XACML Function Annotations

Prathima Rao Dan Lin Elisa Bertino

Department of Computer Science
Purdue University
{*prao,lindan,bertino*}*@cs.purdue.edu*

Abstract

XACML is being increasingly adopted in large enterprise systems for specifying access control policies. However, the efficient analysis and integration of multiple policies in such large distributed systems still remains a difficult task. In this paper, we propose an annotation technique which is a simple extension to XACML, and may greatly benefit the policy analysis process. We also discuss an important consistency problem during XACML policy translation and point out a few possible research directions.

1 Introduction

XACML (Extensible Access Control Markup Language) [6] is the current OASIS standard for specifying access control policies in large enterprise systems. As XACML is being increasingly adopted [7] in large distributed systems with multiple autonomous parties, it is very important that properties like similarity and consistency be verified in order to enable efficient integration and management of policies in such systems. In order to address such requirement, various policy analysis techniques have been proposed [1, 3, 5].

Current policy analysis techniques rely on analyzing Boolean formulae representing the XACML policies. To derive these formulae, the analysis tool needs to be aware of the behavioral semantics of functions used in the *Condition* element of XACML policies. For standard XACML functions with fixed semantics, a translation has to be carried out by an expert who has to manually map each function to a Boolean predicate. In addition, XACML allows users to define their own extension functions. Existing translation techniques will handle these new functions simply as uninterpreted symbols. Such limitation severely hinders the quality of the analysis. Without knowing the exact semantics of the newly defined functions, the analysis is restricted to the very general assumption that two functions are treated equal if they have the same number and data types of input parameters. Therefore, it is crucial that users also be able to specify a semantics for new functions they define. In addition, explicit semantics for standard XACML functions will obviate the necessary manual translation. For this purpose, we need a technique to convey the function semantics explicitly. Motivated by such need, we propose an annotation language to explicitly and succinctly represent the behavior or semantics of both standard and extension XACML functions. To the best of our knowledge, there is no other formal annotation language defined for functions in XACML policies. There are many advantages of such annotations. First, automatic translation can be achieved by parsing these user-specified annotations. Second, the annotations can help system managers to easily understand each policy. Third, the annotations can help in analyzing policy properties. Finally, comparisons among simple policies may be performed directly by comparing their annotations.

Another important issue we need to address with regard to such annotations is how to ascertain the validity of the function annotations provided by the user, since incorrect semantic annotations may lead to incorrect analysis results. For example, policy P_1 specifies its condition by a user-defined function f_1 which compares the absolute values of two input parameters. If the user who defined the f_1 made a mistake or maliciously wrote the semantics as comparing the exact values of two input parameters, the translation will be wrong, and consequently the policy analysis based on this translation will be meaningless. Such issue does not arise in standard XACML functions since their semantics is defined by XACML. For the user-defined functions, veri-

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="Bill-Policy" RuleCombiningAlgId="permit-overrides">
<Rule RuleId="Rule01" Effect="Permit">
<Condition>
<Apply FunctionId="xacml:1.0:function:string-at-least-one-member-of">
     <SubjectAttributeDesignator AttributeId="E-Mail" Datatype="#string"/>
     <Apply FunctionId="xacml:1.0:function:string-bag">
         <AttributeValue Datatype="#string">.gov</AttributeValue>
         <AttributeValue Datatype="#string">.edu</AttributeValue>
</Apply>
    </Apply>
</Condition> </Rule> </Policy>
```

fication is however important. Therefore, we propose the inclusion of a verification module which checks the consistency between annotations and function implementation. The verification module involves computationally intensive tasks such as proof checking, which may affect the overall responsiveness of the system. However, this is just a one time cost incurred initially when the users register their functions and annotations in the function repository of our system.

The rest of the paper is organized as follows. Section 2 presents an overview of XACML. Section 3 presents our proposed annotation syntax, and Section 4 introduces an annotation framework. Finally, Section 5 concludes the paper.

2 XACML

XACML [6] is the OASIS standard language for access control policies. It is written in XML which has the widespread support from the main platform and tool vendors. The main elements of an XACML policy are a *Target*, a *Rule* set and a *rule combining algorithm* (for conflict resolution). The *Target* specifies some restrictions on the subject, resource, action, environment attributes in a request, that must hold in order for the policy to be applicable to that request. A *Rule* element is in turn composed of *Target*, *Condition* and *Effect* elements.

A *Condition* element specifies additional restrictions on request attribute values that must be satisfied in order to yield a *Permit* or *Deny* decision as specified by the *Effect* element. These restrictions are represented in the form of Boolean *functions* over subject, resource, action, environment attributes or functions of attributes. In effect, conditions represent Boolean predicates over attributes. Thus, most policy analysis tools model XACML policies as Boolean formulae. Property verification is then formulated as a satisfiability problem on these formulae. An example of a XACML policy is shown in Figure 1. The Boolean formula for this policy is (E−Mail == ".gov") **or** (E−Mail == ".edu").

3 Annotation Syntax

In this section, we present the annotation syntax for function specifications. As we mentioned, functions in XACML policies can be translated into Boolean formulae. Our annotation syntax can express any Boolean expression of the form $e_1 \odot e_2 \odot e_3$....., where e_1, e_2, e_3... are Boolean predicates involving comparison of various XACML datatypes, and $\odot$ represents the operators *or*/*and*. We design the following seven elements which are aimed to represent all types of Boolean functions.

- ⟨Annotation⟩ ⟨/Annotation⟩ : The *Annotation* element marks the begin and end of an annotation. It contains an attribute called "AnnotationId", which indicates the corresponding function that this annotation explains. This "AnnotationId" is exactly the same as the "FunctionId" so that automatic translation can easily associate the annotation with its function.
- ⟨Operand ⟩ ⟨/Operand ⟩: The *Operand* element is an input parameter of a function. It has two attributes: "DataType" and "value". The "DataType" indicates the data type of the input parameter. The "value" will be substituted with specific input value when the function is called by a policy.
- ⟨Uni−Operator⟩ ⟨/Uni−Operator⟩: The *Uni-Operator* element denotes any unary operator like +,−, *not*.

```
<Annotation AnnotationId=
  "string-at-least-one-member-of">
<Bi-Term>
  <Operand Datatype= "#string-bag"
           value= "&Param_1"/ >
  <Bi-Operator> belongs-to < /Bi-Operator>
  <Operand Datatype="#string-bag"
           value= "&Param_2"/ >
< /Bi-Term>
<Boolean-Form>
< /Boolean-Form>
< /Annotation>
```

(a)

```
<Annotation AnnotationId=
  "string-at-least-one-member-of">
<Bi-Term>
  <Operand Datatype= "#string-bag"
           value={"E-Mail"}/ >
  <Bi-Operator> belongs-to < /Bi-Operator>
  <Operand Datatype="#string-bag"
           value= {".gov", ".edu"}  / >
< /Bi-Term>
<Boolean-Form>
    (E−Mail == ".gov") or
      (E−Mail == ".edu")
< /Boolean-Form>
< /Annotation>
```

(b)

- ⟨Bi−Operator⟩ ⟨/Bi−Operator⟩ : The *Bi-Operator* element denotes a binary operator. The binary operator could be any one of the arithmetic operators like $+, -, *, /, \%$, logical operators like *or*, *and*, comparison operators like $==, \leq, <, \geq, >$, or set operators like **intersect**, **belongsto**, **union**, **subset**, **setequal**.
- ⟨Uni-Term⟩ ⟨/Uni−Term⟩ : The *Uni-Term* element denotes a term with one operand. It contains a ⟨Uni−Operator⟩ followed by an ⟨Operand⟩ element.
- ⟨Bi-Term⟩ ⟨/Bi−Term⟩ : The *Bi-Term* element denotes a term with two operands connected by a binary operator. Accordingly, it contains two ⟨Operand⟩ elements with a <Bi−Operator> element in between.
- ⟨Boolean-Form⟩ ⟨/Boolean-Form⟩: The *Boolean-Form* element contains a Boolean formula with respect to the function annotated. This part is automatically generated by our annotation system.

Figure 2 illustrates an example of the proposed annotation syntax. Figure 2(a) represents the general annotation, for the function "string-at-least-one-member-of", which is provided when the function is defined. This general annotation is then used to generate an annotation instance (Figure 2(b)) at each call site of the function in the policy (Figure 1). The annotation instance is obtained by first substituting "&Param_1" and "&Param_2" with "E-Mail" and {".gov",".edu"} respectively and then generating the corresponding boolean formula.

The Boolean formula generated in the *Boolean-Form* element can be consumed by a policy analysis tool. For example, consider a policy analysis tool that performs *similarity analysis* on policies. Given two policies P_1 and P_2, a similarity analyzer tool, enumerates the relationship between the sets of requests permitted (denied) by P_1 and P_2. Such a tool typically abstracts policies into Boolean expressions and uses SAT solving or model checking techniques to determine the relationships between them. Similarity analysis is very useful in scenarios where one wants to determine compatibility between policies, for example in service virtualization systems where a data owner would want to store his data on a machine whose policies closely matches his own policies.

Let the P_1 represent a data owner policy that permits access to its files between 8 A.M - 5 P.M to users whose E-mail id belongs to the ".edu" domain. Let P_2 be a resource owner policy that permits access to files on its machine between 6 A.M - 10 P.M to users whose E-Mail id belongs to the ".edu" or ".gov" domain. Using our annotation syntax P_1 can be represented by the *Boolean-Form* element:
⟨Boolean-Form⟩
(8 A.M $\leq$ Time $\leq$ 5 P.M) **and** (E-Mail == ".edu")
⟨/Boolean-Form⟩.
Similarly P_2 can be represented as :
⟨Boolean-Form⟩
(6 A.M $\leq$ Time $\leq$ 10 P.M) **and** (E-Mail == ".edu" **or** E-Mail == ".gov")
⟨/Boolean-Form⟩.

A policy similarity analyzer can directly consume the Boolean expressions for P_1 and P_2 to perform the similarity analysis.

The proposed annotation syntax can be used to express different categories of Boolean expressions (as identified in [1]), for which the satisfiability problem is tractable and which occur often in policies of varied domains. Thus our annotation syntax can support state of the art policy analysis tools.

4 The Annotation Framework

We now proceed to describe how annotations work. An overview of the annotation framework is shown in Figure 3. The framework consists of two main components: the *annotated function repository* and the *annotation module*.

The *annotated function repository* stores function names paired with their annotations. Each time a new function is defined, it needs to be registered in this annotated function repository. Note that users are required to provide such annotations only once. Then, when the function is used in a policy, the user does not need to write a specific annotation for it. Instead, our system will automatically generate the corresponding annotation through the *annotation module*.

The *annotation module* takes XACML policies and the annotated function repository as its inputs. There are two components in this module. One is the *annotation verifier*, which is responsible for verifying the consistency between the function implementation and its specified annotation. The other is the *annotation interpreter*, which add annotations below each function in a specific policy, and also generate boolean formulae.

Let us have a closer look at the annotation interpreter. There are two phases in interpreting a function. In the first phase, for each occurrence of a function in a given XACML policy, the annotation interpreter retrieves the annotation for this function from the annotated function repository according to the function name. Then, the interpreter replaces the value of each operand with the actual arguments passed as inputs to the function in the policy, thereby generating specific instances of the annotations.

The second phase is to generate the Boolean formula for the annotation instance. Such step is achieved by concatenating variables with operators according to the order of their appearance in the annotation instance. For some complicated operators such as set operators, their semantics is hard-coded in the annotation interpreter. In the previous example (Figure 2), the interpreter knows that the semantics of the operator "belongs-to" is "at least one member of its first operand set must be equal to one member of its second operand set", and hence the interpreter is able to generate the desired Boolean expression. The annotation syntax and hence the interpreter can be extended to support complex domain specific operators that might be necessary for complicated extension functions.

The Boolean formula generated by the annotation module can be consumed by any external XACML policy analysis tool. It is worth noting that the format of the Boolean expression to be generated can be specified as an additional parameter to the interpreter by the external XACML policy analysis tool.

4.1 Annotation Consistency Verification

As we mentioned in the introduction, the annotation specified by users may be incorrect due to numerous reasons, and incorrect annotations may have negative influence on the policy analysis. To address this issue, we propose the inclusion of an annotation verifier component in our system. This verifier checks the consistency between the given annotation and its associated function implementation.

For Java implementations of the XACML functions, we propose the use of verification techniques employed in the context of JML[4]. JML is an an-

notation language for Java classes and methods, which enables formal specifications of properties and requirements like *pre-/post-conditions*. Several proof tools (e.g. JACK[2]) exist that convert JML annotated Java code to formal models, which in turn are fed as theories to an underlying proof checker in order to prove that a Java method meets its JML specification. In a similar spirit, our XACML function annotation can be viewed as a form of *postcondition* specification for XACML. In order to perform the annotation verification we can either (i) translate our annotations into JML *post-conditions* and then use one of the available proof tools to perform the verification, or (ii) develop a new tool that would translate the annotations to theories suitable for use with a chosen proof checker. The former approach is simpler and more suitable for Java implementations since all the necessary components are readily available. While the latter approach would involve more effort with respect to performing the translation to theories in the chosen proof checker, but is necessary for non Java implementations of functions.

The computation complexity of this verification component could be high. However, this phase is optional and needs to be performed only once when the function and its annotation is registered with repository.

5 Conclusion

In this paper we have presented an annotation technique which can benefit XACML policy analysis tools. We proposed the annotation syntax as well as the annotation framework. Specifically, each function is required to be registered once in the annotated function repository together with its annotation. Then our system can automatically generate annotation instance for specific policies. The generated annotations not only help understanding of policies, but also contains boolean formulae which can be directly consumed by most existing policy analysis tools. Further, we addressed the important issue of verifying the consistency of functions and associated annotations.

References

[1] D. Agrawal, J. Giles, K. W. Lee, and J. Lobo. Policy ratification. In *Proc. Policy*, pages 223–232, 2005.

[2] L. Burdy and A. Requet. Jack : Java applet correctness kit. In *In GDC 2002, Singapore, November 2002.*, 2002.

[3] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. ICSE*, pages 196–205, 2005.

[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[5] P. Mazzoleni, E. Bertino, and B. Crispo. XACML policy integration algorithms. In *Proc. SACMAT*, pages 223–232, 2006.

[6] T. Moses. Extensible access control markup language (XACML) version 1.0. *Technical report, OASIS*, 2003.

[7] http://docs.oasis-open.org/xacml/xacmlRefs.html #Products.