Development of a new policy evaluation procedure for XACML

JORIAN VAN OOSTENBRUGGE

System and Network Engineering, Universiteit van Amsterdam
Email: jorian.vanoostenbrugge@os3.nl

In specifying access control policies eXtensible Access Control Markup Language (XACML) has become the de facto standard. In XACML, when a request is made to access a protected resource, the access decision as returned by the responsible component is obtained by parsing the XACML policy. As the policies can grow very large, simply reading the policy in a top-down manner has a performance penalty which make the whole system act unresponsive. In parsing XACML policies alternative solutions have been developed which improve the speed by which access decisions are generated. However, the existing solutions employ (mostly) decision diagrams, which in continuously changing environments, have a high memory impact when updating those diagrams. The solution presented in this paper uses satisfiability to generate a formula in CNF which is an exact representation of the policy. An existing SMT solver is used to find a solution (if it exists) for the formula. This solution is then used to create the final access decision which can be returned.

Keywords: XACML, *SAT, CNF*

1. INTRODUCTION

In today's businesses, customer data is what makes their business. Large tech companies make their money using their customer's data and as more and more data is stored in the cloud, an important part of their IT security is access control. How does a company makes sure only users that are allowed access particular information, have indeed access without having to little restrictions i.e. allowing everyone to access this information.

In specifying access control policies eXtensible Access Control Markup Language (XACML)[1] has become the de facto standard. XACML does not only provide an XML-based language to specify policies, but also an architecture for the enforcement those policies. It was standardized by the Organization for the Advancement of Structured Information Standards (OASIS) as an open standard for the expression of security policies. As a single common policy language allows for a consolidated view of the security policy as implemented by a system. This eases the management, updates and enforcement of those policies. As XACML uses XML as the policy language it has the extensibility benefits of XML, in syntax as well as semantics, so policies can be created which accommodate the unique requirements as needed by the application [2].

In XACML, when a request is made to access a protected resource, the request is submitted to the Policy Enforcement Point (PEP) which manages this resource. The PEP generates a request in the XACML request language and sends this request to the Policy Decision Point (PDP). The PDP has access to the user created XACML policies (which are written in the XACML policy language) and will based on the policies either permit or deny access to the resource. It will generate a responds in the XACML response language and send it back to the PEP which enforces this decision. In this paper, we will develop an efficient policy evaluation procedure that is applied by the policy decision point (PEP). The state-of-the-art in this research area is represented by XEngine[3] which employs decision diagrams to produce access decision. The problem with this approach is that it is memory hungry and may not scale in certain scenarios where memory is limited. Especially in situations where policies change frequently, rebuilding the policy becomes inefficient.

In this paper we will employ propositional encoding to produce access decisions, by converting XACML policies to logical expressions and use a propositional solver to answer authorization queries. The paper is structured as follows: the next section discusses related work. Section 3 provides background information about both XACML and SAT. Section 4 describes in detail the algorithm that was created during this research. Section 5 will give an overview of the framework that was created. Finally, section 6 concludes this paper.

2. RELATED WORK

Prior work in optimizing the policy evaluation procedure in XACML can be split into two groups:

using (adaptive) reordering [4] and using some decision diagram based approach [3] [5] and [6]. The adaptive reordering technique in [4] is based on statistics and the categorization of policies and rules (within policy sets and policies respectively) such that if a request is received, it is redirected to the policies (or rules) that correspond to its subjects, avoiding any unnecessary evaluations from occurring [4]. However, the problem with this approach is that it will not work as efficiently if the requests do not follow a uniform distribution.

The decision diagram based approaches [3], [5] and [6] work differently, Liu et al. [3] first convert the XACML policy to a numerical policy, so they can use efficient integer comparisons. Then the numerical policy is normalized, so as to convert the hierarchical structure to a flat structure and keeping only one rule combining algorithm. Finally this normalized numerical policy is converted to a (multi value) decision diagram for efficient processing of requests.

Ros et al. [5] propose an optimization for policy evaluations based on two tree structures: the Matching tree (MT) as to allow for fast searching of applicable rules (using a decision diagram) and the Combining Tree (CT) which is used to evaluate the applicable rules. Their approach allows for all comparison functions that are available in XACML to be used and they also support obligations[1] which XEngine does not.

As Cahn et al. [6] point out that the approach taken by [5] still has some drawbacks: it lacks handling of missing attributes and different indeterminate states. The lack in missing attributes leaves the approach open to so-called missing attribute attacks. These attacks work by sending crafted requests, which lack some attributes, as to circumvent the PDP. The lack in indeterminate state handling means rules with indeterminate states are ignored by Ros et al. [5], which in certain cases could result in wrong final decisions. Cahn et al. [6] improve on the research by Ros et al. [5] by introducing multi-data-types interval decision diagrams (MIDD) which are created by analyzing XACML policies as logical expressions. This approach provides correctness and completeness, but still make use of decision diagrams. This approach can cause problems in situations where memory is limited or when XACML policies change often and the decision diagram needs to be recreated which is a costly operation.

3. BACKGROUND

This section gives some background information about XACML and SAT.

3.1. XACML

XACML is declarative language which provides attribute-based access control (ABAC). This means the language uses attributes associated with a request to determine if the desired access to a resource is allowed. XACML uses three elements to describe a policy: a `<Policyset>`, `<Policy>` and a `<Rule>`. A `<Rule>` element contains an expression which, based on attributes, returns an access decision i.e. a rule returns either `permit` or `deny` given a request with attributes. `<Rule>` elements may contain a `<condition>` element which restricts the space of applicable requests. A `<condition>` is a function which determines if a rule applies. A `<Policy>` element contains (a set of) `<Rule>` elements and describes how to combine them to a single decision i.e. how to combine the decisions of each individual rule. A `<Policyset>` element contains other `<Policyset>` elements or (a set of) `<Policy>` elements and describes how to combine the individual decisions to a single decision [2]. All three elements contain `<target>` (they can be empty) elements which contain definitions that describe for which (values of) attributes an element applies.

Given a `<Policyset>` or `<Policy>`, XACML defines algorithms (the *rule-combining algorithm* for `<Policy>` elements and the *policy-combining algorithm* for `<Policyset>` elements) for combining the individual decisions (for rules or policies respectively) to a final decision. The most common combining algorithms[2] are the *deny-overrides*, *permit-overrides*, *first-applicable*, *only-one-applicable*. The *deny-overrides* and *permit-overrides* work as their name suggests, if a single element returns *deny* or *permit* the combined decision is *deny* or *permit* respectively. The result of the *first-applicable* combining algorithm is the same as the result of the first applicable element that is encountered. The *only-one-applicable* combining algorithm can only be applied to policies, it ensures that only one policy (or policy set) is applicable by looking at the `<target>` elements of the respective policies. It results in *NotApplicable* if no policy is applicable and the result is *Indeterminate* if more than one policy can be applied [2].

Each rule in XACML has an effect associated with it. An effect can either be *permit* or *deny*. If a rule evaluates to *true* the rule return the associated effect (i.e. *permit* or *deny*). If the rule evaluates to *false* it returns *NotApplicable*. If for some reason an error occurs when evaluating the rule, it returns *Indeterminate*.

3.2. SAT

The Boolean satisfiability problem (also satisfiability or SAT) is, given a Boolean function f with n variables, the problem of finding appropriate values (i.e. *true* or *false*) of the variables such that f evaluates to *true* or prove that none exists [8].

$$f(x_1, x_2, \ldots, x_n) \tag{1}$$

[1]New in XACML version 3.0 [7]

[2]See [2] for all combining algorithms.

The formula f in equation (1) is called a propositional logic formula. It consists of:

- *variables*, which can either have the value *true* or *false*
- *operators*:
  - *conjunction*, the logical AND operator, denoted by $\wedge$.
  - *disjunction*, the logical OR operator, denoted by $\vee$.
  - *negation*, the logical NOT operator, denoted by $\neg$.
- *parentheses*, for the logical grouping of propositions.

Where a proposition is a function of the type:

$$g : X \rightarrow B \tag{2}$$

In equation (2) X is some arbitrary set and B is a boolean domain, a set consisting of two elements which represent logical values e.g. $B = \{true,\ false\}$.

It is proven that determining if the function f in equation (1) has a satisfying solution belongs to the class of problems known as NP-complete [9]. In addition, SAT is applicable in a wide range of domains such as test case generation [10] and finding bugs in software [11]. As a result, practical SAT solvers are a highly researched subject [12]. Most SAT solvers work on problems where formulas are represented in conjunctive normal form (or CNF) [8] [12]. This form consists of a conjunction of *clauses*. Where a *clause* is a disjunction of one or more *literals*. A *literal* is the smallest logical unit in the problem i.e. a variable or the negation of one (called a positive *literal* or a negative *literal* respectively). An example CNF formula may look like:

$$(p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge (p_5 \vee p_6) \tag{3}$$

Where $p_1, \ldots, p_6$ are *literals* and $(p_1 \vee p_2)$ is a *clause*. The advantage of using CNF is that for a formula to be satisfied, each individual clause must be satisfied. Using CNF is not a limitation on the formulas that can be handled as it is possible to translate any formula into CNF [8].

4. ALGORITHM

This section presents our algorithm which is used to encode XACML policies into CNF formulas. Our algorithm consists of three consecutive steps: the construction of the attribute domains, the flattening of the hierarchical structure of an XACML policy and the final encoding into CNF formulas. We describe the aforementioned steps in detail, showing examples to clarify our approach.

4.1. Constructing attribute domains

XACML policies contain attributes (in the form of the *attributeValue*, *attributeSelector* and *attributeDesignator* elements) with values associated with them. They describe the applicability of the XACML rules. For example, a policy could have a rule stating (the example is edited for brevity):

```
...
<rule Effect="Permit">
...
<AttributeValue DataType="String">admin</AttributeValue>
<AttributeDesignator AttributeId="role" DataType="String"/>
...
</rule>
...
```

Example 1. An example of a rule element in a XACML policy (edited for brevity)

In this case if the request has a *role* of *admin* the requested action would be permitted (assuming no other rules exist). To implement this behavior in our framework, we had to look at what such a rule actually meant. The *AttributeId* specifies the name of the attribute, in this case it's an attribute named *role*. The *DataType* specifies the type of the attribute, in this case it's an attribute of type *String*. Assuming the XACML policy contains more rules, it probably will also contain a few different values for that specific attribute. This means the attribute turns into a set of possible values which exist in the XACML policy, call this set the domain D_{attr} for attribute *attr*. Now let's assume the policy in Example 1 also has rules defined for the *role* attributes: *manager*, *hr* and *user*; the example then turns into the expression $admin \in \{admin, manager, hr, user\}$. The approach of constructing a set of possible values for each type of attribute is the approach we took in our implementation. The type of the attribute has to allow for enumeration i.e. is has to have different yet related values throughout the policy (as with the *role* attribute). To implement this, the first step we took was deciding which attributes we thought were usable for enumeration, and we decided to go for attributes of the type *String*. The second step was enumerating all different values which exist in the XACML policy, the algorithm in pseudo code is shown in Algorithm 1. The algorithm will recursively parse a XACML policy and when an attribute of the correct type is found, the respective value is added as a value to the map m with *DataTypes*.

4.2. Policy flattening

As XACML uses a hierarchical structure for a policy i.e. `<Policyset>` elements can contain other `<Policyset>` elements (see section 3.1), we need a way to flatten this structure to allow for an efficient encoding and optimizing performance by only re-encoding parts of the policy that have changed. To keep the consistency with [13] we use the notation they introduced. For reasons of clarity and as a means to help the reader we give the definition of the *applicability space* and the *decision space* as introduced by [13].

DEFINITION 4.1. *Applicability space: Given a* XACML *policy element p (either a* `<Policyset>`*,* `<Policy>` *or* `<Rule>` *element) which has a (possibly empty) set of*

Algorithm 1 EnumerateVariables

Input: A map m containing the DataTypes as keys and (empty) arrays as values and a policy p

1: **procedure** EnumerateVars(p, m)
2: **for all** target elements **do**
3: update m with values found in the policy target
4: **end for**
5: **for all** variable definitions **do**
6: update m with values found in the variable definitions
7: **end for**
8: **for all** policy elements **do**
9: **if** element is a policy **then**
10: enumerateVars(element,m)
11: **else if** element is a rule **then**
12: update m with values found in the rule targets
13: update m with values found in the rule condition
14: **end if**
15: **end for**
16: **end procedure**

constraints in their targets we define the applicability space of p as the triple $< AS_A, AS_{IN}, AS_{NA} >$ *where* AS_A *represents the access requests to which p applies i.e. the access requests that are allowed by the target constraints of p.* AS_{IN} *represents the access requests for which the access requests are missing required information, hence the applicability of p cannot be determined.* AS_{NA} *represents the access requests for which p does not apply.*

The overall goal of evaluating a XACML policy is to come to an access decision (see section 3.1). The *decision space* groups access requests which have the same access decision together.

Definition 4.2. *Decision space: Given a* XACML *policy element p (either a* `<Policyset>`, `<Policy>` *or* `<Rule>` *element) we define the decision space of p as the tuple* $< DS_P, DS_D, DS_{IN}, DS_{NA} >$ *where each element of the tuple refers to the set of access requests that evaluate to the same access decision. The access decisions are: Permit, Deny, Indeterminate and NotApplicable respectively. Note that the Indeterminate decision space* DS_{IN} *is a tripe* $< DS_{IN(P)}, DS_{IN(D)}, DS_{IN(NA)} >$ *which represent the decisions Indeterminate Permit, Indeterminate Deny, Indeterminate NotApplicable respectively.*

The algorithm as shown in Algorithm 2 is based on the work done by [13], they also provide the mathematical proof that the decision space is mutually exclusive which is very important as it makes sure the PDP will only return a single access decision. Our implementation differs from the one created by [13]

Algorithm 2 FlattenPolicy

Input: A policy p

Output: Decision space $< DS_P, DS_D, DS_{IN(P)}, DS_{IN(D)}, DS_{IN(NA)}, DS_{NA} >$

1: **procedure** flattenPolicy(p)
2: **if** p is a rule **then**
3: $AS^P_A = AS^T_A \cap AS^C_A$
4: $AS^P_{IN} = AS^C_{IN} \cup AS^T_{IN}$
5: **if** effect of p is Permit **then**
6: $DS_P = AS^P_A$
7: $DS_D = \emptyset$
8: $DS_{IN(P)} = AS^P_{IN}$
9: $DS_{IN(D)} = \emptyset$
10: **else if** effect of p is Deny **then**
11: $DS_P = \emptyset$
12: $DS_D = AS^P_A$
13: $DS_{IN(P)} = \emptyset$
14: $DS_{IN(D)} = AS^P_{IN}$
15: **end if**
16: $DS_{IN(PD)} = \emptyset$
17: $DS_{IN(NA)} = \overline{(DS_P \cup DS_D \cup DS_{IN(P)} \cup DS_{IN(D)} \cup DS_{IN(PD)})}$
18: **return** $(DS_P, DS_D, DS_{IN(P)}, DS_{IN(D)}, DS_{IN(PD)}, DS_{IN(NA)})$
19: **else if** p is a policy (set) **then**
20: policies $= \emptyset$
21: **for all** elements e of p **do**
22: result = flattenPolicy(e)
23: add result to policies
24: **end for**
25: combiningAlg = combining algorithm of p
26: **return** applyCA(policies, combiningAlg)
27: **end if**
28: **end procedure**

in that we use only SAT formulas i.e. only boolean predicates, whilst they use SMT formulas and are hence not bound by using only boolean predicates. We only implemented the encoding of a single policy containing a set of rules, their implementation is able to parse a complete policy including nested policy sets. Algorithm 2 works as follows: we feed the algorithm a policy p, the first step is to find the first available applicability constraints. As these applicability constraints work in a top-down manner i.e. the applicability constraints get more specific as you come closer to a specific rule. In our case the first constraints we find are the ones for the rule's *target*, the applicability space of the *target* is combined with the applicability space as induced by the *condition*'s constraints. The *applicability Indeterminate space* is created in the same way, using both the rule *target* and *condition*.

The *decision spaces* are obtained from the *applicability spaces*. The *decision spaces* are created starting from the atomic unit of a XACML policy i.e. a rule and grow in a bottom-up manner. The combined *decision*

space of a policy can be combined with the *decision space* of another policy and so on. The combining of the *decision spaces* is performed according to the combining algorithm as specified by a policy (or policy set). Combining the *decision spaces* is the task of the function *applyCA*.

This function will combining the provided *decision spaces* according to the provided *combingingAlg*. The details of how to combine the elements of the *decision spaces* is clearly described in [13] (see Figure 1: Encoding of XACML v3 combining algorithms), so we are not going to repeat that here. This function will recursively combine the *decision spaces* and when finished return the final *decision space*.

4.3. SAT encoding

Given the final *decision space* as returned by the function *applyCA*, the final step of our algorithm is combining the individual elements of the *decision space* to form the CNF formula. As the individual elements are mutually exclusive, the final CNF formula has the following form:

$$DS_P \cup DS_D \cup DS_{IN(P)} \cup DS_{IN(D)} \cup DS_{IN(PD)} \cup DS_{IN(NA)} \quad (4)$$

The formula in Equation 4, if satisfiable, should return *true* and this means only a single element e.g. DS_P is *true* (as the elements are mutually exclusive). Knowing which element is true, we can deduce the resulting access decision. Which is the final result of our algorithm.

5. FRAMEWORK

The goal of the framework that was created during this research is evaluating a XACML request against a user-defined XACML policy and — using SAT — generating a access decision. It was based on the work done by [13], with the main difference that they used SMT (a superset of SAT) instead of SAT. Hence our framework was also implemented in Java and also employs z3, a theorem prover developed by Microsoft [10], compiled with the Java bindings. As the latest version of XACML (XACML v3) was standardized in January 2013 [2], we decided to only support the latest version as the authors assume it is the most widely used version. However, to the best of our knowledge no current usage statistics exist. The general architecture of the framework is discussed in section 5.1.

5.1. Overview

The framework consists of three components: the preprocessor, the SMT solver and the post processor. A schematic representation of the framework is shown in Figure 1. The preprocessor is used to parse a XACML document (in this case both the XACML and the request) and convert it to CNF as described in section 4. In our implementation the preprocessor only was used to parse the XACML policy and a set of predicates was used to represent the request which made the implementation of the preprocessor considerably easier.

The resulting formulas, describing the policy and request, are then passed on to the SMT solver, which as described before is the open source SMT solver z3 [10]. The SMT solver takes the formula and outputs either of two possible answers: *sat* or *unsat*. If the output is satisfiable i.e. *sat* it means z3 was able to find boolean assignments for the predicates such that the formula returned *true*. If the output is unsatisfiable i.e. *unsat* it means z3 was unable to find such an assignment. In any case the output is not enough for the PDP to return an access decision (recall that the PDP evaluates the request against the XACML policy/policies). Because at this point the output needs to be interpreted as in the case of *sat* z3 only tells us that there is an assignment, but not what it is. To further process the output the post processor was created.

Given the *sat* output of z3 and the mutual exclusivity of the four possible outcomes: *Permit*, *Deny*, *Indeterminate*, *NotApplicable* only one of the outcomes can be *true*. The post processor calls z3 one more time, this time asking for the variable assignment...

If, however, the output is *unsat* it means neither of the four possible outcomes evaluated to *true*, which in this case points to an error somewhere in the implementation, as an XACML request should always receive one of the four available answers. In this case the post processor will return an error describing in which component the problem occurred.

6. CONCLUSION

In this paper we presented a novel approach for the evaluation of XACML policies. Our solution makes use of SAT by converting the XACML policy to a SAT formula which can be solved by using existing (and proven) SMT solvers. Using SAT has the advantage over existing approaches in that is thus not rely on creating tree structures to represent the XACML policy which when created are costly in their memory usage.

In our framework we have only implemented a limited set of functions that are available in the XACML specification [2]. The other existing functions are for now being ignored.

As future work we are planning on evaluating the performance and accuracy of our framework against the existing solutions. This experimental analysis will show if the encoding of XACML to SAT will have the expected performance benefits.

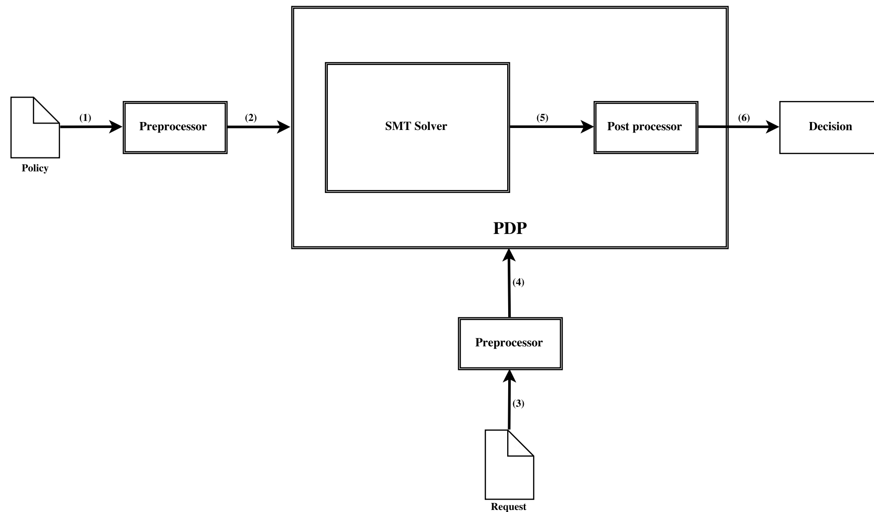

FIGURE 1. A graphical representation of the framework as created during this research. Step (1) shows the XACML policy being parsed by the preprocessor creating the CNF formula. Step (2) is feeding the formula combined with the parsed request (in step (3) and (4)) to the SMT solver which returns either *sat* or *unsat*. Depending on if the formula was satisfiable or not the output is fed to the post processor which returns the final access decision (step (6)).

REFERENCES

[1] extensible access control markup language (xacml). `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`. Accessed: 2016-06-23.

[2] extensible access control markup language (xacml) version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`. Accessed: 2016-06-23.

[3] Liu, A. X., Chen, F., Hwang, J., and Xie, T. (2011) Designing fast and scalable xacml policy evaluation engines. *IEEE Transactions on Computers*, **60**, 1802–1817.

[4] Marouf, S., Shehab, M., Squicciarini, A., and Sundareswaran, S. (2011) Adaptive reordering and clustering-based framework for efficient xacml policy evaluation. *IEEE Transactions on Services Computing*, **4**, 300–313.

[5] Pina Ros, S., Lischka, M., and Gómez Mármol, F. (2012) Graph-based xacml evaluation. *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pp. 83–92. ACM.

[6] Ngo, C., Makkes, M. X., Demchenko, Y., and de Laat, C. (2013) Multi-data-types interval decision diagrams for xacml evaluation engine. *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*, pp. 257–266. IEEE.

[7] Enhancements and new features in #xacml 3.0. `http://www.webfarmr.eu/2010/07/enhancements-and-new-features-in-xacml-3-axiomatics/`. Accessed: 2016-06-24.

[8] Malik, S. and Zhang, L. (2009) Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, **52**, 76–82.

[9] Cook, S. A. (1971) The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158. ACM.

[10] De Moura, L. and Bjørner, N. (2008) Z3: An efficient smt solver. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer.

[11] Jackson, D. and Vaziri, M. (2000) Finding bugs with a constraint solver. *ACM SIGSOFT Software Engineering Notes*, pp. 14–25. ACM.

[12] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001) Chaff: Engineering an efficient sat solver. *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535. ACM.

[13] Turkmen, F., den Hartog, J., Ranise, S., and Zannone, N. (2015) Analysis of xacml policies with smt. *International Conference on Principles of Security and Trust*, pp. 115–134. Springer.