Conformance Checking of Access Control Policies Specified in XACML

Vincent C. Hu[1] Evan Martin[2] JeeHyun Hwang[2] Tao Xie[2]

[1] Computer Security Division, National Institute of Standards and Technology, USA

[2] Department of Computer Science, North Carolina State University, USA

vincent.hu@nist.gov, eemartin@ncsu.edu, jhwang4@ncsu.edu, xie@csc.ncsu.edu

Abstract

Access control is one of the most fundamental and widely used security mechanisms. Access control mechanisms control which principals such as users or processes have access to which resources in a system. To facilitate managing and maintaining access control, access control policies are increasingly written in specification languages such as XACML. The specification of access control policies itself is often a challenging problem. Furthermore, XACML is intentionally designed to be generic: it provides the freedom in describing access control policies, which are well-known or invented ones. But the flexibility and expressiveness provided by XACML come at the cost of complexity, verbosity, and lack of desirable-property enforcement. Often common properties for specific access control policies may not be satisfied when these policies are specified in XACML, causing the discrepancy between what the policy authors intend to specify and what the actually specified XACML policies reflect. In this position paper, we propose an approach for conducting conformance checking of access control policies specified in XACML based on existing verification and testing tools for XACML policies.

1. Introduction

Access control is one of the most fundamental and widely used security mechanisms, especially in web applications. It controls which principals such as users or processes have access to which resources in a system. To facilitate managing and maintaining access control, access control policies are increasingly written in specification languages such as XACML [2] and Ponder [10]. Whenever a principal requests access to a resource, that request is passed to a software component called a *Policy Decision Point* (PDP). A PDP evaluates the request against the specified access control policies, and permits or denies the request accordingly.

Assuring the correctness of policy specifications is becoming an important and yet challenging task, especially as access control policies become more complex and are used to manage a large amount of sensitive information organized into sophisticated structures. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure the policy specifications truly encapsulate the desires of the policy authors.

For its independency of platform systems and flexibility in expression, XACML [2] is one of the popular mechanisms for specifying access control policies. XACML is designed with the basic enforcement mechanisms of access control in mind, but not constructed toward any particular policy or model; it provides the freedom in describing access control policies, which are well-known or invented ones. The flexibility and expressiveness of XACML comes at the cost of complexity and verbosity; furthermore, there is no efficient feature in XACML that allows policy authors to check the conformance and integrity of the specified policy (or its model) with respect to the semantic consistency of an access control policy [15]. To address the issue, we propose an approach for conducting conformance checking of access control policies specified in XACML, based on previous XACML policy verification and testing tools.

The rest of the paper is organized as follows. Section 2 presents background information of XACML. Section 3 presents an XACML policy verification tool developed by other researchers and XACML policy testing tools developed in our previous work. Section 4 describes the proposed approach of conformance checking based on previous verification and testing tools. Section 5 describes related work, and Section 6 concludes.

2. XACML

The OASIS (Organization for the Advancement of Structured Information Standards) standards XACML (eXtensi-

ble Access Control Markup Language) [2] and SAML (Security Assertion Markup Language) [3] are two of the important authorization-related standards [26]. XACML is an XML-based general-purpose language used to describe policies, requests, and responses for access control policies. It provides a flexible and mechanism-independent representation of access rules that vary in granularities, allowing the combination of different authoritative domains' policies into one policy set for making access control decisions in a widely distributed system environment.

The five basic elements of XACML policies are `PolicySet`, `Policy`, `Rule`, `Target`, and `Condition`. A policy set is simply a container that holds other policies or policy sets. A policy is expressed through a set of rules. With multiple policy sets, policies, and rules, XACML must have a way to reconcile conflicting rules. A collection of combining algorithms serves this function [2]. Each algorithm defines a different way to combine multiple decisions into a single decision. Both *policy* combining algorithms and *rule* combining algorithms are provided. Seven standard combining algorithms are provided but user-defined combining algorithms are also allowed [4].

To aid in matching requests with the appropriate policies, XACML provides a target [2], which is basically a set of simplified conditions for the subject, resource, and action that must be met for a policy set, policy, or rule to apply to a given request. Once a policy or policy set is found to apply to a given request, its rules are evaluated to determine the response.

XACML also provides attributes, attribute values, and functions. Attributes are named values of known types that describe the subject, resource, and action of a given access request [2]. A request is formed of attributes that will be compared to attributed values in a policy to make the access decisions. Attribute values from a request are resolved through two mechanisms: the `AttributeDesignator` and the `AttributeSelector` [2]. The former lets the policy specify an attribute with a given name and type, whereas the latter allows a policy to look for attribute values through an XPath query.

Figure 1 shows an example XACML policy, which is revised and simplified from a sample Fedora[1] policy. This policy has one policy element which in turn contains two rules. The rule composition function is "first-applicable", meaning the first applicable rule encountered during evaluation is returned as the decision. Lines $2-13$ define the target of the policy, which indicates that this policy only applies to those access requests of an object "demo:5". The target of Rule 1 (Lines $15-25$) further narrows the scope of applicable requests to those asking to perform "Dissemination" action on object "demo:5". Its condition (Lines $26-35$) indicates that if the subject's "loginId" is "testuser1", "testuser2", or "fedoraAdmin", then the request should be denied. Otherwise, according to Rule 2 (Line 37) and the rule composition function of the policy (Line 1), a request applicable to the policy should be permitted.

[1] `http://www.fedora.info`

```
 1<Policy Id="demo" RuleCombAlgId="first-applicable">
 2 <Target>
 3   <Subjects> <AnySubjects/> </Subjects>
 4   <Resources>
 5    <Resource>
 6     <ResourceMatch MatchId="equal">
 7      <AttrValue>demo:5</AttrValue>
 8      <ResourceAttrDesignator AttrId="objectid"/>
 9     </ResourceMatch>
10    </Resource>
11   </Resources>
12   <Actions> <AnyAction/></Actions>
13 </Target>
14 <Rule RuleId="1" Effect="Deny">
15  <Target> <Subjects><AnySubject/></Subjects>
16   <Resources> <AnyResource/> </Resources>
17   <Actions>
18    <Action>
19     <ActionMatch MatchId="equal">
20      <AttrValue>Dissemination</AttrValue>
21      <ActionAttrDesignator AttrId="actionid"/>
22     </ActionMatch>
23    </Action>
24   </Actions>
25  </Target>
26  <Condition FunctionId="not">
27   <Apply FunctionId="at-least-one-member-of">
28    <SubjectAttrDesignator AttrId="loginid"/>
29    <Apply FunctionId="string-bag">
30     <AttrValue>testuser1</AttrValue>
31     <AttrValue>testuser2</AttrValue>
32     <AttrValue>fedoraAdmin</AttrValue>
33    </Apply>
34   </Apply>
35  </Condition>
36 </Rule>
37 <Rule RuleId="2" Effect="Permit"/>
38</Policy>
```

Figure 1. An example XACML policy

3. Policy Verification and Testing

Policy verification and testing are important techniques for high assurance of correct specification of access control policies. Our proposed conformance checking approach is based on an existing XACML policy verification tool (developed by Fisler et al. [14]) and policy testing tools (developed in our previous work [22–25]). Because the policy verification tool can handle only a subset of XACML features and may not handle well complex policies or properties, our proposed approach also additionally exploits our policy testing tools to conduct conformance checking.

3.1. Policy Verification

Margrave [14] is a software tool suite written in PLT Scheme [13] for verifying properties against access control policies written in XACML. Margrave is implemented on top of the CUDD package [27]. CUDD provides an effi-

cient implementation of multi-terminal binary decision diagrams (MTBDDs). Margrave represents XACML policies as MTBDDs, which are a decision diagram that maps bit vectors over a set of variables to a finite set of results. Margrave allows the user to specify various forms of constraints as properties in the Scheme programming language. Margrave's API can verify these properties and if there exist any counterexamples (being specific requests) that violate the specified properties, these counterexamples are produced.

3.2. Policy Testing

Our previous work has developed a set of testing tools for XACML policies, including a fault model and its supporting mutation testing tool [24], a structural coverage measurement tool [25], and several test generation tools [22, 23]. In policy testing, test inputs are access requests and test outputs are access responses. The execution of a test input occurs as a request is evaluated by the PDP against the access control policy under test. Policy authors can inspect request-response pairs to check whether they are as expected. As with software verification, formal policy verification and testing techniques are complementary means to achieve the same goal.

3.2.1 A Fault Model and Mutation Testing

A fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software. In our case, we will model things that could go wrong when constructing an access control policy. With this fault model, we can guide the development of testing techniques and investigate these techniques' effectiveness against the fault model. Any fault results in a semantic change in the policy but we broadly categorize faults as being semantic or syntactic as follows:

Semantic Faults. Semantic faults are more elusive because they involve incorrect use of the logical constructs of the policy specification language. For XACML policies, these logical constructs include policy or rule combining algorithms, policy evaluation order, rule evaluation order, and various functions found in the condition. Because these are logical errors in the construction of the policy, it is unlikely that static analysis can find such errors. We define and implement several mutation operators that emulate semantic faults [24].

Syntactic Faults. Syntactic faults are easier to make and consist of simple typos that result in a syntactically correct policy but a semantically faulty one. Indeed syntactic faults may result in syntactically incorrect policies but we assume that basic static analysis tools exist to check for such inconsistencies. For example, in XACML, an XML schema definition (XSD) can be used to check for obvious syntactic flaws. Syntactic faults that do not violate the XSD can occur due to typos in attribute values. We define and implement three mutation operators that emulate syntactic faults [24].

Mutation testing [11] has historically been applied to general-purpose programming languages in measuring the quality of tests or selecting tests. Based on the proposed fault model for access control policies, we have developed a mutation testing tool [24] that automatically seeds a policy under test with faults by applying these mutation operators, thereby producing numerous faulty policies.

3.2.2 Structural Coverage Criteria

Our previous work [25] proposes structural coverage criteria for XACML policies based on observing whether each individual policy element is involved when a request is evaluated. If no requests are evaluated against a rule during testing, then potential errors in that rule cannot be discovered. Thus, it is important to generate requests so that a large portion of rules are involved in the evaluation of at least one of the requests. In XACML, we can see there are three major entities: policies, rules for each policy, and a condition for each rule. We define three policy coverage metrics for each of these entities [25]:

- *Policy coverage*. A policy is covered by a request if the policy is applicable to the request *and* the policy contributes to the decision; in other words, all the conditions in the policy's target are satisfied by the request and the PDP has yet to fully resolve the decision for the given request. Policy coverage is the number of covered policies divided by the number of total policies.
- *Rule coverage.* A rule for a policy is covered by a request if the rule is also applicable to the request *and* the policy contributes to the decision; in other words, the policy is applicable to the request and all the conditions in the rule's target are satisfied by the request and the PDP has yet to fully resolve the decision for the given request. Rule coverage is the number of covered rules divided by the number of total rules.
- *Condition coverage.* The evaluation of the condition for a rule has two outcomes: true and false, which are called as the true condition and false condition, respectively. A true condition for a rule is covered by a request if the rule is covered by the request and the condition is evaluated to be true. A false condition for a rule is covered by a request if the rule is covered by the request and the condition is evaluated to be false. Condition coverage is the number of covered true conditions and covered false conditions divided by twice of the number of total conditions.

To automate the measurement of structural coverage, we developed a measurement tool [25] implemented by instrumenting Sun's open source XACML implementation [4].

3.2.3 Test Generation

Our previous work [22, 23] developed two test generation techniques, which have different levels of analysis cost and quality of generated tests.

Test Generation based on Solving Single-Rule Constraints. To generate tests for achieving high coverage based on structural coverage criteria, we developed a technique that considers each rule in isolation and attempts to satisfy the constraints required for that rule to be applied [22]. A request set is generated that satisfies all possible combinations of truth values for each independent clause. Therefore, a predicate with n independent clauses has 2^n possible assignments and so at most 2^n requests are generated for each rule.

Test Generation based on Change-Impact Analysis. We developed a test generation tool [23] that generates tests by iteratively manipulating inputs to Margrave [14], which can also conduct change-impact analysis. Based on the policies under test, we automatically synthesize two policy versions whose differences are the coverage targets for test generation. Then these two versions are fed to Margrave, which generates counterexamples to witness the behavioral differences of the two versions, thus covering the coverage targets.

4. Policy Conformance Checking

We propose to conduct conformance checking on XACML policies against binding of policies rules (Section 4.1) and some generic features of any access control mechanism (Section 4.2). We also propose an implementation of conformance checking based on previous XACML policy verification and testing tools (Section 4.3).

4.1. Rule Binding

Rule-binding properties assure that the XACML implementation of an access control policy does not authorize users' access requests that are not permitted by the rules of the access control policy being implemented. We focus on conformance checking for rule bindings in multilevel access control [6], role-based access control [12], and Chinese wall [8]. We focus on these three types of access control policies not only because they are popular but also because they can be formally modeled; so we can verify our results (of inconsistency faults) formally against the model. In future work, we plan to include more types of policies that may be hard to described by a formal model, such as Discretionary Access Control (DAC) [1] and other composed policies. But the following major three shall be significant and useful enough for most XACML access control implementations.

Multilevel Access Control. We check whether a policy specified in XACML enforces the **Bell LaPadula** confidentiality model and the **Biba** integration models [6]. There are two types of checking. First, we check binding of **Bell LaPadula** confidentiality model to make sure that the XACML policy is confined to the following properties of the **Bell LaPadula** model:

- Check if **security classes** $C(s)$ or $C(o)$ for every subject s and object o are checked for every access request;
- Check if the **Simple Security Property** is enforced: a subject s may have **read** access to an object o only if $C(o) \leq C(s)$;
- Check if the *** (star) Property** is enforced: a subject s who has **read** access to an object o may have **write** access to an object p only if $C(o) \leq C(p)$.

Second, we check binding of **Biba** integration model to make sure that the XACML policy is confined to the following properties of the **Biba** model:

- Check if integrity classification scheme, i.e., $I(s)$ and $I(o)$ for every subject s and object o are checked for every request;
- Check if the **Simple Integrity Property** is enforced: subject s can **modify** (have **write** access to) object o only if $I(s) \geq I(o)$;
- Check if **Integrity *-Property** is enforced: if subject s has **read** access to object o with **integrity level** $I(o)$, s can have **write** access to object p only if $I(o) \geq I(p)$.

Role-Based Access Control (RBAC) [12]. The checking should check whether the following two core RBAC properties are enforced in an XACML policy for RBAC [5]. The first core property is role authorization property: a subject can never have an active role that is not authorized for its user. The second property is object access authorization property: a subject s can perform an operation op on object o only if there exists a role r that is included in the subject's active role set and there exists an permission that is assigned to r such that the permission authorizes the performance of op on o.

Chinese Wall [8]. We check if the XACML policy enforces the **Conflict-of-Interest separation** models of **Chinese Wall policy** [8]: if subject s has accessed to object o in group x, subject s will not be granted access to group y if objects in group x and group y are conflict-of-interest (COI). Thus, the checking should detect whether the XACML policy rules allow a subject to access objects in the COI groups.

4.2. Access Control Features

While rule-binding properties are specific to an access control policy itself, access control features are common to most of all access control policies including formal (being described by a model) or improvised (rule-based with no formal model) ones. We focus on conformance checking for two generic checkable features of any access control mechanism.

Safety. Based on the constraints of access control rules (for example, user x is not allowed to perform y on object z), the checking should be able to check whether there is any leaking of privilege that the access (x perform y on z) is granted through the XACML specification. The checking is to detect whether any specified combinations of XACML rules grant an access (subject s perform operation p to object o), which is not allowed by the access control policy.

Separation of Duties (SOD). An SOD policy makes sure that any subject s in user group A will not be granted to access objects in group X if s is also a member of group B. This checking checks whether there is a subject assignment in XACML that allows a subject s being a member of groups A and B to access object in group X, if the constraint is defined.

4.3. Proposed Implementation

To implement conformance checking for XACML, we propose to build tools upon Margrave [14], an XACML policy verification tool and our previous testing tools [22–25].

We propose to develop a tool for generating concrete checkable properties for a given XACML policy based on both information in the policy and generic properties (the rule bindings or features) described in Section 4, because these generic properties may not be directly checkable statically or dynamically and need to be instantiated to concrete properties with elements in the given XACML policy.

To conduct conformance checking statically, we propose to synthesize generated concrete properties specified in Scheme [13] (in the format that can be recognized by Margrave [14]) and then invoke Margrave with the policy and these properties. Margrave will report property violations if any.

Because sometimes the XACML policy under checking may not be able to be handled by Margrave, we propose to conduct conformance checking dynamically through policy testing. Given a property for conformance checking, we statically scan through the policy to identify likely locations that may be related to the property and consider these locations as coverage targets. Given these coverage targets, our policy test generation tools [22, 23] can be used to generate requests to cover these locations. Then the request-response pairs can be checked against the property.

We propose to empirically investigate the relationship between conformance checking and structural coverage [25]. In particular, we plan to investigate whether a full-structural-coverage test suite can help detect all or most faults that are caused by property violations in our conformance checking. We also propose to empirically investigate the relationship between conformance checking and fault models [24]. In particular, we plan to investigate whether some specific types of faults (implemented in mutation operators) would be more likely to cause the violation of the properties described in Section 4 for conformance checking.

5. Related Work

Much work has been done in verification of access control policies. One important aspect of policy verification is to formally check general properties of access control policies, such as inconsistency and incompleteness [7, 18, 20, 21]. In the former case, an access request can be both accepted and denied according to the policy, while in the latter case the request is neither accepted nor denied. Although efficient algorithms have been proposed to perform such verification for specific systems [17, 19], this problem can be intractable or even undecidable when dealing with policies that involve complex constraints.

Besides the verification of general properties, several tools have been developed to verify properties for XACML policies [2]. Hughes and Bultan translated XACML policies to the Alloy language [16] and checked their properties using the Alloy Analyzer. Fisler et al. [14] developed a tool called Margrave that uses multi-terminal binary decision diagrams [9] to verify user-specified properties and perform change-impact analysis. Zhang et al [29] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML [28]. Given an XACML policy and generic properties, our proposed approach conducts conformance checking statically by automatically synthesizing concrete properties for static policy verification (based on Margrave [14]). In addition, our proposed approach also exploits testing tools to conduct conformance checking dynamically.

6. Conclusion

Access control has been one of the most fundamental and widely used security mechanisms. To facilitate managing and maintaining access control, access control policies are increasingly written in specification languages such as XACML. XACML is intentionally designed to be generic: it provides much freedom and flexibility in describing access control policies. XACML does not provide specific

features to allow the policy authors to check the conformance and integrity of the specified policy (or its model) with respect to the semantic consistency of an access control policy. To address the issue, we have proposed an approach for conducting conformance checking of access control policies specified in XACML, based on previous XACML policy verification and testing tools. In particular, we propose to synthesize concrete properties from the policy under checking and desirable generic properties, and then feed the synthesized concrete properties to a policy verification tool or policy testing tools.

References

[1] National Computer Security Center. A guide to understanding discretionary access control in trusted systems, Report NCSC-TG-003, Version 1, 30 September 1987.

[2] OASIS eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`, 2005.

[3] OASIS Security Assertion Markup Language (SAML). `http://www.oasis-open.org/committees/security/`, 2005.

[4] Sun's XACML implementation. `http://sunxacml.sourceforge.net/`, 2005.

[5] A. Anderson. XACML profile for role based access control (RBAC). OASIS Committee Draft 01, February 2004.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations, 1973. MITRE Corporation.

[7] P. Bonatti, S. Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. ACM Conference on Computer and Communication Security*, Athens, Greece, November 2000.

[8] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[9] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis*, 1993.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[12] D. Ferraiolo and R. Kuhn. Role based access control. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[13] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A Progamming Environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[14] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. 27th International Conference on Software Engineering*, pages 196–205, 2005.

[15] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn. Assessment of access control systems. Interagency Report 7316, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Fort Collins, Colorado, September 2006.

[16] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.

[17] T. Jaeger, X. Zhang, and F. Cacheda. Policy management using access control spaces. *ACM Transactions on Information and System Security*, 6(3), 2003.

[18] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[19] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[20] M. Kudo and S. Hada. XML document security based on privisional authorization. In *Proc. ACM Conference on Computer and Communication Security*, Athens, Greece, November 2000.

[21] E. C. Lupu and M. Sloman. Conflict in policy-based distributed systems management. *IEEE Transaction on Software Engineering*, 25(6):852–869, 1999.

[22] E. Martin and T. Xie. Automated test generation for access control policies. In *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering*, November 2006.

[23] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems (SESS 2007)*, May 2007.

[24] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 11th International Conference on World Wide Web*, 2007.

[25] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security*, pages 139–158, December 2006.

[26] M. Naedele. Standards for XML and web services security. *Computer*, 36(4):96–98, 2003.

[27] F. Somenzi. CUDD: CU Decision Diagram Package Release, 1998.

[28] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proc. 2004 ACM Workshop on Formal Methods in Security Engineering*, pages 56–65, 2004.

[29] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. 8th International Conference on Information Security*, pages 446–460, September 2005.