Multiple-Implementation Testing for XACML Implementations

Nuo Li[1,2] JeeHyun Hwang[1] Tao Xie[1]
[1] Department of Computer Science, North Carolina State University, NC 27606, USA
[2] School of Computer Science and Engineering, Beihang University, Beijing 100083, China
{nli3,jhwang4}@ncsu.edu xie@csc.ncsu.edu

ABSTRACT

Many Web applications enhance their security via access-control systems. XACML is a standardized policy language, which has been widely used in access-control systems. In an XACML-based access-control system, policies, requests, and responses are written in XACML. An XACML implementation implements XACML functionalities to validate XACML requests against XACML policies. To ensure the quality of an XACML-based access-control system, we need an effective means to test whether the XACML implementation correctly implements XACML functionalities. The test inputs of an XACML implementation are XACML policies and requests. The test outputs are XACML responses. This paper proposes an approach to detect defects in XACML implementations via observing the behaviors of different XACML implementations for the same test inputs. As XACML has been widely used, we can collect different XACML implementations, and test them with the same XACML polices and requests to observe whether the different implementations produce different responses. Based on the analysis of different responses, we can detect defects in different XACML implementations. We show the feasibility of the proposed approach with a preliminary study on three XACML implementations.

Categories and Subject Descriptors

D2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Testing, Security

Keywords

XACML, access control policy, policy decision point, multiple-implementation testing

TAV-WEB – Workshop on Testing, Analysis and Verification of Web Software, July 21, 2008

1. INTRODUCTION

Access control provides an effective means to enhance the security of Web applications. An access-control system ensures that only authorized principals can access certain resources (such as data, programs, or services) of a Web application. As a result of the complexity introduced by hard-coding policies into programs, an increasing trend is to define policies in a standardized specification language such as XACML (eXtensible Access Control Markup Language) [15]. XACML enables to describe access control policies, requests, and responses. An XACML implementation is an access control policy evaluation engine, which can receive XACML requests and retrieve XACML policies applicable to the requests. Existing approaches focus on testing access control policies written in XACML [12, 10, 11], but no approach focuses on testing an XACML implementation.

To test XACML implementations, we need XACML policies and requests as test inputs. The test outputs are XACML responses. However, test inputs of XACML implementations are XML files, which are complex to be automatically generated by traditional software testing tools. Moreover, without a test oracle, we cannot automatically determine test results to be passed or failed, and it is tedious for testers to manually determine test results. Fortunately, XACML is a kind of markup language, i.e., there is an XML schema to define the format of XACML policies, requests, and responses, and the XACML specification defines various standard functionalities, which should be implemented by all XACML implementations. In other words, the inputs and outputs of all XACML implementations take the same formats, and for a given test input, different XACML implementations should produce the same outputs. Therefore, we can apply multiple-implementation testing [9, 14, 17] to test XACML implementations. We test different XACML implementations with the same XACML policies and requests, and observe whether the XACML implementations produce the same responses. For a pair of policy and request, if an XACML implementation produces a response different from the majority of other responses, we determine that this XACML implementation does not evaluate this request correctly. To get the test inputs for testing XACML implementations, we can use XML-generation tools, such as TAXI [8]. XML-generation tools accept the XACML schema as input and automatically generate XACML policies and requests. However, the XACML schema defines only the format of XACML policies, requests, and responses. The XACML policies and requests generated by XML-generation tools have a low chance to expose whether an XACML implementation implements certain XACML functionalities correctly, because the XACML schema does not define XACML functionalities.

In this paper, we propose an approach to generate test inputs and determine test results for XACML-implementation testing. We

automatically synthesize XACML policies based on a set of predefined policy templates. Each policy template focuses on a particular XACML functionality. We next automatically generate requests for each policy. To automatically determine test results, we apply multiple-implementation testing to test XACML implementations. As XACML has been widely used [3], we can collect different XACML implementations, and test them with the same XACML polices and requests to observe whether the different implementations produce different responses. Based on the analysis of different responses, we can detect defects in different XACML implementations.

We conduct a feasibility study using three different XACML implementations (Sun XACML 1.2 [5], XACML.NET 0.7 [4], and Parthenon XACML 1.1.1 [2]) with 374 pairs of XACML policies and requests. Each pair of policy and request contains a particular XACML standard functionality. Among these three XACML implementations, XACML.NET fails in supporting 34 of the functionalities (since XACML.NET implements a previous version of the used XACML standard functionalities), and Sun XACML 1.2 fails in supporting 11 of the functionalities.

The rest of this paper is structured as follows. Section 2 explains the background. Section 3 illustrates our approach through an example. Section 4 explains our approach in detail. Section 5 presents a feasibility study. Section 6 gives further discussion. Section 7 discusses related work, and Section 8 concludes the paper.

2. BACKGROUND

This section introduces the background information on XACML (the language that describes our test inputs and outputs), XACML implementations (the systems under test), and multiple-implementation testing (the approach that we use to determine test results).

2.1 XACML

XACML [15] is an XML-based language used to describe policies, requests, and responses for access control policies. XACML 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. XACML is approved by OASIS (Organization for the Advancement of Structured Information Standards) [6], and used in a number of commercial products, such as the products of BEA Systems, IBM, and Sun Microsystems [3].

The five basic elements of XACML policies are PolicySet, Policy, Rule, Target, and Condition. A policyset is simply a container that holds other policies or policysets. A policy is expressed through a set of rules. With multiple policysets, policies, and rules, XACML must have a way to reconcile conflicting rules. A collection of combining algorithms serves the function of reconciling conflicting rules. 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 [5].

To aid in matching requests with appropriate policies or rules, XACML provides a target, 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 policyset 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. A request is formed with attribute values that are compared to attribute values in a policy to make the access decisions.

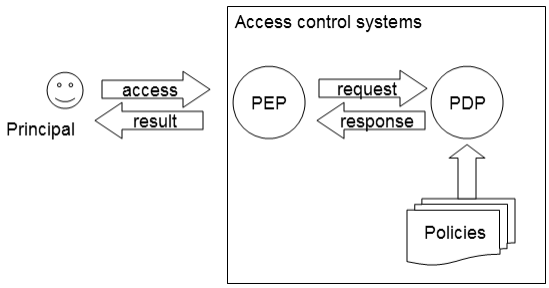

Figure 1: Overview of access-control system

2.2 XACML Implementation

Figure 1 presents an overview of a typical access-control system. A principal sends an access request to a Policy Enforcement Point (PEP) in an access-control system. The access request from the principal indicates what resources the principal wants to access. The PEP forms a formalized request to describe the principal's attributes to other components in the access-control system. The PEP sends this formalized request to a Policy Decision Point (PDP). Based on the attributes of the request, PDP selects a policy from a set of policies, which are pre-stored in the access-control system. PDP evaluates the request against the selected policy to generate a response. The response determines whether the principal is authorized. Finally, PDP returns the response to PEP, and PEP allows or denies the access of the principal based on the response. An XACML implementation consists of mainly a PEP and a PDP. The requests, policies, and responses, which are evaluated in an XACML implementation, should be written in XACML.

2.3 Multiple-Implementation Testing

Chen and Avizienis [9] introduced N-version programming. Based on the practice of N-version programming, project managers may ask more than one team to develop the same program independently to improve the reliability of software operation. Using N-version programming, testers test different versions of the same program, which are developed by different teams, with the same test inputs and vote each program based on whether the outputs are the same as the majority outputs of these programs. In this way, testers can detect defects in different versions. McKeeman [14] proposed differential testing for testing several implementations of the same functionality. Specifically, his differential testing focuses on testing different implementations of C compilers. Tsai et al. [17] applied a group testing technique on Web service testing. Their approach tests different Web services, which have the same specification, the same business logic, and the same input and internal states. They test such Web services with the same test inputs and rank the Web services under test based on whether they produce the same or close outputs as the majority Web services under test.

All of the preceding approaches test multiple implementations of the same program with the same test inputs, and then observe whether some implementations produce outputs that are different from the other implementations. In our approach, we use multiple-implementation testing to refer to such an approach. Without requiring test oracles, multiple-implementation testing provides us a means to alleviate the burden of manually determining test results.

```
 1<Policy PolicyId="test" RuleCombiningAlgId="deny-overrides">
 2 <Target>
 3   <Subjects> <AnySubject/> </Subjects>
 4   <Resources><AnyResource/> </Resources>
 5   <Actions> <AnyAction/> </Actions>
 6 </Target>
 7 <Rule RuleId="1" Effect="Permit">
 8  <Target>
 9   <Condition FunctionId="double-is-in">
10   <AttributeValue>5.55</AttributeValue>
11   </Condition>
12  </Target>
13 </Rule>
14</policy>
```

Figure 2: An example XACML policy

```
 1<Request>
 2  <Subject>
 3   <AttributeValue>Julius Hibbert</AttributeValue>
 4   <AttributeValue>5.5</AttributeValue>
 5   <AttributeValue>5.55</AttributeValue>
 6  </Subject>
 7  <Resource>
 8   <AttributeValue>BartSimpson</AttributeValue>
 9  </Resource>
10  <Action>
11   <AttributeValue>read</AttributeValue>
12  </Action>
13</Request>
```

Figure 3: An example XACML request

3. EXAMPLE

This section describes our approach through an illustrative example. Consider that we develop an XACML PDP that implements `double-is-in` function and we want to test whether the PDP correctly implements this function.

Figures 2 illustrates an example XACML policy involving the `double-is-in` function. This example is adapted from the XACML conformance test suite [1] that is used to test whether an XACML PDP is correctly implemented with regards to an XACML specification. In Figure 2, the policy uses the `deny-overrides` algorithm, which determines to return `Deny` if any rule evaluation results in `Deny` or no rule is applicable. As the policy does not specify any restriction on its target elements by allowing any subject, resource, and action (Lines 3-5 in Figure 2), the policy is applicable to any request. There is only one rule in the policy. Lines 7-13 in Figure 2 define the (permit) rule, whose meaning is that any subject can do any action on any resource under the condition that a request includes an attribute (double) value 5.55 (shown as `AttributeValue`). The XACML standard `double-is-in` function is used to specify the condition.

Figure 3 shows a request with one subject (`Julius Hibbert` with attribute values 5.5 and 5.55), one resource (`BartSimpson`), and one action (`read`).

To test whether our XACML PDP implements the XACML `double-is-in` function correctly, we use the preceding policy-request pair as test inputs. As one of the `AttributeValue` of the request is 5.55, which is the same as the `AttributeValue` of the rule in the policy, the expected response is `Permit`. Figure 4 presents the corresponding XACML response.

```
1<Response>
2 <Result>
3   <Decision>Permit</Decision>
4 </Result>
5</Response>
```

Figure 4: An example XACML response

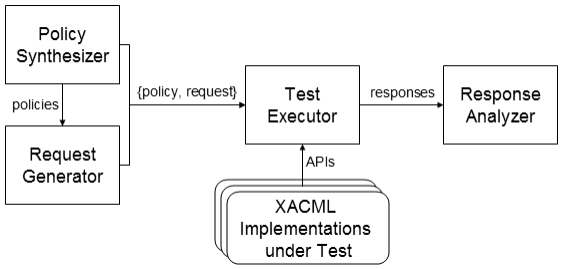

Figure 5: Overview of multiple-implementation testing for XACML implementations

However, usually when we test an XACML PDP with automatically generated test inputs, we have no test oracle and testers need to manually verify the XACML responses. As XACML has hundreds of functionalities and we need several test inputs for testing each functionality to cover different combinations of algorithms, targets, valid or invalid requests, etc., the manual verification of test results is tedious and cumbersome.

To automatically determine the test results, we collect different XACML implementations and invoke their PDP APIs and our PDP API with the preceding policy-request pair. We next automatically extract the value of the decision in each response, and compare these decision values. We use the majority decision values as the expected decision. If the decision value of our XACML PDP is different from the majority decision, we determine that our PDP does not implement the `double-is-in` function correctly.

4. APPROACH

Figure 5 shows a high-level overview of the components in our approach. Our approach consists of four components. First, the policy synthesizer synthesizes policies based on predefined policy templates. Each policy template focuses on a particular XACML functionality, such as the XACML `double-is-in` function explained in Section 3. Second, the request generator generates requests for each policy synthesized by the policy synthesizer. Third, the test executor invokes different XACML implementations with the same policy-request pairs. Fourth, the response analyzer detects the XACML implementations that generate different responses from the majority responses and identifies that these implementations do not implement certain XACML functionalities correctly.

4.1 Policy Synthesizer

To test an XACML implementation, we need XACML policies and requests that include different XACML functionalities to check whether the XACML implementation can generate correct responses. However, it is difficult to get a large number of real-life XACML policies as access control policies are often deemed confidential. Furthermore, if we generate XACML policies based on the XACML schema randomly, the effectiveness of testing can be low, because randomly generated XACML policies have a low possibility to cover different XACML functionalities.

To improve the effectiveness of testing and facilitate the test-result determination, we predefine XACML-policy templates, each of which focuses on one XACML functionality, and develop a policy synthesizer to automatically synthesize XACML policies based on the predefined XACML-policy templates. The XACML specification [15] provides the category of different XACML standard functionalities and we define an XACML-policy template as a customized and simplified XACML schema. For example, consider

the `double-is-in` function shown in Figure 2. We define an XACML-policy template including targets and rules, and explicitly define that the `FunctionId` of a condition for the target of a rule is `double-is-in`. For the other attributes, we predefine their candidate values. For example, a `RuleCombiningAlgId` has four candidate values: `deny-overrides`, `permit-overrides`, `first-applicable`, and `only-one-applicable`. When the policy synthesizer synthesizes policies, it synthesizes four policies, each of which selects a different candidate value for the `RuleCombiningAlgId`. In addition, the policy synthesizer composes different XACML-policy templates to construct policies that cover combinations of different XACML functionalities.

4.2 Request Generator

The request generator [13] randomly generates requests for a given policy. The request generator analyzes the policy under test and generates requests on demand by randomly selecting requests from the set of all combinations of attribute id-value pairs found in the policy. A particular request is represented as a vector of bits. The length of this vector is equal to the number of different attribute values found in the policyset targets, policy targets, rule targets, and rule conditions of the policy under test. Each attribute value appears in the request if its corresponding bit in the vector is 1; otherwise, the value is not present. Each request is generated by setting each bit in the vector to 0 or 1 with a probability of 0.5. The number of randomly generated requests can be configured by the user and the configured number can be considerably smaller than the total number of combinations.

To help achieve adequate coverage with a small set of random requests, we modify the random test generation algorithm to ensure that each bit was set to 0 and 1 at least once. In particular, we explicitly set the i^{th} bit to 1 for the first n generated requests where $i = 1, 2, ...n$. Similarly, for the next n requests, we explicitly set the $(i - n)^{th}$ bit to 0. This improved algorithm guarantees that each attribute value is present and absent at least once as long as the number of randomly generated requests is greater than $2n$.

For this component, we reuse the tool developed in our previous work [13] for generating XACML requests based on policies.

4.3 Test Executor

We collect different XACML implementations and the test executor invokes all implementations with each policy and request pair. XACML implementations provide APIs for developers to build and customize PDPs, PEPs, or other related components in an access-control system. The test executor accepts the policies and requests generated by the policy synthesizer and the request generator, and invokes a particular API of the different XACML implementations with each policy-request pair. The test executor next collects the generated XACML responses. Moreover, the test executor records what XACML response each system under test produces for a certain policy-request pair.

4.4 Response Analyzer

The response analyzer analyzes the responses collected by the test executor and computes the most common response for each policy-request pair. Each XACML response contains a `Decision` node (shown in Figure 4), whose value implies the result of evaluating a request against a policy. There are four possible values for a decision node: `Permit`, `Deny`, `Indeterminate`, and `NotApplicable`. The `Permit` decision node describes that a requested access is permitted. The `Deny` decision node describes that a requested access is denied. The `Indeterminate` decision node describes that a PDP is unable to evaluate the requested access. Reasons for such inability include missing attributes, network errors while retrieving policies, division by zero during policy evaluation, syntax errors in the decision request or in the policy, etc. The `NotApplicable` decision node describes that a PDP does not have any policy that applies to a request.

The response analyzer extracts the decision values in the XACML responses generated by different XACML implementations for the same policy-request pairs and compares the decision values. The response analyzer treats the most common value as the expected value. If an XACML implementation produces a response with the same decision value as the expected value, the test result for that XACML implementation is "passed". If an XACML implementation produces a response with a different decision value with the expected value, the test result for that XACML implementation is "failed". (In the case that the response analyzer cannot determine the most common value, for example, when three XACML implementations produce three different responses, the response analyzer reports "uncertain" as the test results for each XACML implementation.) In addition, as the test executor records the mapping among XACML implementations, XACML responses, and policy-request pairs, when the response analyzer detects a failed test result, the response analyzer can detect which policy-request pair induces an XACML implementation to produce an incorrect XACML response. As each policy focuses on a particular XACML functionality, the response analyzer reports which XACML functionality is not correctly implemented by the XACML implementation.

5. FEASIBILITY STUDY

We conduct a feasibility study on three XACML implementations: Sun XACML 1.2 [5], XACML.NET 0.7 [4], and Parthenon XACML 1.1.1 [2]. This study shows the preliminary results of our approach. The objective of this feasibility study is to show that, without expected responses, we can use policy-request pairs to detect which XACML implementations fail in supporting which XACML functionalities. As the policy synthesizer is still under development, this feasibility study uses the policy-request pairs provided in an XACML conformance test suite [1]. The test cases in the conformance test suite have been classified into different categories based on which XACML functionality is included by each test case. Although the conformance test suite contains policy-request pairs and expected responses, in the feasibility study, we use only the policy-request pairs but discard the expected responses. Our future work plans to conduct a full evaluation of our approach, including the policy synthesizer, request generator, test executor, and response analyzer.

5.1 Instrumentation

In the XACML conformance test suite, the test cases, i.e., XACML policies, XACML requests, and XACML responses, are divided into those that exercise mandatory-to-implement functionalities and those that exercise optional functionalities. Furthermore, the test cases for mandatory-to-implement functionalities are classified into five types, i.e., attribute references, target matching, function evaluation, combining algorithms, and schema components, based on the XACML functionalities included in the policies.

In our test executor, we invoke the PDP APIs of these three XACML implementations with each policy-request pair. Sun XACML 1.2 provides sample code, a class named `SimplePDP`, to demonstrate how to construct an actual PDP object and evaluate requests against given policies. XACML.NET 0.7 and Parthenon XACML 1.1.1 provide command-line drivers to invoke their PDPs. Therefore, our test driver invokes the `SimplePDP` of Sun XACML 1.2 and the command-line drivers of XACML.NET and Parthenon XA-

CML with each policy-request pair as their parameters.

We set a threshold of 0.5 for the response analyzer, i.e., the response analyzer sets a decision value (D) as the expected value, if two of the XACML implementations produce the same decision value D. Otherwise, the response analyzer determines the test result as uncertain.

5.2 Results

The XACML conformance test suite provides 374 policy-request pairs. As some of the policy-request pairs include invalid syntax or some functionalities that are not implemented by the XACML implementations under test, the XACML implementations do not produce responses for some policy-request pairs. In such cases, some of the XACML implementations throw out exceptions, and some of the XACML implementations stop running without any output. If an XACML implementation does not produce a response for a policy-request pair, the response analyzer records the decision value as N/A. For the 374 policy-request pairs, Sun XACML 1.2 produces 1 N/A responses, XACML.NET produces 35 N/A responses, and Parthenon XACML produces 2 N/A responses. For each PDP, the decision value in its response has five candidate values: Permit, Deny, NotApplicable, Indeterminate, and N/A. The detailed experimental data is listed in our project webpage[1].

The response analyzer compared the decision values generated by the three XACML implementations for each policy-request pair, and found that 47 policy-request pairs produce inconsistent decisions. Based on the analysis of inconsistent decisions, the response analyzer determines that 34 test results of XACML.NET are failed, i.e., XACML.NET does not correctly deal with 34 XACML functionalities involved in the conformance test suite, and 11 test results of Sun XACML 1.2 are failed. In addition, there are 2 test results that are uncertain, i.e., the three XACML implementations produce three different responses.

Table 1 presents the category of the failed tests for each XACML implementation. In Table 1, Column 1 lists the six categories of functionalities involved in the conformance test suite. The first five categories belong to the mandatory-to-implement XACML functionalities, and the last one is not mandatory-to-implement, but is normative when implemented. The "attributes references" category represents those tests that exercise referencing of attribute values in a request by a policy. The "target matching" category stands for those tests that exercise various forms of target matching. The "function evaluation" category represents those tests that exercise each of the mandatory-to-implement functionalities. The "combining algorithms" category represents those tests that exercise each of the mandatory combining algorithms. The "schema components" category represents the tests for certain elements of the schema not exercised by the preceding categories of test cases. Columns 2-4 of Table 1 list that, for each category of functionalities, the number of failed tests of each XACML implementation. For example, Sun XACML 1.2 does not correctly evaluate 2 policy-request pairs that contain attributes references, 3 policy-request pairs that contain function evaluation, and 6 policy-request pairs that contain optional functionalities. The "2(u)" in Table 1 represents that there are two uncertain test results for the policy-request pairs that contain schema components.

We observe that XACML.NET fails in producing responses for 35 policy-request pairs, which are much more than the other two XACML implementations (Sun XACML 1.2 produces 1 N/A decisions and Parthenon XACML produces 2 N/A decisions). This result is mainly because XACML.NET conforms to XACML 1.0, but the conformance test suite used in this study conforms to XACML 1.1. The 35 N/A decisions cover all of the six categories of XACML functionalities listed in Table 1. Therefore, we determine that, among the three XACML implementations under test, XACML.NET supports a smaller set of XACML functionalities than the other two XACML implementations.

[1] http://ase.csc.ncsu.edu/projects/multixacmltest/

Functionality category	Sun XACML 1.2	XACML .NET	Parthenon
Attributes references	2	4	0
Target matching	0	8	0
Function evaluation	3	2	0
Combining algorithms	0	11	0
Schema components	2(u)	2(u)	2(u)
Optional functionalities	6	8	0

Table 1: Category of the failed tests for each XACML implementations

Excluding the inconsistent decisions that are induced by the N/A results generated by XACML.NET and the uncertain results, Table 2 lists the details of the other inconsistent decisions. In Table 2, each row contains inconsistent decisions produced by the three XACML implementations for a particular policy-request pair. Column 1 lists the categories of functionalities included in the policy-request pairs. Columns 2-4 list the decision value of each XACML implementation.

To ensure the security of a system, if the response from a PDP is NotApplicable or Indeterminate, the request is usually denied. In other words, all of Deny, NotApplicable, and Indeterminate can be treated as opposite decisions of Permit. In this way, we detect that XACML.NET produces an opposite decision of Sun XACML 1.2 and Parthenon for a mandatory-to-implement functionality, whose category is "function evaluation", and Sun XACML 1.2 produces opposite decisions of XACML.NET and Parthenon for five optional functionalities.

If the decision of an XACML implementation is N/A, the test result is considered as failed with the following reason. When an XACML PDP cannot evaluate a request, the XACML PDP should produce NotApplicable or Indeterminate decisions instead of producing no decision. Therefore, an N/A decision reflects that the XACML PDP fails. Based on such an analysis, in Table 2, Sun XACML 1.2 fails for an attribute-references functionality.

6. DISCUSSION

The feasibility study uses policy-request pairs in a conformance test suite to show that, without expected responses, we can detect which XACML implementations fail in supporting which XACML functionalities. Therefore, although XACML.NET implements a different version of the XACML specification from which specification version Sun XACML 1.2 and Parthenon XACML 1.1.1 implement, we include XACML.NET as a subject in the feasibility study. Based on the test result, we can determine that, among the XACML functionalities used in the conformance test suite, XACML-.NET implements fewer functionalities than Sun XACML 1.2 and Parthenon XACML 1.1.1.

In the future, when we implement our policy synthesizer, we shall evaluate our approach through testing different XACML implementations with the policies generated by our policy synthesizer. The conformance test suite covers most XACML functionalities, but each policy in the conformance test suite focuses on

Category	Sun XACML 1.2	XACML .NET	Parthenon
Attribute references	NotApplicable	Indeterminate	Indeterminate
	N/A	Indeterminate	Indeterminate
Function evaluation	NotApplicable	Indeterminate	Indeterminate
	NotApplicable	Indeterminate	Indeterminate
	NotApplicable	Indeterminate	Indeterminate
	Permit	NotApplicable	Permit
Optional functionalities	Indeterminate	Permit	Permit
	Indeterminate	Deny	Deny
	NotApplicable	Permit	Permit
	NotApplicable	Permit	Permit
	NotApplicable	Permit	Permit
	NotApplicable	Permit	Permit

Table 2: Details of inconsistent decisions

one XACML functionality. Our policy synthesizer composes different XACML-policy templates to construct policies. The policies that composes different XACML functionalities may detect defects that cannot be detected by a simple policy. However, the combination of XACML functionalities increases the number of test cases and the difficulty of debugging. To alleviate the efforts of debugging, we may compare two similar policy-request pairs that make an XACML implementation produce different responses. We then analyze the difference between the two policy-request pairs. In this way, when debugging, we can focus on the different parts in the policy-request pairs.

Our approach is based on a set of predefined policy templates that help the response analyzer automatically determine which XACML implementation fails in dealing with which XACML functionalities. However, predefining policy templates requires manual efforts. A possible solution to avoid such manual efforts is that we automatically generate XACML policies based on the XACML schema and then we use the existing classified policies, whose initial set shall be the conformance test suite used in Section 5, to automatically classify the newly generated policies into an existing category.

7. RELATED WORK

Existing work on XACML testing focuses on testing XACML policies [10, 12]. Our work focuses on testing XACML implementations and uses XACML policies and requests as test inputs. As the inputs of an XACML implementation are XML files, an XACML implementation is a kind of XML-input application. We classify testing of XML-input applications into three groups: specification-based testing, mutation-based testing, and multiple-implementation testing.

Specification-based testing of XML-input applications generates test inputs based on the definition of acceptable inputs such as XML schema of an application. Bertolino et al. [8] implement a tool, called TAXI, for XML-based testing. Given an XML Schema, TAXI automatically generates XML instances. TAXI implements the XML-based partition testing approach and provides a set of weighted test strategies to guide the systematic derivation of instances. Bai and Dong [7] propose an approach of WSDL-based test-case generation for Web-service testing. They parse and transform a WSDL file into a structured DOM tree, and then generate test cases from two aspects: test data and test operations. Test data is generated by analyzing the message data types according to standard XML schema syntax. Test operations are generated based on an operation-dependency analysis. However, enumerating all possible inputs based on an XML schema provides exhaustive testing but is infeasible in practice. In addition, without test oracles, it is tedious to check all the test results manually.

Mutation-based testing of XML-input applications generates test inputs based on the definition of acceptable inputs for an application under test, and mutates the definition to generate invalid inputs to test the robustness of the application under test. Offutt and Xu [16] present an approach to test Web services based on data perturbation, i.e., modifying XML messages based on rules defined on the message grammars. They implement two types of data perturbation: data value perturbation and interaction perturbation. The data value perturbation generates test inputs of invalid data types. The interaction perturbation generates invalid sequences of messages among multiple Web services. Our approach complements mutation-based testing: we can use mutation-based testing to generate invalid policies or requests, and use multiple-implementation testing to automatically determine test results to be passed or failed.

Multiple-implementation testing has been used by Tsai et al. [17] on Web-service testing. To save testing time, they initially test a subset of Web service implementations randomly selected from the set of all Web service implementations to be tested. They test the subset of Web service implementations by the same test cases, detect defective Web service implementations based on their different outputs, and determine expected outputs. They next rank the test cases according to their defect-detection capacities and test all Web service implementations to be tested by the test cases with high defect-detection capacities. Different from their approach, we focus on testing XACML implementations and generate test inputs based on a set of predefined XACML policy templates. As each template includes a particular XACML functionality, when we detect different outputs, we can determine which XACML functionality is not implemented by an XACML implementation correctly.

8. CONCLUSION

We have proposed an approach to test XACML implementations and automatically determine the test results. We first synthesize XACML policies based on a set of predefined policy templates, each of which focuses on a particular XACML functionality. We next generate requests for each policy. To automatically determine expected responses, we test different XACML implementations with the same polices and requests to observe whether the different XACML implementations produce different responses. Based on the analysis of different responses, we detect the XACML implementations that do not implement certain XACML functionalities correctly.

We conducted a feasibility study using three different XACML implementations. We use 374 pairs of XACML policies and requests to test the three XACML implementations. Each pair of policy and request contains a particular XACML standard functionality. Among the three XACML implementations, we detect that XACML.NET fails in supporting 34 of those functionalities (since XACML.NET implements a previous version of the used XACML standard functionalities), and Sun XACML 1.2 fails in supporting 11 of those functionalities.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579.

9. REFERENCES

[1] XACML 1.1 Committee Specification Conformance Tests, 2002. http://www.oasis-open.org/committees/xacml/ConformanceTests/ConformanceTests.html.

[2] Parthenon Policy Tester, 2005. http://www.parthcomp.com/xacml_toolkit.html.

[3] XACML 2.0 Approved as OASIS Standard, 2005. http://xml.coverpages.org/XACMLv20-Standard.html.

[4] XACML.NET, 2005. http://mvpos.sourceforge.net/.

[5] Sun's XACML Implementation, 2006. http://sunxacml.sourceforge.net/.

[6] Organization for the Advancement of Structured Information Standards, 2008. http://www.oasis-open.org/home/index.php.

[7] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for Web services testing. In *Proc. SOSE*, pages 215–220, 2005.

[8] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. TAXI–a tool for XML-based testing. In *Proc. ICSE COMPANION*, pages 53–54, 2007.

[9] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. FTCS*, pages 3–9, 1978.

[10] V. C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance checking of access control policies specified in XACML. In *Proc. IWSSE*, pages 275–280, July 2007.

[11] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *Proc. SESS*, pages 5–11, 2007.

[12] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. WWW*, pages 667–676, 2007.

[13] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. ICICS*, pages 139–158, 2006.

[14] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[15] OASIS. *eXtensible Access Control Markup Language (XACML)*. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[16] J. Offutt and W. Xu. Generating test cases for Web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[17] W.-T. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou, and X. Wei. Adaptive testing, oracle generation, and test case ranking for Web services. In *Proc. COMPSAC*, pages 101–106, 2005.