New Approach for Testing the Correctness of Access Control Policies

Suraj Sharma[1], S.K.Jena[2], and K.Satyababu[3]
NIT Rourkela/CSE, Rourkela, India
[1]suraj.sine@gmail.com,[2] skjena@nitrkl.ac.in, [3]ksatyababu@nitrkl.ac.in

***Abstract*—To increase the confidence in the correctness of specified policies, policy developers can conduct policy testing by supplying typical test inputs (request) and subsequently checking test output (responses) against expected ones to enhance the correctness of specified policies. Testing of Access Control Policies along with the Application program is not a worthful practice. Unlike Software Testing we have the tools and technique for Access Control Policy Testing. Unfortunately, manual testing is tedious and time consuming job. We designed a model called ACPC (Access Control Policy Checker) which include mutation operators for comparing the original policy response with the response of mutant policy and check the correctness of the original policy. The ACPC includes two sections in first section we generate the requests set automatically which is previously not available and in second section we perform testing. This model uses the policy written in XACML (eXtensible Access Control Markup Language) [1] which is the standard language for writing Access Control Policies. We have used a tool called Margrave [8] for Change Impact Analysis and other programming languages like Java and C++ for building different module.**

***Index Terms*— Access-control policies, change-impact analysis, verification, mutation, XACML**

I. INTRODUCTION

The purpose of **A**ccess **C**ontrol is to limit the actions or operations that a legitimate user of a computer system can perform. Access control constrains what a user can do directly, as well what programs executing on behalf of the user are allowed to do. In this way access control seeks to prevent activity which could lead to breach of security [4].

Access Control relies and coexists with other security services in a computer system, as in figure 1.

- Access Control is concerned with limiting activity of legitimate users. It is enforced by a Reference Monitor which mediates every attempted access by a user (or program executing on behalf of that user) to objects in the system.
- The Reference Monitor consults an Authorization Database in order to determine if the user attempting to do an operation is actually authorized to perform that operation.
- Authorizations in this database are administered and maintained by a Security Administrator. The Administrator sets these authorizations on the basis of the security policy of the organization.

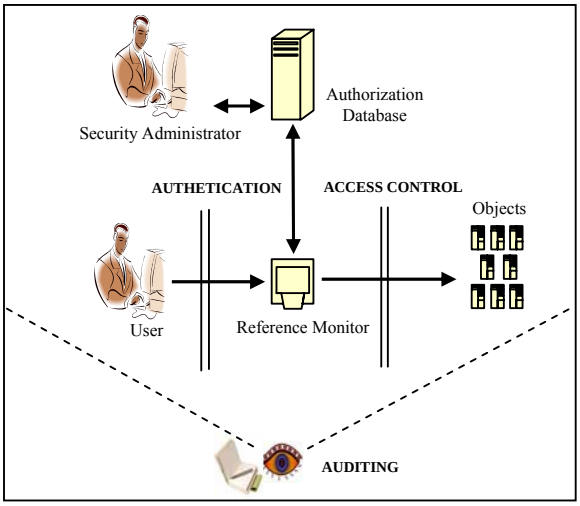

Figure 1. Access Control and Other Security Services

- Auditing monitors and keeps a record of relevant activity in the system.

It is important to make clear distinction between authentication and Access Control. Correctly establishing the identity of the user is the responsibility of the Authentication of the user has been successfully verified prior to enforcement of Access Control via a reference monitor.

In general, there do not exist policies which are "better" than others, rather there exist policies which ensure more protection than others. However, not all system has the same protection requirements. Policies suitable for a given system my not be suitable for another. For instance, very strict Access Control policies, which are crucial to some system, may be inappropriate for environments where users require greater flexibility. The choice of Access Control policy depends on the particular characteristics of the environment to be protected.

Let us have a brief idea about **A**ccess **C**ontrol **P**olicies:-

1. Classical Discretionary Policies
2. Classical Mandatory Policies and
3. The emerging Role-Based Policies

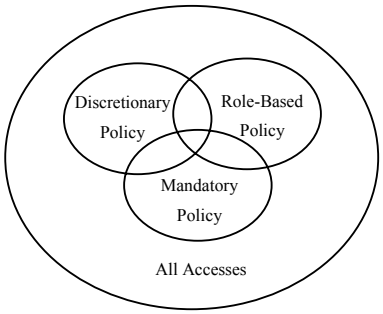

Figure 2. Multiple Access Control Policies

It should be noted that access control policies are not necessarily exclusive. Different policies can be combined to provide a more suitable protection system. Such combination of policies is relatively straightforward as long as there are no conflicts where one policy asserts a particular access must be allowed while another one prohibits it [4].

Access control is one of the most fundamental and widely used security mechanisms. 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 [1] and Ponder [7]. 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.

The rest of the paper is organized as follows: Section II discusses related work. Section III present Motivation of the paper. Section IV shows an example of XACML policy and description of it. In section V, we propose the framework which checks the correctness of the policies in two different sections. Validating the Framework in section VI and section VI concludes the paper.

II. Related Work

One important aspect of policy verification is to formally check general properties of access control policies, such as inconsistency and incompleteness [14,15]. In 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 [14,18] 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 [1]. Hughes and Bultan translated XACML policies to the Alloy language [17] and checked their properties using the Alloy Analyzer. E. Martin and T. Xie [3,6] given the fault model for verification of access control policies and used minimal cover concept for reducing the number of generated requests . Fisler et al. [5] developed a tool called Margrave that uses multi-terminal binary decision diagrams [8] to verify user-specified properties and perform change-impact analysis. Zhang et al [16] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML. These existing approaches assume that policies are specified using a simplified version of XACML. It is challenging to generalize these verification approaches to support full-feature XACML policies with complex conditions. In addition, most of these approaches require users to specify a set of properties to be verified; however, policy properties often do not exist in practice. The Mutation policy testing approach proposed in this paper works on full-feature XACML policies without requiring properties, complementing the existing policy verification approaches.

Flaws in the previous models:

1. Manual generation of requests.
2. Random method of generation of requests [3].
3. Over burden to reduce the requests set according to minimal coverage [6].

III. Motivation

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 through systematic testing to ensure that the policy specifications truly encapsulate the desires of the policy authors. Like software verification and testing techniques, formal policy verification and testing techniques are complementary means to achieve the same goal.

To reduce the flaws of the previous work we proposed the new framework for making the whole testing processes automated.

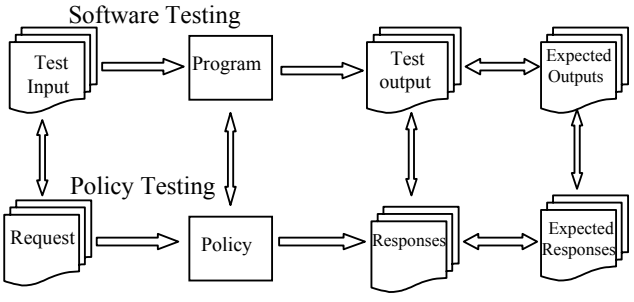

Figure 3: Analogy Between traditional Software and Policy Testing

IV. XACML

XACML (eXtensible Access Control Markup Language) is a language specification standard designed by OASIS[1]. It can be used to express domain-specific access control policy languages as well as access request languages. Besides offering a large set of built-in functions, data types, and combining logic, XACML also provides standard extension interfaces for defining application-specific features. Since it was proposed, XACML has received much attention from both the academia and the industry. Many domain-specific access control languages have been developed using XACML .Open source XACML implementations are also available for different platforms (e.g., Sun's XACML implementation [2] and XACML.NET). Therefore, XACML provides an ideal platform for the development of policy testing techniques that can be easily applied to multiple domains and applications. The basic concepts of access control in XACML include *policies*, *rules*, *targets*, and *conditions*. A single access control policy is represented by a policy element, which includes a target element and one or more rule elements. A target element contains a set of constraints on the subject (e.g., the subject's role is equal to faculty), resources (e.g., the resource name is grade), and actions (e.g., the action name is assign)[1] . A target specifies to what kinds of requests a policy can be applied. If a request cannot satisfy the constraints in the target, then the whole policy element can be skipped without further examining its rules.

We next describe how a policy is applied to a request in details. A policy element contains a sequence of rule elements. Each rule also has its own target, which is used to determine whether the rule is applicable to a request. If a rule is applicable, a condition (a boolean function) associated with the rule is evaluated. If the condition is evaluated to be true, the rule's effect (Permit or Deny) is returned as a decision; otherwise, NotApplicable is returned as a decision. If an error occurs when a request is applied against policies or their rules, Indeterminate is returned as a decision. More than one rule in a policy may be applicable to a given request.

To resolve conflicting decisions from different rules, a rule combining algorithm can be specified to combine multiple rule decisions into a single decision.

```
1<Policy PolicyId="demo" RuleCombinationAlgId= "first-applicable">
2 <Target>
3 <Subjects> <AnySubjects/> </Subjects>
4 <Resources>
5 <Resource><ResourceMatch MatchId="equal">
7 <AttributeValue>demo:5</AttributeValue>
8 <ResourceAttributeDesignator AttributeId="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 <AttributeValue>Dissemination</AttributeValue>
21 <ActionAttributeDesignator AttributeId="actionid"/>
22 </ActionMatch>
23 </Action>
24 </Actions>
25 </Target>
26 <Condition FunctionId="not">
27 <Apply FunctionId="at-least-one-member-of">
28 <SubjectAttributeDesignator AttributeId="loginid"/>
29 <Apply FunctionId="string-bag">
30 <AttributeValue>testuser1</AttributeValue>
31 <AttributeValue>testuser2</AttributeValue>
32 <AttributeValue>fedoraAdmin</AttributeValue>
33 </Apply>
34 </Apply>
35 </Condition>
36 </Rule>
37 <Rule RuleId="2" Effect="Permit"/>
38 </policy>
```

Figure 4. ACPC (**A**ccess **C**ontrol **P**olicy **C**hecker) Model

For example, a deny overrides algorithm determines to return Deny if any rule evaluation returns Deny or no rule is applicable. A first applicable algorithm determines to return what the evaluation of the first applicable rule returns. In general, an XACML policy specification may also include multiple policies, which are included with a container element called PolicySet. When a request can also be applied to multiple policies, a policy combining algorithm can also be specified in a similar way. Figure 4 shows an example XACML policy, which is revised and simplified from a sample Fedora[2] policy. This policy has one policy element which in turn contains two rules. The rule composition function is "first-applicable", whose meaning has been explained earlier. Lines 2-13 define the target of the policy, which indicates that this policy applies only 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 a "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.

V. PROPOSED FRAMEWORK

ACPC (**A**ccess **C**ontrol **P**olicy **C**hecker) is the proposed model for the automated testing the correctness of the Access Control policies. This model will work for the policies written in XACML and having two sections.

1. In first section we generate the sets of *Requests*
2. In second section we *check the correctness* of the Policies.

[1]Conditons of "Any Resource", and "Any Action" can be satisfied by any subject, resource, or action, respectively.

[2]http://www.fedora.info

The ACPC model of illustrated below in fig. 5 having two sections:

A. Request Generator

The framework receives a set of policies under test and outputs a set of tests (in the form of request) for policy authors to inspect for correctness. The framework consists of four major components: derivation, change-impact analysis, request generator, and request reduction. The key notion of the framework is to synthesize two versions of the policy under test in such a way that test coverage targets (e.g., certain policies, rules, or conditions) are encoded as the differences of the two synthesized versions. A change-impact analysis tool can then be leveraged to generate counterexamples to witness these differences, thus covering the test coverage targets. Based on the generated counterexamples, the framework generates tests (in the form of requests).

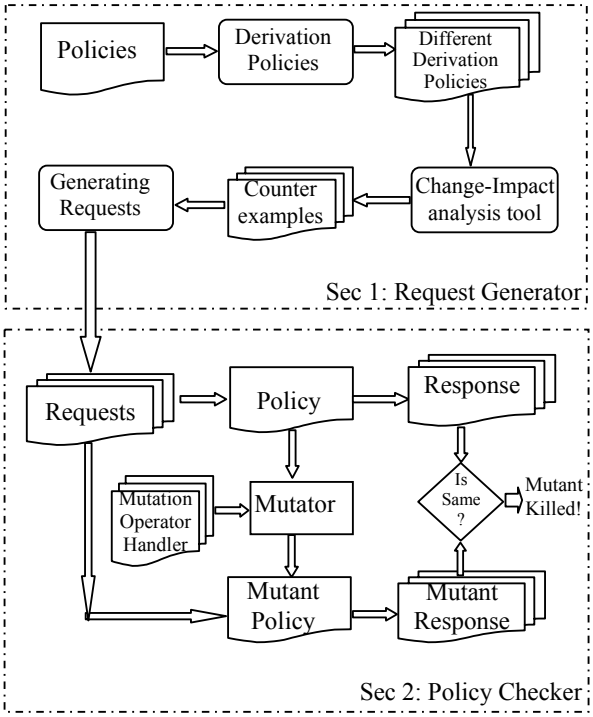

Figure 5. ACPC (**A**ccess **C**ontrol **P**olicy **C**hecker) Model

i. Derivation

Given the policy under test, the derivation component synthesizes the policy's versions, which are later fed to a change-impact analysis tool. We provide two variants of version synthesis below called one-to-empty and all-to-negate-one.

One-to-empty: For each rule r in p, the two derived versions are an empty policy and a policy that contains only r. If r is a permitting rule, the derived empty policy is an empty denying policy. If r is a denying rule, the derived empty policy is an empty permitting policy. The reason for this mechanism is as follows. Comparing a permitting rule r with an empty permitting policy will not help generate requests to cover r because no counterexamples are generated for these two versions. Similarly, comparing a denying rule r with an empty denying policy will not help generate requests to cover r. This derivation process is applied n times. So there are n pairs of policy versions synthesized for p.

All-to-negate-one: For each rule r in p, the two synthesized versions are p and p where the decision of r is negated. This process is applied n times so there are n pairs of policy versions synthesized for p. The preceding two variants are specifically developed for achieving high rule coverage. Because the coverage of a rule implies the coverage of the policy that contains the rule, our two variants also indirectly target at achieving high policy coverage. In principle, we can develop variants of derivation for achieving high condition coverage by negating each condition one at a time.

ii. Change–Impact Analysis

Given two versions of a policy, a change-impact analysis tool outputs counterexamples that lustrate semantic differences between the two policies. More specifically, each counterexample represents a request that evaluates to a different response when applied to the two policy versions. For example, a particular request r evaluates to permit for policy p but the same request evaluates to deny for policy p′. Change-impact analysis [5] is usually performed on mature policies that are undergoing maintenance or updates to avoid accidental injection of anomalies. In our case, we exploit the functionality of change-impact analysis to automatically generate access requests by iteratively manipulating the inputs to a change-impact analysis tool.

We use tool Margrave's API [5,8] to perform a change-impact analysis on the original policy and each of the policy versions. Based on the counterexample produced by Margrave, the request generator generates request. Exactly one request is generated from each version. Margrave package running in PLT scheme with drscheme[10] package and for generation counterexample CUDD tool [9] is necessary.

iii. Request Generator

Generating the test suite manually is very tedious and time consuming. To generate it automatically we are using the Margrave tool which takes the two versions as input and gives output in the form of counterexample. From the counterexample we generate the request. That part we will do with Java programming language.

B. Policy Checker

This section presents a model for access control policies and a set of mutation operators that implement that model. In general, 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 are modeling

things that could go wrong when constructing an access control policy. We use this model to measure the fault-detection effectiveness of automatic test generation and selection techniques. Any fault results in a semantic change in the policy but we broadly categorize faults as being semantic or syntactic as follows: syntactic faults are a result of simple typos whereas semantic faults are associated with the logical constructs of the policy language. This section of framework consists of three main components: Mutation operator handler, Mutant/original policy testing, Difference Checker (Comparer).

i. Mutation Operator Handler

This module is for making the faulty policies by the help of some effective mutation operators. Mutation operators [12, 13] describe modification rules for modifying access control policies to introduce faults into the policies [11]. We are using some five-six mutation operators which give better performance to analyze the correctness of the Access Control policies. We have different operators like PPT, PTF, RTT, RTF, RCT, RCF and CRE. The first six operators emulate syntactic faults because these mutation operators manipulate the predicates found in the target and condition elements. The last one emulates semantic faults because they manipulate the logic constructs of XACML policies.

***P**olicy **T**arget **T**rue* (PTT): Ensure that the policy is applied to all requests simply by removing the <Target> tag of each Policy element. The number of mutants created by this operator is equal to the number of Policy elements with a <Target> tag.

***P**olicy **T**arget **F**alse* (PTF): Ensure that the policy is never applied to a request by modifying the <Target> tag of each Policy element. The number of mutants created by this operator is equal to the number of Policy elements.

***R**ule **T**arget **T**rue* (RTT): Ensure that the rule is applied to all requests simply by removing the <Target> tag of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements with a <Target> tag.

***R**ule **T**arget **F**alse* (RTF): Ensure that the rule is never applied to a request by modifying the <Target> tag of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements.

***R**ule **C**ondition **T**rue* (RCT): Ensure that the condition always evaluates to True simply by removing the condition of each Rule element. The number of mutants created by this operator is equal to the number of Rule elements with a <Condition> tag.

***R**ule **C**ondition **F**alse* (RCF): Ensure that the condition always evaluates to False by manipulating the condition value or the condition function. The number of mutants created by this operator is equal to the number of Rule elements.

***C**hange **R**ule **E**ffect* (CRE): Invert each rule's Effect by changing Permit to Deny or Deny to Permit. The number of mutants created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants unless a rule is unreachable, a strong indication of an error in the policy specification.

These operators will pass in to Mutant module which take the original policy and convert it in to mutant policy. This module we are implementing in Java programming language.

ii. Mutant/Original Policy Testing

The Mutant and the original policies are passing through the analyzer called Oasis XACML, which takes the request and policy i.e. Mutant and original one for getting responses. The response we will take from the oasis XACML tool and do further process. So it will give the respective responses, when we provide request with policy.

iii. Difference Checker

We prepare this module in Java programming language, which take the responses of both original policy and mutant policy and evaluate weather both response is different or same. If the responses are different we say "mutant killed" and "mutant alive" otherwise.

VI. VALIDATING THE FRAMEWORK

To understand the working of ACPC we will take an example of sample policy having the following detail: Subject →Faculty, Resources → ExternalGrades, InternalGrades, Actions → Assign, View, with one rule having the 'Permit' effect.

We can find this example in the Margrave's example folder after installing the Margrave tool, named as RSP_Faculty.xml. Let we first start with First section Request Generation:

i. Derivation : In this module we have two way to derivate the original policy first one is One-to-empty, for that we change the rule effect 'Permit' to 'Deny' and make that rule empty. Second one is All-to–negate-one, for that we change the rule effect 'Permit to 'Deny'. In this way for that particular example we get to Derived version of original policy.

ii. Change-Impact Analysis: For analysis the changes we have tool call Margrave in that we use the PLT scheme called Drscheme. By the help of Drscheme we get the counterexample, for this particular example the counterexample:

1:/Action, command, View/
2:/Action, command, Assign/
3:/Resource, resourceclass,InternalGrades/
4:/Resource, resourceclass,ExternalGrades/
5:/Subject, role, Faculty/
12345
{
01011 P>D
01101 P>D

10011 P>D
10101 P>D
}

Here 1 shows presence of the attribute and 0 shows the absence in the counterexample P>D shows that rule effect changed from 'Permit' to 'Deny'

iii. Request Generator: For the help of counterexample, we have different request sets like:

Subject: Faculty
Resource: ExternalGrades
Action: Assign and
Subject: Faculty
Resource: InternalGrades
Action: Assign etc.

After generating the request sets we provide these request sets to the second section called policy checker:

i. Mutation operator handler: With the help of different operators mentioned earlier we can make different mutant policies.

ii. Mutant/original policy testing: we do the testing in this way, first we find out the response of the original policies and mutant policies by supplying the policy and response in XACML like: xacml_demo RPS_Fculty.xml Request1.xml in command line (with syntax xacml_demo <policy_file_name> request_file_name>) for original policy likewise for mutant policy with same request, and repeat this for different mutant policies and for different request sets.

iii. Difference checker: In this module we just check the mutant policy response and original policy response if these two are different than we say that the mutation has been killed otherwise not killed. The percentage of killed mutation shows the percentage of correctness of the policy.

VII. CONCLUSION

We have proposed a Model called ACPC which is used to check the correctness of the Access control policies and developed an automated mutation testing framework that implements that model. In this framework, we have defined a set of mutation operators. We have implemented a mutator that generates a number of mutant policies based on the defined mutation operators. We evaluate each request in a given request set on both the original policy and a mutant policy. The request evaluation produces two responses for the request based on the original policy and the mutant policy, respectively. If these two responses are different, then we determine that the mutant policy is killed by the request. We have also leveraged a change-impact analysis tool to detect equivalent mutants among generated mutants. We have conducted an experiment on various XACML policies to evaluate the mutation operators as well as request generation and selection techniques in terms of fault-detection capabilities.

REFERENCES

[1]. OASIS eXtensible Access Control Markup Language (XACML). 2005.

[2]. Sun's XACML implementation. 2005.

[3]. 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.

[4]. R. Sandlhu and P. Samarati. Access control: Principles and practice. IEEE Comm., pages 2-10, Sept. 1994

[5]. 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.

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

[7]. 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.

[8]. Margrave's API version 2.

[9]. F. Somenzi. CUDD: The CU decision diagram package.

[10]. PLT scheme with Drscheme package.

[11]. L. J. Morell. A theory of fault-based testing. IEEE Trans.Softw. Eng., 16(8):844–857, 1990.

[12]. A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C.Zapf. An experimental determination of sufficient mutation operators. ACM Transactions on Software Engineering Methodology, 5:99, April 1996.

[13]. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, pages 45–55, October 2000.

[14]. 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.

[15]. M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. ACM Conference on Computer and Communication Security*, pages 87– 96, Athens, Greece, November 2000.

[16]. N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. 8th International Conference on Information.*

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

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