Fault-Based Testing of Combining Algorithms in XACML3.0 Policies

Dianxiang Xu, Ning Shen, Yunpeng Zhang
Department of Computer Science
Boise State University
Boise, ID 83725, USA
{dianxiangxu, ningshen, yunpengzhang}@boisestate.edu

***Abstract*— With the increasing complexity of software, new access control methods have emerged to deal with attribute-based authorization. As a standard language for attribute-based access control policies, XACML offers a number of rule and policy combining algorithms to meet different needs of policy composition. Due to their variety and complexity, however, it is not uncommon to apply combining algorithms incorrectly, which can lead to unauthorized access or denial of service. To solve this problem, this paper presents a fault-based testing approach for determining incorrect combining algorithms in XACML 3.0 policies. It exploits an efficient constraint solver to generate queries to which a given policy produces different responses than its combining algorithm-based mutants. Such queries can determine whether or not the given combining algorithm is used correctly. Our empirical studies using sizable XACML policies have demonstrated that our approach is effective.**

Keywords*— Combining algorithm, constraint solving, fault-based testing, test generation, XACML.

I. Introduction

In security-intensive software, access control is a fundamental mechanism for preventing malicious or accidental violation of security requirements by regulating user access to resources. An access control policy defines the conditions under which access to resources can be granted and to whom. Given an access request, it yields an access decision such as permit or deny. With the increasing complexity of software, access control methods have evolved from popular role-based access control to Attribute-Based Access Control (ABAC). ABAC enables fine-grained access control by combining various attributes of authorization elements into access control decisions. These attributes are predefined characteristics of subjects (e.g., job title and age), resources (e.g., data, programs, and networks), actions, and environments (e.g., current time and IP address) [7]. ABAC also facilitates collaborative policy administration within a large enterprise or across multiple organizations. In a large enterprise, for example, elements of authorization policies may be managed by different departments, such as the Information Technology department, Human Resources, the Legal department, and the Finance department [13]. Individual rules or policies are composed into a whole in order to make consistent access decisions.

XACML (eXtensible Access Control Markup Language) [13] is an OASIS standard for specifying ABAC policies in the XML format. To support flexible policy composition, XACML 3.0 provides 11 rule combining algorithms and 12 policy combining algorithms. A combining algorithm aims at rendering a single access decision by combining the decisions of individual access control rules or policies. Due to the variety of combining algorithms and subtle similarities between the combining algorithms, it is not uncommon to use them incorrectly when XACML3.0 policies are authored. A user may inadvertently select an incorrect combining algorithm or intentionally apply an incorrect combining algorithm due to misunderstanding. Furthermore, for certain rules (or policies), different combining algorithms can be functionally equivalent and result in the same response to every access request. In an evolving process of policy development and maintenance, however, a previously working combining algorithm may become incorrect after new rules or policies are added in a way that implicitly breaks the constraints on functional equivalence. Needless to say, incorrect combining algorithms in XACML policies can lead to devastating consequences, such as unauthorized access and denial of service.

This paper presents a fault-based testing approach for determining existence or absence of incorrect combining algorithms in XACML 3.0 policies. Given an XACML policy (or policy set), our approach analyzes whether the given combining algorithm is functionally equivalent to each of the candidate combining algorithms with respect to the rules in the given policy (or policies in the given policy set). If they can be different, our approach exploits a constraint solver to generate a query to which the two combining algorithms result in different responses. The combining algorithm is correct only if it produces correct responses to such queries. In theory, the query generation involves an NP-hard problem because the targets and conditions in XACML rules, policies and policy sets can be complex first-order logic formulas with user-defined functions. In practice, our case studies have demonstrated that the implementation of our approach based on an efficient constraint solver Z3-str [6][15] is both feasible and effective for dealing with sizable XACML policies.

The remainder of this paper is organized as follows. Section II gives a brief introduction to XACML policies and combining algorithms. Section III describes the fault-based testing approach. Section IV elaborates on fault-based test generation. Section V presents the empirical studies. Section VI reviews related work. Section VII concludes this paper.

(DOI reference number: 10.18293/SEKE2015-244)

II. XACML POLICIES AND COMBINING ALGORITHMS

The main components of the XACML3.0 model are rule, policy, and policy set. A rule consists of a target, a condition, and an effect. The target is a logical expression that specifies the set of requests to which the rule is intended to apply. The condition is a Boolean expression that refines the applicability of the rule established by the target. Predicates in target and condition are defined over attributes and attribute values (e.g., age>=18). A policy comprises a policy target, a rule-combining algorithm identifier, and a list of rules. A policy set consists of a policy set target, a policy-combining algorithm identifier, and a list of policies or policy sets. Figure 1 shows the relationships between the main elements of XACML3.0. For simplicity, this paper focuses on policies and rule combining algorithms.

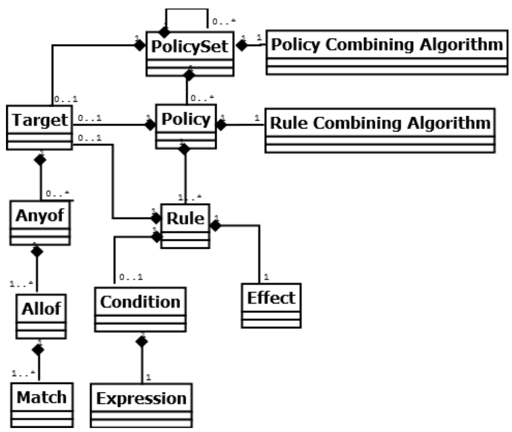

Figure 1. Main language elements of XACML 3.0

Formally, a policy $P=$ <PT, CA, R> consists of a policy target PT, a rule combining algorithm CA, and a list of rules R[1]. Each rule $r_i \in R$ is a triple <rt_i, rc_i, re_i>, where rt_i is the rule's target, rc_i is the rule's condition, and re_i is the rule's effect (either *Deny* or *Permit*). r_i is called a permit rule if re_i=*Permit*; r_i is called a deny rule if re_i=*Deny*; rt_i and rc_i are optional. A rule without target and condition, denoted by <_, _, re_i> is called a default rule.

An access request (also called query) consists of a list of attribute assignments: $\{x_1=V_1,\ x_2=V_2, \ldots\}$, where x_i is an attribute name and V_i is a value assigned to x_i. The decision of rule r=<rt, rc, re> with respect to request q, denoted by $d(r, q)$, is defined as follows:

- *Permit*: access is granted when rule effect re = *Permit*, query q matches policy target PT and rule target rt, and rule condition rc is true with respect to q.
- *Deny*: access is denied when re = *Deny*, q matches PT and rt, and rc is true with respect to q.
- *N/A*: q is not applicable – q does not match rt or rc evaluate to false with respect to q.
- *I(D)*: An error occurred when rt or rc was evaluated and re=*Deny*. The decision could have evaluated to *Deny* if no error had occurred.
- *I(P)*: An error occurred when rt or rc was evaluated and re=*Permit*. The decision could have evaluated to *Permit* if no error had occurred.

For convenience, we use *N/A*, *I(D)*, *I(P)*, and *I(DP)* to denote the following decisions respectively: *NotApplicable*, *Indeterminate {D}*, *Indeterminate {P}*, and *Indeterminate {DP}*. So $d(r, q) \in$ {*Permit*, *N/A*, *I(P)*} if r is a permit rule, and $d(r, q) \in$ {*Deny*, *N/A*, *I(D)*} if r is a deny rule. For a default rule r = <_, _, re>, $d(r, q) = re$ for any q.

Given query q, rules $r_1, r_2, \ldots, r_n$ in policy P=<PT, CA, R> may yield different decisions. The rule combining algorithm CA combines the decisions of individual rules into a single policy-level decision, denoted as $d(P, q)$. In XACML 3.0, there are 11 rule combining algorithms. Four are for compatibility support of old versions - *Legacy Ordered-deny-overrides*, *Legacy Permit-overrides*, *Legacy Ordered-permit-overrides*, and *Legacy Ordered-permit-overrides*. In Balana [1] (an open source implementation of XACML3.0 based on which our approach is developed), the implementations of *Ordered-deny-overrides* and *Ordered-permit-overrides* are the same as *Deny-overrides* and *Permit-overrides*. Thus, this paper focuses on five rule combining algorithms: *Deny-overrides*, *Deny-unless-permit*, *Permit-overrides*, *Permit-unless-deny*, and *First-applicable*. Their meanings are as follows:

- *Deny-overrides*: Intended for those cases where a deny decision should have priority over a permit decision;
- *Permit-overrides*: Intended for the cases where a permit decision should have priority over a deny decision.
- *Deny-unless-permit*: Intended for those cases where a permit decision should have priority over a deny decision, and an "*Indeterminate*" or "*NotApplicable*" must never be the result.
- *Permit-unless-deny*: Intended for those cases where a deny decision should have priority over a permit decision, and an "*Indeterminate*" or "*NotApplicable*" must never be the result.
- *First-applicable*: Rules are evaluated in the order in which they are listed. If a rule's target matches and condition evaluates to "True", then return the rule's effect (*Permit* or *Deny*). If the target or condition evaluates to "False", the next rule is evaluated. If no further rule exists, then return "*NotApplicable*". If an error occurs, then return "*Indeterminate*", with the appropriate error status.

Given policy P=<PT, CA, R>, the set of possible policy decisions depends on CA. For example, *Deny-overrides*, *Permit-overrides*, and *First-applicable* may yield one of the following six decisions: {*Permit*, *Deny*, *N/A*, *I(D)*, *I(P)*, *I(DP)*}, where *I(DP)* refers to *Indeterminate{DP}*. *I(DP)* results from one of the following situations: (a) an error occurred when policy target PT was evaluated and the decision could have evaluated to *Deny* or *Permit* if no error had occurred; (b) there is a permit rule that evaluates to *I(P)* and a deny rule that evaluates to *I(D)* or *Deny* when CA=*Permit-overrides;* (c) there is a deny rule that evaluates to *I(D)* and a permit rule that evaluates to *I(P)* or *Permit* when CA=*Deny-overrides*. *Deny-unless-permit* and *Permit-unless-deny* result in either *Permit* or *Deny*.

[1] In XACML, a policy also has other components, such as obligations and advice. We do not consider these components due to their irrelevance to the research in this paper.

III. Fault-Based Testing of Combining Algorithms

Fault-based testing aims to determine the existence or absence of a hypothesized fault [12]. It has been widely used to generate test cases or evaluate the quality of given tests. This paper focuses on fault-based test generation for incorrect combining algorithm in policy $P = <PT, CA, R>$. The basic idea is as follows: assuming CA is faulty and CA' is the correct combining algorithm, the fault-based approach generates a query q such that $d(P, q) \neq d(P', q)$, where $P' = <PT, CA', R>$, called P's mutant. P' has the same policy target and rules as P. According to the correct response to q (called oracle value, denoted as $o(q)$), we can determine whether CA or CA' is faulty. Note that, when testing P, we do not know which combining algorithm is the right one. However, it must be in the given set of rule combining algorithms (denoted as RCA). RCA does not have to contain all the combining algorithms in XACML. It can be a subset, depending on the application. For instance, a meaningful set of combining algorithms to be considered for a particular application might be *{Permit-overrides, Permit-unless-deny, First-applicable}*, rather than all the 11 rule combining algorithms in XACML 3.0. As such, our approach considers each possible mutant $P' = <PT, CA', R>$ where $CA' \in RCA$ and $CA' \neq CA$ and aims to generate a query to show the difference between P and each P'.

Although CA and CA' are meant to be different, P and P' can be functionally equivalent for certain PT and R, i.e., $d(P, q)=d(P', q)$ for any query q. For example, if R has only permit rules, *Deny-overrides* and *Permit-overrides* would make no difference. Let *query (P, P')* denote the function that returns *null* if P and P' are functionally equivalent, otherwise returns a query q such that $d(P, q) \neq d(P', q)$. Let $Q = \{q: q= query\ (P, P') \wedge q \neq null$ for each mutant $P' =<PT, CA', R>, CA' \in RCA$ and $CA' \neq CA\}$. CA in P is correct if and only if $d(P, q)=o(q)$ for any $q \in Q$. In other words, CA is incorrect if there exists $q \in Q$ such that $d(P, q) \neq o(q)$. Here, determining whether the given combining algorithm is correct or not requires user to define $o(q)$ according to the access control requirements. In our approach, the maximum number of queries for which user needs to define oracle values is $|RCA|$ -1. This is much more effective than reviewing all the rules in the policy or testing the policy with many queries. As reviewed in Section VI, the existing testing methods for XACML policies do not target the detection of incorrect combining algorithms. They all generate a large number of queries to which user has to define the oracle value of each query.

The fault-based testing of XACML combining algorithms in our approach involves two issues: (1) determine when P and P' are functionally equivalent with respect to the given policy target and rules; and (2) when P and P' are not functionally equivalent, find a query q such that $d(P, q) \neq d(P', q)$. To address the first issue, our technical report [14] has formalized the semantic differences between the five rule combining algorithms and between the six policy combining algorithms with 49 theorems. These theorems describe the necessary and sufficient conditions under which different combining algorithms are functionally equivalent. Based on [14], this paper focuses on the second issue by exploiting a constraint solver for automated test generation. For example, the following two theorems capture the semantic difference between rule combining algorithms *Deny-overrides* and *Permit-overrides*. Detailed proofs can be found in [14].

Theorem 1. Given policy $P = <PT, Deny\text{-}overrides, R>$ and $P'= <PT, Permit\text{-}overrides, R>$. If r_i ($1 \leq i \leq n$) are all permit rules or r_i ($1 \leq i \leq n$) are all deny rules, then P and P' are functionally equivalent.

Theorem 2. Given policy $P=<PT, Deny\text{-}overrides, R>$ and $P'=<PT, Permit\text{-}overrides, R>$, where R has at least one permit rule and at least one deny rule. For any q, $d(P, q) \neq d(P', q)$ if and only if there exists permit rule $r_i=<rt_i, rc_i, Permit> \in R$, deny rule $r_j =<rt_j, rc_j, Deny> \in R$, and query q, such that:

(a) $d(r_i, q) = Permit \wedge d(r_j, q) \in \{Deny, I(D)\}$ or
(b) $d(r_i, q) = I(P) \wedge d(r_j, q) = Deny$.

The above theorems lay the foundation for generating query q such that $d(P, q) \neq d(P', q)$. The corresponding test generation algorithm is described in the next section.

IV. Fault-Based Test Generation

This section discusses how to design and implement *query (P, P')* using constraint solver Z3-str. Z3 [6] is an efficient SMT (Satisfiability Modulo Theories) Solver from Microsoft Research. SMT generalizes Boolean Satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. Z3 supports basic data types (e.g., *Int* and *Bool*) as well as data structures (e.g., *Array, List, BitVec, and Records*). However, Z3 does not directly deal with strings. To address this issue, Z3-str [15] extends Z3 by treating strings as a primitive type and supporting common string operations.

In the following, we first introduce the basic functions that generate queries for a pair of rules and then describes how they are used in the query generation algorithms for comparing combining algorithms. These basic functions represent the queries used in the detailed proofs of the theorems [14]. We also discuss how to implement the basic query generation functions by transforming the corresponding targets and conditions of an XACML policy into the input of Z3-str.

A. Query Generation Functions

Suppose $r_1 =<rt_1, rc_1, re_1>$ and $r_2 =<rt_2, rc_2, re_2>$ are two rules. E, N, and I stand for *Effect* (*Permit* or *Deny*), *N/A*, and *Indeterminate*, respectively. For simplicity, here we do not consider targets of policies or policy sets, which are handled similarly. The basic query generation functions are as follows:

- *queryE_E*(r_1, r_2): generate a query q to make both r_1 and r_2 produce the specified effects re_1 and re_2, respectively (i.e., $d(r_1, q) = re_1$ and $d(r_2, q) = re_2$). In this case, the rule targets and conditions are all satisfied, i.e., $rt_1 \wedge rc_1 \wedge rt_2 \wedge rc_2$.
- *queryE_N*(r_1, r_2): generate a query q to make r_1 produce the specified effect re_1 and r_2 produce *N/A* (i.e., $d(r_1, q) = re_1$ and $d(r_2, q) = N/A$). In this case, $rt_1 \wedge rc_1 \wedge \neg (rt_2 \wedge rc_2)$.
- *queryE_I*(r_1, r_2): generate a query q to make r_1 produce the specified effect re_1 and r_2 produce *Indeterminate*

(i.e., an error in the process of evaluation). $d(r_1, q) = re_1$ and $d(r_2, q) = I(D)$ when re_2= *Deny* or $I(P)$ when re_2= *Permit.*

- *queryI_N*(r_1, r_2)*:* generate a query q to make r_1 produce *Indeterminate* and r_2 produce *N/A*. In this case, $d(r_1, q) = I(D)$ when re_1= *Deny* or $I(P)$ when re_1= *Permit.* $d(r_2, q) = N/A$.
- *queryN_N*(r_1, r_2)*:* generate a query q to make both r_1 and r_2 produce *N/A* (i.e., $d(r_1, q) = N/A$, $d(r_2, q) = N/A$). In this case, $\neg(rt_1 \wedge rc_1) \wedge \neg(rt_2 \wedge rc_2)$.
- *queryI_I*(r_1, r_2)*:* generate a query to make both r_1 and r_2 produce *Indeterminate*.

Using the above functions, we can formalize the algorithms for each pair of the combining algorithms according to the formalized semantic difference [14]. Consider *Deny-overrides* and *Permit-overrides* as an example. Algorithm 1 below describes the query generation process based on Theorems 1 and 2. According to Theorem 1, if the rules are all permit rules or all deny rules, they are functionally equivalent and thus no query can be generated. This is corresponding to lines 1-4 in Algorithm 1. According to Theorem 2, if a query makes a pair of permit and deny rules produce *Permit* and *Deny* (or *I(D)*) respectively (i.e., condition (a) in Theorem 2), then *Deny-overrides* and *Permit-overrides* produce different responses to this query. This is corresponding to lines 6-18 in Algorithm 1. Similarly, if a query makes a pair of deny and permit rules produce *Deny* and *I(P)* respectively (i.e., condition (b) in Theorem 2), *Deny-overrides* and *Permit-overrides* also produce different responses to this query. This is done by lines 19-26 in Algorithm 1.

Algorithm 1**:** *query(P=<PT, Deny-overrides, R>, P'=<PT, Permit-overrides, R>)*
Function: generate q such that $d(P, q) \neq d(P', q)$ if feasible.
Input: *P=<PT, Deny-overrides, R>, P'=<PT, Permit-overrides, R>*
Output: query q or *null*

```
1.   if re_i= Permit for all i (1 ≤i≤ n)  // Theorem 1
2.      return null;
3.   else if re_i= Deny for all i (1 ≤i≤ n) // Theorem 1
4.        return null;
5.   else // Theorem 2
6.      for r_i = 1st permit rule to last permit rule, do
7.         for r_j =1st deny rule to last deny rule, do:
8.            q = queryE_E(r_i, r_j);
9.            if (q!=null)
10.              return q;
11.           else
12.              q = queryE_I(r_i, r_j);
13.              if (q!=null)
14.                    return q;
15.              end if
16.           end if
17.        end for
18.    end for  // condition (a)
19.    for r_i = 1st deny rule to last deny rule, do:
20.        for r_j = 1st permit rule to last permit rule, do:
21.           q = queryE_I(r_i, r_j);
22.           if (q!=null)
23.              return q;
24.           end if
25.        end for
26.    end for // condition (b)
27.    return null;
28. end if
```

B. Transforming XACML Constructs to Z3-str

The aforementioned basic query generation functions are realized by transforming XACML constructs (i.e., targets and conditions) to the input of Z3-str, executing Z3-str with the transformed input, and translating the result of Z3-str to an XACML query. Converting XACML targets and conditions consists of two steps. In the first step, attributes in the given targets and conditions (i.e., rt_1, rc_1, rt_2, and rc_2 in the aforementioned basic query generation functions) are defined as typed variables in Z3-str. The attributes have to be renamed in Z3-str because the syntax of identifiers is different. The data type of each XACML attribute is also changed to a data type in Z3-str. XACML3.0 has 17 basic data types: *string*, *Boolean*, *integer*, *double*, *time*, *date*, *dateTime*, *anyURI*, *hexBinary*, *base64 Binary*, *dayTimeDuration*, *yearMonthDuration*, *rfc822Name*, *x500Name*, *xpathExpression*, *ipAddress*, and *dnsName*. Each of these data types can be mapped to a basic data type or data structure in Z3-str. For example, *date* in XACML can be corresponding to a record with three integer fields. In the second step, the logical expressions of targets and conditions are converted into Z3-str expressions. As the conversion involves many non-trivial details, here we use some examples to illustrate the idea. Consider the following rule target in XACML (for clarity, URI links are omitted):

```
<AnyOf>
   <AllOf>
      <Match MatchId="…:function:string-equal">
         <AttributeValue DataType="…string">book</AttributeValue>
         <AttributeDesignator AttributeId="…resource:resource-id"
            Category="…attribute-category:resource"
            DataType="…string" MustBePresent="true"/>
         </AttributeDesignator>
      </Match>
      <Match MatchId="…:function:string-equal">
         <AttributeValue DataType="…string">buy</AttributeValue>
         <AttributeDesignator AttributeId="…:action:action-id"
            Category="…:attribute-category:action"
            DataType="…string" MustBePresent="true"/>
         </AttributeDesignator>
      </Match>
   </AllOf>
   <AllOf>
      <Match MatchId="…function:string-equal">
       <AttributeValue DataType="…string">teacher</AttributeValue>
       <AttributeDesignator AttributeId="…subject:subject-id"
            Category="…subject-category:access-subject"
            DataType="…string" MustBePresent="true"/>
       </AttributeDesignator>
      </Match>
    </AllOf>
</AnyOf>
<AnyOf>
   <AllOf>
      <Match MatchId="…function:string-equal">
      <AttributeValue DataType="…string">workday</AttributeValue>
       <AttributeDesignator AttributeId="…environment:day"
            Category="…environment-category: environment"
            DataType="…string" MustBePresent="true"/>
       </AttributeDesignator>
```

```
    </Match>
  </AllOf>
</AnyOf>
```

The above target has the same meaning as the following logic formula:

$((resource\text{-}id = book \wedge action\text{-}id = buy)$
$\vee\, subject\text{-}id = teacher) \wedge (day{=}workday)$

where attributes resource-id, action-id, subject-id, and day are all of the string type. A non-error query should provide a value for each attribute because of *MustBePresent="true"*. To generate a query to satisfy the target condition, it can be converted into the following Z3-str input:

```
(declare-variable resourceid String)
(declare-variable actionid String)
(declare-variable subjectid String)
(declare-variable day String)
(assert (and (or (and (=resourceid "book")(= actionid "buy")) (and (=subjectid "teacher"))) (or (and (= day "workday")))))
(check-sat)
(get-model)
```

The "declare-variable" statements define variables for the attributes, and the "assert" expression describes the constraint to be solved.

For query generation functions $queryE\_E(r_1, r_2)$, $queryE\_N(r_1, r_2)$, $queryN\_N(r_1, r_2)$, we only need to make the targets and conditions true or false (e.g., $rt_1 \wedge rc_1 \wedge rt_2 \wedge rc_2$ for $queryE\_E(r_1, r_2)$). The other functions, $queryE\_I(r_1, r_2)$, $queryN\_I(r_1, r_2)$, and $queryI\_I(r_1, r_2)$, however, generate queries to produce *Indeterminate* by triggering an error status. Generation of such queries is much more complicated as discussed below. Typically, such a query should make part of a target (or condition) produce an error while ensuring the other part to evaluate to true or false. Therefore, query generation may involve selecting an appropriate attribute to trigger an error. In the above example, if we choose attribute *day* to trigger an error (e.g., a query that provides no value for *day*), then we have to ensure the resultant query must satisfy the following condition:

$$((resource\text{-}id{=}book \wedge action\text{-}id = buy) \vee subject\text{-}id = teacher)$$

If a query does not meet this condition, then *day=workday* will not be evaluated. Thus, it will not produce an error. If we choose *subject-id* to produce an error, then the resultant query should make *(resource-id = book* $\wedge$ *action-id = buy)* evaluate to false, otherwise *subject-id = teacher* will not be evaluated.

Generally, there are a great variety of errors that can result in a response of *Indeterminate* in XACML 3.0 [12]. The errors can be caused by problematic policies, queries, or both. Here our focus is on the errors caused by queries, assuming that the given policy is well-defined except for incorrect combining algorithm. In addition, $queryE\_I(r_1, r_2)$, $queryN\_I(r_1, r_2)$, and $queryI\_I(r_1, r_2)$ need to consider interactions of attributes in both rules. When both rules use the same set of attributes, it may be infeasible to create a particular type of error to obtain *Indeterminate*. This is because a query making one rule evaluate to *I(D)* or *I(P)* may also make the other rule evaluate to *I(D)* or *I(P)*.

V. Empirical Studies

We have implemented our approach based on Balana [1] and applied it to nine case studies with different levels of complexity. The case studies are summarized in Table I. K-market is a sample application of Balana with a total of 12 rules in three policies. It is the only one that is originally encoded in XACML 3.0. *itrust*, *pluto*, *conference*, and *fedora* are real-world policies from literature. They were originally encoded in XACML 2.0 or 1.0. In this paper, we manually converted them into XACML 3.0 with the same semantics. *itrustX* (*X=5*, *10*, *20*, or *40*) is a policy synthesized from *itrust*. It has *X* times as many rules as *itrust*. The new rules in *itrustX* are created by replicating the existing rules with new attribute values. Because the real-world policies from literature have a small number of rules, we use *itrustX* to evaluate whether or not our approach is applicable to large-scale policies.

Table I. Subject policies of empirical studies

Name	***#Rules***	***Combining algorithm***	***Equivalent combining algorithms***
K-market [1]	12	Deny-overrides	None
itrust[2]	64	First-applicable	Permit-overrides/ Deny-overrides
pluto	21	Permit-overrides	None
conference	15	Permit-overrides	None
fedora[3]	12	Deny-overrides	None
itrust5	320	First-applicable	Permit-overrides/ Deny-overrides
itrust10	640	First-applicable	Permit-overrides/ Deny-overrides
itrust20	1,280	First-applicable	Permit-overrides/ Deny-overrides
itrust40	2,560	First-applicable	Permit-overrides/ Deny-overrides

We treat the combining algorithm in each original policy as the correct one and inspect each policy to determine which combining algorithms are functionally equivalent and which are non-equivalent for each given policy. As shown in Table I, the policies in *itrust* and its variations have equivalent algorithms. As the correct combining algorithms in the given policies are already assumed, the goal of our evaluation is to demonstrate whether or not our approach can detect incorrect combining algorithms and functionally equivalent combining algorithms. Let P_0 and CA_0 denote the correct policy (or policy set) and original combining algorithm respectively. We used the following protocol to conduct the experiment:

- Use the correct policy P_0 to create a policy or policy set P with a different combining algorithm CA (i.e., $CA \neq CA_0$);
- Apply our approach to P, comparing CA to each of the other combining algorithms (including CA_0) and try to generate a query for each pair;
- If no query is generated for $<P, P_0>$ and $d(P, q) = d(P_0, q)$ for each query q generated in the above step, then CA is correct and functionally equivalent to CA_0, otherwise CA is incorrect.

[2] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start
[3] http://www.fedora.info

The results of our experiments have shown that our approach was able to identify all correct and equivalent combining algorithms as defined in Table I. Consider *itrust* (or *itrustX*). *First-applicable*, *Deny-overrides*, and *Permit-overrides* are equivalent. When any two of them were compared, no query was generated, which means they have no difference. When one of them was compared to *Deny-unless-permit* or *Permit-unless-deny*, however, a query was generated, which means they are different. In *K-market*, *pluto*, *conference*, or *fedora*, a query was generated for each pair of combining algorithms. This means that all the combining algorithms are different with respect to the given policy target and rule.

VI. RELATED WORK

In Cirg [9], tests are generated from counterexamples produced by the change-impact analysis of two synthesized versions. The difference of the two versions of a policy targets a test coverage goal (e.g., rule, or condition). Targen [10] is a test generator for XACML policies that derives access requests to satisfy all the possible combinations of truth values of the attribute id-value pairs found in a given policy. Access requests generated by Cirg and Targen typically use a limited number of subject, resource, action, and environment attributes. A real request, however, could use any combination of attributes. Because requests are encoded in XML, they must conform to the XML Context Schema. To address this issue, Bertolino et al., have developed different test generation algorithms by considering the structures of the Context Schema [2][3][5]. These algorithms can generate requests that use more than one subject, resource, action, or environment attribute. They can also produce robustness tests, where invalid attribute values are generated randomly.

Li et al. have applied symbolic execution technique to generation of access requests for testing XACML policies [8]. They convert the policy under test into semantically equivalent C Code Representation (CCR) and symbolically execute CCR to create test inputs and translate the test inputs to access control requests. Mutation of the XACML policies [4][11] has been commonly used to evaluate the above testing methods. In this paper, however, we use combining algorithm-based mutants to generate queries for determining whether or not the given combining algorithm is correct.

VII. CONCLUSIONS

We have presented the fault-based approach to automated test generation for determining existence or absence of incorrect combining algorithms in XACML3.0 policies. Based on the formalized semantic differences between combining algorithms, our approach exploits a constraint solver to generate a query to show the difference between the given combining algorithms and each of the mutants. Our case studies have demonstrated that the approach is effective and applicable to sizeable policies. As a byproduct, our approach can be a useful tutoring tool for learning about XACML combining algorithms and their essential differences. When a user is uncertain about which combining algorithm should be used, she may compare similar algorithms and generate requests to show the difference. This will help the user get an accurate understanding and choose the right combining algorithm.

This paper offers a first step towards general fault-based testing of XACML policies. Incorrect combining algorithms are just one type of faults in XACML policies. Other fault types include incorrect (policy set, policy, and rule) target, incorrect rule effect, and incorrect rule conditions [4][11]. Our future work will investigate fault-based test generation algorithms for each of these uncovered fault types.

ACKNOWLEDGMENT

This work was supported in part by US National Science Foundation (NSF) under grants CNS 1123220 and 1359590.

REFERENCES

[1] Balana, “Open source XACML 3.0 implementation,” http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/, 2012.

[2] A. Bertolino, S. Daoudagh, F. Lonetti, and E.marchetti. "Automatic XACML requests generation for policy testing." Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST), 2012, pp.842-849.

[3] A. Bertolino, S. Daoudagh, F. Lonetti, and E.marchetti. "The X-CREATE Framework-A Comparison of XACML Policy Testing Strategies." *Proc. of the 8th International Conference on Web Information Systems and Technologies (WEBIST)*. pp.155-160.

[4] A.Bertolino, S.Daoudagh, F.Lonetti, and E.Marchetti. "Xacmut: Xacml 2.0 mutants generator." Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),. 2013, pp.28-33.

[5] A.Bertolino, S.Daoudagh, F.Lonetti, E.Marchetti and L.Schilders. "Automated testing of extensible access control markup language-based access control systems." Software, IET 7.4 (2013), pp.203-212.

[6] L.De.Moura, and N.Bjørner. "Z3: An efficient SMT solver." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008, pp.337-340.

[7] V.C.Hu, D.Ferraiolo, R.Kuhn, A.Schnizer, K.Sandlin, R.Miller and K.Scarfone. "Guide to Attribute Based Access Control (ABAC) Definition and Considerations." NIST Special Pub 800 (2014): 162.

[8] Y.C.Li, Y.Li, L.Z.Wang, and G.Chen. "Automatic XACML Requests Generation for Testing Access Control Policies." Proc. of the 26th International Conf. on Software Engineering and Knowledge Engineering (SEKE'14), Vancouver. July 2014.

[9] E. Martin and T. Xie. “Automated test generation for access control policies,” in *Supplemental Proc. of ISSRE*, November 2006.

[10] E.Martin, and T.Xie. "Automated test generation for access control policies via change-impact analysis." Proceedings of the Third International Workshop on Software Engineering for Secure Systems. IEEE Computer Society, 2007, pp.5-11.

[11] E.Martin, and T.Xie. "A fault model and mutation testing of access control policies." Proceedings of the 16th International Conference on World Wide Web. ACM, 2007, pp.667-676.

[12] L.J. Morell. “A theory of fault-based testing”, IEEE Trans. on Software Engineering, Vol. 16, no.8, August 1990, pp. 844-857.

[13] OASIS, “eXtensible Access Control Markup Language (XACML) Version 3.0,” http://www.oasisopen.org/committees/xacml/. 2013.

[14] D. Xu, Y. Zhang, N. Shen. “Formalizing semantic differences of combining algorithms in XACML 3.0,” Technical Report, Boise State University. http://cs.boisestate.edu/~dxu/research/TR-BSU-CS-SEAL2014-001.pdf

[15] Y. Zheng, X. Zhang, and V. Ganesh, “Z3-str: A Z3-based string solver for web application analysis,” Proc. of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE’13), pp.114-124.