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

{nl3, 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. 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. Although we can automatically generate XACML policies and requests based on XACML schema, randomly generated XACML policies and requests have a low chance to expose defects of an XACML implementation. Furthermore, without test oracles, we cannot automatically determine test results to be passed or failed. This paper proposes an approach to detect defects in XACML implementations via observing the behaviors of different XACML implementations for the same test inputs. In our approach, a policy synthesizer synthesizes XACML policies based on the predefined policy templates, each of which focuses on a particular XACML functionality. A request generator next generates requests for each policy. To automatically determine test results, we apply multiple-implementation testing to test XACML implementations. As XACML has been widely used, we can collect different XACML implementations, and test them with the same XACML policies 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 evaluate our approach using four different XACML implementations with 374 pairs of XACML policies and requests. Each pair of policy and request contains a particular XACML standard functionality. Among these four XACML implementations, our approach detects that Sun XACML 1.1 fails in supporting most of the 374 functionalities, XACML.NET fails in supporting 34 of the functionalities, and Sun XACML 1.2 fails in supporting 11 of the functionalities.

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

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) [17] and integrate the policies with applications through the use of a Policy Decision Point (PDP), a component that evaluates requests against policies and makes policy decisions. XACML enables to describe access control policies, requests, and responses. When an XACML PDP receives an XACML request, the XACML PDP analyzes the request, retrieves applicable XACML policies to the request, and validates the request with such policies. Existing approaches focus on testing access control policies that are written in XACML [14, 10, 13, 11], but only few approach focuses on testing XACML implementations, i.e., the program that receives XACML requests, evaluates XACML requests against XACML policies, and generates XACML responses.

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 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, 16, 19] 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. A policy synthesizer synthesizes XACML policies based on the predefined policy templates. A request generator next generates requests for each policy. To automatically determine test results, we apply multiple-implementation testing to test XACML implementations. As XACML has been widely used [2], we can collect different XACML implementations, and test them with the same XACML policies 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 evaluate our approach using four different XACML implementations (Sun XACML 1.1 [5], Sun XACML 1.2 [5], XACML.NET 0.7 [4], and Parthenon XACML 1.1.1 [3]) with 374 pairs of XACML policies and requests. Each pair of policy and request contains a particular XACML standard functionality. Among these four XACML implementations, our approach detects that Sun XACML 1.1 fails in supporting most of the 374 functionalities, XACML.NET fails in supporting 34 of the 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 the evaluation of our approach. 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 [17] is an XML-based general-purpose 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 (Advancing Open Standards for the Information Society) [6], and used in a number of commercial products, such as the products of BEA Systems, IBM, and Sun Microsystems [2].

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. Attribute values from a request are resolved through two mechanisms: the AttributeDesignator and the AttributeSelector. 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.

2.2 XACML Implementation

Figure 1 presents an overview of a typical access-control system. A principal sends a request to a Policy Enforcement Point (PEP) in a 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 [16] used the name 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. [19] applied 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 with 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.

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.

Figure 4: An example XACML response

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 improve the effectiveness of testing and facilitate the test-result determination, we predefine XACML-policy templates, each 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.

4. APPROACH

Figure 5 shows an 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 and identifies that these implementations does 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 all 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 an policy synthesizer to automatically synthesize XACML policies based on the predefined XACML-policy templates. The XACML specification [17] 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 a policy, the synthesizer randomly selects a value from the four candidate values for a RuleCombiningAlgId.

4.2 Request Generator

The request generator [15] 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 for generating XACML requests based on policies [15].

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 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, i.e., test oracle. If an XACML implementation produces a response with the same decision value as the test oracle, the test result for that XACML implementation is "passed". If an XACML implementation produces a response with a different decision value with the test oracle, the test result for that XACML implementation is "failed". (In the case that the response analyzer cannot determine the most common value, for example, three XACML implementations produce three different responses, the response analyzer reports "uncertain" as the test result 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 failing 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. PRELIMINARY RESULTS

This section presents the evaluation results. We test four XACML implementations: Sun XACML 1.1 [5], Sun XACML 1.2 [5], XACML.NET 0.7 [4], and Parthenon XACML 1.1.1 [3]. We test each XACML implementation with the policy-request pairs provided in an XACML conformance test suite [1].

The XACML conformance test suite is developed by the XACML Technical Committee. As the XACML Technical Committee classifies the conformance test suite into different categories and explicitly specifies which XACML functionality is included by each test case, we use the policies and requests provided in the conformance test suite to test the preceding four XACML implementations. To avoid analysis cost of request generation and evaluation in our preliminary evaluation, we do not use our request generator to generate a large number of requests for each policy in the conformance test suite but reuse the requests in the conformance test suite. The evaluation for our approach of request generation can be found in our previous work [15].

![Figure 5: Overview of multiple-implementation testing for XACML implementations](image-url)
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 four XACML implementations with each policy-request pair. Both Sun XACML 1.1 and Sun XACML 1.2 provide 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’s XACMLs and the command-line drivers of XACML.NET and Parthenon XACML 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 test oracle, if three or four of the XACML implementations produce the same decision value D, or two of the XACML implementations produce the same decision value D while the other two XACML implementations produce two different decision values. 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. If an XACML implementation does not generate a response for a policy-request pair, the response analyzer records the decision value as Empty. Sun XACML 1.1 and 1.2 generated 373 responses for the 374 policy-request pairs, XACML.NET generated 339 responses, and Parthenon XACML generated 372 responses. For each PDP, the decision value in its response has five candidate values: Permit, Deny, NotApplicable, Indeterminate, and Empty. The detailed experimental data is listed in our project webpage.

The response analyzer compared the decision values generated by the four XACML implementations for each policy-request pair, and found that 272 policy-request pairs produce different decisions. Based on the analysis of these different decisions, the response analyzer determines that 242 test results of Sun XACML 1.1 are failed, i.e., Sun XACML 1.1 does not correctly deal with 242 XACML functionalities included in the conformance test suite. Table 1 presents the summary of our evaluation results. In Table 1, Column 1 lists the five decision-value types, and Columns 2-5 list that, among the detected 272 inconsistent responses, how many responses with a particular decision value are produced by each XACML implementation. From Table 1, we notice that Sun XACML 1.1 generates 263 NotApplicable decisions and 0 Permit decisions, but the other three XACML implementations produce much fewer NotApplicable decisions (23, 1, and 13) and more Permit decisions (210, 214, and 215) than Sun XACML 1.1. We hypothesize that, because of the evolving of XACML, Sun XACML 1.1 does not support most of the XACML functionalities involved in the conformance test suite used in this experiment, and we decide to exclude Sun XACML 1.1 in our further analysis. As we exclude Sun XACML 1.1, we adjust the threshold to 2/3 for determining test oracles.

When we focus on the inconsistent responses among Sun XACML 1.2, XACML.NET 0.7, and Parthenon XACML 1.1.1, the response analyzer reported that 47 policy-request pairs produce inconsistent decisions. Based on the analysis of inconsistent decisions, the response analyzer determined that 43 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 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 2 presents the category of the failed tests for each XACML implementation. In Table 2, Column 1 lists the six categories of functionalities involved in each policy-request pair. 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 2 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 2 represents that there are two uncertain test results for the policy-request pairs that contain schema components.

From Table 1, we observe that XACML.NET fails in producing response for 35 policy-request pairs, which is much more than the other two XACML implementations (Sun XACML 1.2 and Parthenon XACML). The 35 Empty results cover all of the six categories of XACML functionalities listed in Table 2. 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.

Excluding the inconsistent decisions that are induced by the Empty results generated by XACML.NET and the uncertain results, Table 3 lists the details of the other inconsistent decisions. In Table 3, Column 1 lists the categories of functionalities included in the policy-request pairs. Columns 2-4 list the decision value of each

<table>
<thead>
<tr>
<th>Decision Value</th>
<th>Sun XACML 1.1</th>
<th>Sun XACML 1.2</th>
<th>XACML.NET</th>
<th>Parthenon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>1</td>
<td>1</td>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>NotApplicable</td>
<td>263</td>
<td>23</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>6</td>
<td>20</td>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td>Permit</td>
<td>0</td>
<td>210</td>
<td>214</td>
<td>215</td>
</tr>
<tr>
<td>Deny</td>
<td>2</td>
<td>18</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 1: Summary of the detected inconsistency among Sun XACML 1.1, Sun XACML 1.2, XACML.NET 0.7, and Parthenon XACML 1.1.1
<table>
<thead>
<tr>
<th>Functionality category</th>
<th>Sun XACML 1.2</th>
<th>XACML .NET</th>
<th>Parthenon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes references</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Target matching</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Function evaluation</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Combining algorithms</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Schema components</td>
<td>2(u)</td>
<td>2(u)</td>
<td>2(u)</td>
</tr>
<tr>
<td>Optional functionalities</td>
<td>6</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Category of the failed tests for each XACML implementations

<table>
<thead>
<tr>
<th>Category</th>
<th>Sun XACML 1.2</th>
<th>XACML .NET</th>
<th>Parthenon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute references</td>
<td>NotApplicable</td>
<td>Indeterminate</td>
<td>Indeterminate</td>
</tr>
<tr>
<td>Function evaluation</td>
<td>NotApplicable</td>
<td>Indeterminate</td>
<td>Indeterminate</td>
</tr>
<tr>
<td>Optional functionalities</td>
<td>Indeterminate</td>
<td>Permit</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>Indeterminate</td>
<td>Deny</td>
<td>Deny</td>
</tr>
<tr>
<td></td>
<td>NotApplicable</td>
<td>Permit</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>NotApplicable</td>
<td>Permit</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>NotApplicable</td>
<td>Permit</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>NotApplicable</td>
<td>Permit</td>
<td>Permit</td>
</tr>
</tbody>
</table>

Table 3: Details of inconsistent decisions

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. Specially, if an XACML implementation produces an Empty decision, the test result is failed. When an XACML PDP cannot evaluate a request, the XACML PDP produces NotApplicable or Indeterminate decisions. An Empty decision reflects that the XACML PDP does not work as expected. Based on such an analysis, in Table 3, Sun XACML 1.2 is failed for an attribute-references functionality.

6. DISCUSSION

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

Most of the existing work about XACML testing focuses on testing XACML policies [10, 12, 14]. 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 the 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 is different from other schema-based XML-generation tools as 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 automatic 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, to enumerate all possible inputs based on an XML schema is exhaustive but 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 [18] 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 is used in our approach. Tsai et al. [19] apply multiple-implementation testing on Web-service testing. To save testing time, they initially test a subset of Web services randomly selected from the set of all Web services to be tested. They test the subset of Web services by the same test cases, detect defective Web services based on their different outputs, and determine test oracles. They next rank the test cases according to their defect-detection capacities and test all Web services 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 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 test oracles, we test different XACML implementations with the same policies 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. Although the request generator generates requests for a policy randomly, our approach is different from randomly generating policies and requests by XML-generation tool. As we predefine XACML functionalities in policy templates and generate policies based on the templates, the XACML functionalities are included in the generated policies. When the request generator generates requests based on the generated policies, the request generator guarantees that each attribute value is present and absent at least once.

In our evaluation, we tested four different XACML implementations with the XACML conformance test suite. Among these tested implementations, our approach detects that Sun XACML 1.1 fails in supporting most of the 374 XACML standard functionalities included in the XACML conformance test suite. XACML.NET fails in supporting 34 of those functionalities, and Sun XACML 1.2 fails in supporting 11 of those functionalities.

9. REFERENCES


