, 2009,

[16] B. Stepien, A. Felty, S. Matwin, Strategies for reducing risks of inconsistencies in access control policies, in ARES 2010 proceedings.

[17] B. Stepien, A. Felty, S. Matwin, Advantages of a non-technical XACML notation in role-based models, in PST 2011 proceedings.