Coverage-based Test Cases Selection for XACML Policies

Antonia Bertolino*, Yves Le Traon†, Francesca Lonetti*, Eda Marchetti*, Tejeddine Mouelhi†
*Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR
Pisa, Italy
{firstname.lastname}@isti.cnr.it
†Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg
Luxembourg
{firstname.lastname}@uni.lu

***Abstract*—XACML is the de facto standard for implementing access control policies. Testing the correctness of policies is a critical task. The test of XACML policies involves running requests and checking manually the correct response. It is therefore important to reduce the manual test effort by automatically selecting the most important requests to be tested. This paper introduces the XACML smart coverage selection approach, based on a proposed XACML policy coverage criterion. The approach is evaluated using mutation analysis and is compared on the one side with a not-reduced test suite, on the other with random and greedy optimal test selection approaches. We performed the evaluation on a set of six real world policies. The results show that our selection approach can reach good mutation scores, while significantly reducing the number of tests to be run.**

I. Introduction

In modern dynamic distributed systems, where resources and data are continuously exchanged and shared, security is a primary concern. Thus appropriate mechanisms that guarantee the data confidentiality, integrity and availability must be put in place. Among them, one of the most important components is the access control system that: mediates all requests of access to protected data and resources; ensures that only the intended (i.e., authorized) users are given access; and provides them with the level of access that is required to accomplish their tasks (and not higher).

In this context, XACML [1] is the de facto standard for defining and implementing access control systems. In particular, an XACML policy specifies the constraints and conditions that a subject needs to comply with for accessing a resource and doing an action in a given environment. XACML specifies also the access control system architecture: incoming access requests are transmitted to the Policy Decision Point (PDP) that grants or denies the access based on the defined XACML policies. Due to the complexity of the XACML language, the process of writing the policies and implementing them can be error-prone. Faults could lead to security flaws, by either denying accesses that should be allowed or even worse allowing accesses to non authorized users. It is hence important to perform a careful testing activity both of the policy and its implementation. However, most of the test cases generation approaches available in literature for XACML policies are based on combinatorial methodologies [2], [3], thus the generated number of test cases can rapidly grow to cope with the policy complexity. Executing a huge number of test cases can drastically increase the cost of the testing phase, mainly due to the effort required to check the test results and decide whether they are correct or not. As a matter of fact, in the context of access control systems this step is usually performed manually, because the complexity of the XACML language prevents the use of automated support. Considering the strict constraints on testing budget, it is extremely important to reduce as much as possible the number of tests to be executed while trying at the same time to maximize the fault detection effectiveness of the reduced test suite. This papers addresses this specific issue, by proposing a test selection approach, called *XACML smart coverage*, based on a coverage criterion, specifically conceived on the peculiarities of the XACML language. In software testing, white-box approaches based on the coverage of specified entities are considered a valuable complement to black-box ones [4], as coverage information can provide an indication of the thoroughness of the executed test cases, and can help to maintain an effective test suite. However, as demonstrated by some empirical results [5], the performance of the reduced test suite could vary according to the considered program, and the adopted coverage criterion. Therefore, we investigate in this paper the loss of fault detection effectiveness due to the execution of the reduced test suite. By means of mutation analysis we inject faults into the XACML policy and challenge the tests to detect these seeded faults. The goal is to end up with a reduced test suite able to minimize the loss in mutation score.

There have been a few proposals in literature close to the one of this paper. In particular, Martin et al. [6] proposed the definition of some policy coverage metrics and a greedy algorithm for test case selection able to increase the overall coverage measure. Hwang et al. [7] focused on regression testing and used some monitoring facilities, directly integrated on the program code, to establish correlation between executed test cases and the entities covered in an XACML policy. *XACML smart coverage* differs from these works

in several aspects: it integrates and extends the coverage criteria proposed in [6] by including additional constraints and XACML elements; it is not specifically conceived for regression testing, therefore it does not rely on some coverage information obtained from a previous execution of test cases; it does not require any instrumentation of program code nor any monitoring facility for coverage measurement, because it exploits only the XACML policy information and the test case specification. In addition, we improve on the trustworthiness of the experimental analyses presented in [6] and [7] by: using a set of XACML policies, that is larger and includes policies of varying structural complexity; adopting a test generation strategy that has been proven to be more effective than existing ones; considerably increasing the set of mutation operators for the evaluation of test suite effectiveness.

Summarizing, the contributions of this paper include: the introduction of a new more complete coverage criterion for XACML policies; the definition of a strategy to select the test cases based on the proposed coverage criterion; an empirical study that compares the performance of the selected test cases with that of both the whole test suite without reduction, and other selection strategies (i.e., random and a greedy optimal solution) in terms of mutation score. The experimental results show that the *XACML smart coverage* selection approach substantially reduces the test suite size with a negligible loss of fault detection effectiveness.

The remainder of this paper is organized as follows. Section II introduces the XACML language. Section III presents the proposed coverage criterion and the selection algorithm. Then, Section IV shows the empirical evaluation of the proposed approach, and Section V discusses its validity. Finally, Section VI presents related work and Section VII concludes the paper, also hinting at future work.

II. XACML Language

XACML [1] is a de facto standard specification language that defines access control policies and access control decision requests/responses in an XML format. An XACML policy defines the access control requirements of a protected system. An access control request triggers a policy evaluation and aims at accessing a protected resource in a given system whose access is regulated by a security policy. At the evaluation time, the request is evaluated against the policy and the access is granted or denied.

An XACML policy can contain one or more *policy set* or *policy* elements. A *policy set* contains one or more *policy sets* or one or more *policies*. It includes a *target* to be matched from a request before considering the *policies* (*policy sets*) in that *policy set* to be applicable. A *target* contains four parameters, a set of subjects, a set of resources, a set of actions and finally a set of environments. A request is matching a *target*, when the subject, resource, action and environment of the request are included in the corresponding *target* sets. A *policy* contains one or more *rules*. A *rule* contains a decision type (Permit or Deny) and a *target*. When the request matches the *target* then the request is applicable to the *rule*. In that case the decision type is returned. A rule might also contains a *condition* element, i.e., a boolean function that specifies constraints on the subjects, resources, actions and environments values so that if the *condition* evaluates to true, then the rule's decision type is returned. A *combining algorithm* is used to select which policy (*policy-combining algorithm*) or rule (*rule-combining algorithm*) has to be considered in case the request matches more than one policy (or rule). For instance, the *first-applicable* combining algorithm will select the first applicable policy (or rule).

An access request contains subject, resource, action, and environment attributes. At the decision making time, the Policy Decision Point evaluates an access request against a policy, by comparing all the attributes in an access request against the attributes in all the *target* and *condition* elements of the *policy set*, *policy* and *rule* elements. If there is a match between the attributes of the request and those of the policy, the *effect* of a matching rule is returned, otherwise the *NotApplicable* decision is drawn.

Listing 1 shows an example of a simplified XACML policy for library access. The policy set target (line 3) is empty, which means that it applies to any subject, resource, action and environment. The policy target (lines 5-12) says that this policy applies to any subject, any action, any environment and the "books" resource. This policy has a first rule (*ruleA*) (lines 13-34) with a target (lines 14-33) specifying that this rule applies only to the access requests of a "read" action of "books", and "documents" resources with any environment. The effect of the second rule (*ruleB*) (lines 35-50) is *Permit* when the subject is "Julius", the action is "write", the resource and environment are any resource and any environment respectively.

```
1  <PolicySet PolicySetId="policySetExample"
2   PolicyCombiningAlgId="first-applicable">
3   <Target/>
4   <Policy PolicyId="policyExample" RuleCombiningAlgId="
        permit-overrides">
5    <Target>
6     <Resource>
7      <ResourceMatch MatchId="anyURI-equal">
8       <AttributeValue DataType="anyURI">books</
            AttributeValue>
9       <ResourceAttributeDesignator AttributeId="resource-id
            " DataType="anyURI"/>
10     </ResourceMatch>
11    </Resource>
12   </Target>
13   <Rule RuleId="ruleA" Effect="Deny">
14    <Target>
15     <Resources><Resource>
16      <ResourceMatch MatchId="anyURI-equal">
17       <AttributeValue DataType="anyURI">books</
             AttributeValue>
18       <ResourceAttributeDesignator AttributeId="resource-
             id" DataType="anyURI"/>
19      </ResourceMatch>
20     </Resource>
21     <Resource>
22      <ResourceMatch MatchId="anyURI-equal">
```

```
        <AttributeValue DataType="anyURI">documents</
            AttributeValue>
        <ResourceAttributeDesignator AttributeId="resource-
            id" DataType="anyURI"/>
       </ResourceMatch>
      </Resource></Resources>
     <Actions><Action>
       <ActionMatch MatchId="string-equal">
        <AttributeValue DataType="string">read</
            AttributeValue>
        <ActionAttributeDesignator AttributeId="action-id"
            DataType="string"/>
       </ActionMatch>
      </Action></Actions>
    </Target>
   </Rule>
   <Rule RuleId="ruleB" Effect="Permit">
    <Target>
     <Subjects><Subject>
       <SubjectMatch MatchId="string-equal">
        <AttributeValue DataType="string">Julius</
            AttributeValue>
        <SubjectAttributeDesignator AttributeId="subject-id
            " DataType="string"/>
       </SubjectMatch>
      </Subject></Subjects>
     <Actions><Action>
       <ActionMatch MatchId="string-equal">
        <AttributeValue DataType="string">write</
            AttributeValue>
        <ActionAttributeDesignator AttributeId="action-id"
            DataType="string"/>
       </ActionMatch>
      </Action></Actions>
    </Target>
   </Rule>
  </Policy>
</PolicySet>
```

Listing 1. An XACML Policy

Listing 2 shows an example of a simple request specifying that the subject Julius wants to write the "books" resource.

```
<Request xmlns="urn:oasis:names:tc:xacml:2.0
    :context:schema:os">
 <Subject>
  <Attribute AttributeId="subject-id1" DataType="string">
   <AttributeValue>Julius</AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
  <Attribute AttributeId="resource-id" DataType="string">
   <AttributeValue>books</AttributeValue>
  </Attribute>
 </Resource>
 <Action>
  <Attribute AttributeId="action-id" DataType="string">
   <AttributeValue>write</AttributeValue>
  </Attribute>
 </Action>
 <Environment/>
</Request>
```

Listing 2. An XACML request

III. Coverage based selection approach

This section presents the *XACML smart coverage* selection approach. First, an XACML coverage criterion is defined, then an algorithm developed to select a set of requests that achieve this coverage criterion is presented.

A. XACML coverage criterion

We define here the XACML rule coverage criterion on which the *XACML smart coverage* approach is based. We first provide some generic definitions concerning the policy (Definitions 1 and 2) and request elements (Definition 3).

Definition 1 (Target Tuple): Given a Rule *R*, a Policy *P*, a PolicySet *PS*, with $R \in P$ and $P \in PS$, and given the set of XACML Elements, called XE = {xe : xe is *PS* or *P* or *R*}, the Target Tuple of an xe $\in$ XE, called TT_{xe}, is a 4-tuple (S, Res, A, E), where: S (Res, A, E) is a finite set of subjects (resources, actions, environments) in the XACML target of xe.

Definition 2 (Rule Target Set): Given a Rule *R*, its Target Set is a set of Target Tuple, ordered by the XACML hierarchy elements relation, defined as

$$TS_R = \left\{ TT_{xe} : \quad TT_{xe} = \begin{cases} TT_{PS} & \text{if } R \in PS \\ TT_P & \text{if } R \in P \\ TT_R & \text{otherwise} \end{cases} \right\}.$$

Definition 3 (Request Target Tuple): Given a request Req, the Request Target Tuple, called TT_{req} is a 4 tuple (S_r, Res_r, A_r, E_r) where S_r, Res_r, A_r, E_r are the subject, resource, action and environment belonging to the request Req.

We can now define the XACML rule coverage criterion as follows:

Definition 4 (XACML Rule Coverage): Given a rule R, the condition C of R, the Rule Target Set TS_R, and the request Req with Request Target Tuple TT_{req}=(S_r, Res_r, A_r, E_r), Req covers R if and only if

- for each Target Tuple TT_E =(S, Res, A, E) $\in TS_R$ such that TT_E is a TT_{PS}, TT_P or TT_R, S_r $\in S$ or S is $\emptyset$, Res_r $\in Res$ or Res is $\emptyset$, A_r $\in A$ or A is $\emptyset$, and E_r $\in E$ or E is $\emptyset$.
- C is evaluated to True or False against TT_{req} [1].

Considering the policy of Listing 1, according to Definition 2, the Target Set of $ruleA$ is
TS_{RuleA} = {$TT_{PS_{policySetExample}}$, $TT_{P_{policyExample}}$, TT_{RuleA}} = {($\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$), ($\emptyset$, {$books$}, $\emptyset$, $\emptyset$),($\emptyset$, {$books$, $documents$}, {$read$}, $\emptyset$)}
while the Target Set of $ruleB$ is
TS_{RuleB} = {$TT_{PS_{policySetExample}}$, $TT_{P_{policyExample}}$, TT_{RuleB}} = {($\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$), ($\emptyset$, {$books$}, $\emptyset$, $\emptyset$), ({$Julius$}, $\emptyset$, {$write$}, $\emptyset$)}.

Considering the XACML request of Listing 2, according to Definition 3, the Request Target Tuple of this request is $TT_{requestExample}$ = ({$Julius$}, {$books$}, {$write$}, $\emptyset$).

According to Definition 4, the request of Listing 2 covers $ruleB$ but it does not cover $ruleA$ since the action of the request (*write*) is not included in the Target Set of $RuleA$.

[1]Note that only the condition is evaluated against the request values, without having policy execution.

In a nutshell, the defined XACML rule coverage criterion involves selecting tests that match the Rule Target Sets. The Rule Target Set is the union of the target of the rule, and all enclosing policy and policy sets targets. The main idea is that according to the XACML language in order to match the rule target, requests must first match the enclosing policy and policy sets targets (note that there could be several enclosing policy sets). For instance, if a rule contains no condition, and it has a target containing the elements {Subject1,Action1,Resource1} and the policy and policy set targets which it belongs to are both empty, then in order to match that rule a request should contain exactly these three elements. If the rule target has several subjects, resources, actions, and environments and the enclosing policy and policy set targets are empty, to cover the rule target the request should include a subject contained in target subjects set, a resource contained in the target resources set, an action contained in the target actions set, an environment contained in the target environments set. Finally, if the Rule Target Set of a rule is empty and its condition is evaluated to True or False, all requests are covering this rule.

B. Test case selection algorithm

Algorithm 1 is used to select the test cases. Roughly, it takes as input the Rule Target Sets and a set of requests. Then, it loops through the requests and selects those ones that match one Rule Target Set. Once a Rule Target Set is matched, it is removed from the set of Rule Target Sets. This prevents selecting all requests for empty Rule Target Sets.

Algorithm 1 Coverage-Based Selection of Test Cases

input: $S = \{Req_1, ..., Req_n\}$ ▷ Unordered set of n XACML requests
input: P ▷ The XACML policy
output: $Result$ ▷ List of m selected XACML requests with m ¡ n
$Result \leftarrow \{\}$
$TargetsConds \leftarrow computeAllRulesTargetsConds(P)$
$i \leftarrow 0$
while $i < TargetsConds.size()$ **do**
  $ContainsReq \leftarrow False$
  $j \leftarrow 0$
  while $!ContainsReq$ **do**
    $ReqTarget_j \leftarrow extractReqTarget(Req_j)$
    if $containsReq(TargetCond_i, ReqTarget_j)$ **then**
      $Result \leftarrow Result \cup \{Req_j\}$
      $ContainsReq \leftarrow True$
    end if
    $j \leftarrow j + 1$
    if then$j == n$
      Break loop
    end if
  end while
  $i \leftarrow i + 1$
end while
return $Result$

Algorithm 2 allows all Rule Target Sets and Rule Conditions to be computed by taking into consideration the rule and its enclosing policy and policy sets. In addition, when a target contains more than one subject, action or resource, the algorithm divides that target into several targets, each having only one of these elements. For instance, a target with 2 subjects, 1 action and 3 resources leads to creating 6 targets (each one with 1 subject, 1 action and 1 resource). In fact, according to the XACML language a rule containing for instance a target with 3 subjects is equivalent to three rules having a target with 1 subject. XACML offers this facility to avoid creating several rules, however for the sake of rule evaluation, it is safer to consider several rules having each a target with only one element. For test cases selection, having targets with one subject, action, resource and environment enables us to select test cases covering all subjects, actions, resources and environments and helps improving the quality of test cases.

IV. Experiment

This section presents and discusses the results of the application of the *XACML smart coverage* selection approach to a set of real XACML policies. First, we present the case study and the experiment setup, which includes the XACMUT tool for policy mutants generation and the X-CREATE tool for test case generation. Then we present the research questions and discuss the outcome of the experiments.

A. Policies details

One of the objectives of this paper is to improve the trustworthiness of the experimental analysis presented in previous related works (like [6] and [7]) by using a larger and more representative set of XACML policies. Thus, we include in the experiment six real world policies, which differ from each other in terms of the complexity of their structure and the number of elements they include. This information is summarized in Table I, which shows the

Algorithm 2 Compute All Targets and Conditions

input: $P = \{Rule_1, ..., Rule_n\}$ ▷ XACML policy having n rules
output: L ▷ List of n Targets with Condition
$L \leftarrow \{\}$
$i \leftarrow 0$
while $i < n$ **do**
  $TargCond_i \leftarrow \{\}$
  $EnclosingPol \leftarrow retrievePolForRule(Rule_i)$
  $PolTarget_i \leftarrow extractPolicyTarget(EnclosingPol)$
  $TargCond_i \leftarrow L \cup \{PolTarget_i\}$
  $EnclosingPolSet \leftarrow retrievePolSetForPol(EnclosingPol)$
  $PolSetTarget_i \leftarrow extractPolicySetTarget(EnclosingPolSet)$
  $TargCond_i \leftarrow L \cup \{PolSetTarget_i\}$
  while $isPolicySetEnclosedInPolicySet(Enclosing_{P}olSet)$ **do**
    $PolSet \leftarrow getParent(PolSet)$
    $PolSetTarget_i \leftarrow extractPolicySetTarget(EnclosingPolSet)$
    $TargCond_i \leftarrow L \cup \{PolSetTarget_i\}$
  end while
  $TargCond_i \leftarrow TargCond_i \cup \{Cond_i\}$
  $L \leftarrow L \cup \{TargCond_i\}$
end while
return L

Table I
DESCRIPTION OF THE SIX POLICIES

Name	# Rul.	# S	# Res	# A	# E	# Pol.set	# Pol.
ASMS	117	8	5	11	3	1	1
itrust	64	7	46	9	0	1	1
VMS	106	7	3	15	4	1	1
continue-a	298	16	29	4	0	111	266
LMS	42	8	3	10	3	1	1
pluto	21	4	90	1	0	1	1

sizes of the XACML policies in terms of the number of subjects, resources, actions and environments, in addition to the structure in terms of rules, policy sets and policies.

Briefly, the policy labeled *LMS* rules a Library Management System, *VMS* represents a Virtual Meeting System and *ASMS* is conceived for an Auction Sales Management System. All these policies are relative to three Java-based systems, which have been used in previous papers (e.g., [8]). The policy named *continue-a* [9] is used by the *Continue application*, a web-based conference management tool; *pluto* policy is used by the ARCHON system, a digital library management tool [10]; and *itrust* policy is part of the iTrust system, a health-care management system [11].

B. Experiment setup

The experiment carried out in this paper relies on two existing tools, performing policy mutants generation and test cases generation. In the remaining of this section they will be briefly introduced.

1) Mutation analysis: To derive the set of policy mutants useful for assessing the fault detection capability of the selected requests we rely on the mutation operators implemented in the XACMUT tool[2] [12]. To the best of our knowledge, XACMUT is currently the most complete tool for mutants derivation, since it combines together XACML mutants taken from the literature with new ones that have been conceived to address the specific features of XACML 2.0 policies. The XACMUT mutation operators are listed in Table II. For lack of space we do not provide a description beyond their name in the table, and refer to the respective sources for more information. Specifically, PSTT, PSTF, PTT, PTF, RTT, RTF, RCT, RCF, CPC, CRC and CRE have been introduced in [13]), RTT, ANR, and RER in [14], and the remaining ones in [12]. Relying on a more powerful tool than those used so far in literature increases the reliability of the fault detection effectiveness results obtained by our experiment and contributes to depict more realistic situations.

2) X-CREATE tool: Among the available tools for test cases generation we refer in this paper to X-CREATE[3]. X-CREATE [15], [3], [16] provides different strategies based on combinatorial approaches of the subject, resource, action and environment values taken from the XACML policy for deriving the access requests. Experimental results presented in [15], [3], [16] showed that the fault detection effectiveness of X-CREATE test suites is similar or higher than that of comparable tools (like for instance Targen [2]).

Among X-CREATE test strategies in this paper we consider the *Simple Combinatorial* one, because it combines the simplicity of the test case generation with the power of the combinatorial approach applied to the policy values. Specifically, for the test cases generation four data sets called *SubjectSet*, *ResourceSet*, *ActionSet* and *EnvironmentSet* are defined. Those sets are filled with the values of elements and attributes referring to the `<Subjects>`, `<Resources>`, `<Actions>` and `<Environments>` of the policy respectively. These elements and attributes values are then combined in order to obtain the entities. Specifically, a *subject entity* is defined as a combination of the values of elements and attributes of the *SubjectSet* set, and similarly the *resource entity*, the *action entity* and the *environment entity* represent combinations of the values of the elements and attributes of the *ResourceSet*, *ActionSet*, and *EnvironmentSet* respectively. Then, an ordered set of combinations of *subject entities*, *resource entities*, *action entities* and *environment entities* is generated in the following way: first, pair-wise combinations are generated to obtain the *PW* set; then, three-wise combinations are generated to obtain the *TW* set; finally, four-wise combinations are generated to obtain the *FW* set. These sets have the following inclusion propriety $PW \subseteq TW \subseteq FW$. The maximum number of requests derived by this strategy is equal to the cardinality of the *FW* set. More details about this strategy are in [3].

Table II
MUTATION OPERATORS [12]

ID	Description
PSTT,PSTF	Policy Set Target True/False
PTT,PTF	Policy Target True/False
RTT,RTF	Rule Target True/False
RCT,RCF	Rule Condition True/False
CPC,CRC	Change Policy/Rule Combining Algorithm
CRE	Change Rule Effect
RPT (RTT)	Rule Type is replaced with another one
ANR	Add a New Rule
RER	Remove an Existing Rule
AUF,RUF	Add/Remove Uniqueness Function
CNOF	Change N-OF Function
CLF	Change Logical Function
ANF,RNF	Add/Remove Not Function
CCF	Change Comparison Function
FPR,FDR	First the Rules having a *Permit*/*Deny* effect

[2]A release of the XACMUT tool is available at http://labse.isti.cnr.it/tools/xacmut

[3]A release of the X-CREATE tool is available at http://labse.isti.cnr.it/tools/xcreate

C. Results

As for any test strategy that relies on a combinatorial approach, the cardinality of the test suite derived by the *Simple Combinatorial* strategy may rapidly grow up in relation with the policy complexity. As we discussed in the introduction, this may result into a huge increase of time and effort due for the test execution and results analysis. In this paper we propose a solution based on the *XACML smart coverage* selection approach, and provide some experimental results to reply to the following research questions:

- RQ1: Is *XACML smart coverage* a good approach for test selection in terms of fault detection effectiveness?
- RQ2: Is *XACML smart coverage* better than other selection approaches in terms of fault detection effectiveness?

To tackle these research questions we first derived for each policy in Table I the set of test cases by applying the *Simple Combinatorial* strategy provided by the X-CREATE tool. Table III third column shows the cardinality of each derived test suite: as shown, the number of test cases has a great variation (from the 360 of *pluto* to the 2835 of *iTrust*) reflecting the differences in structures and values of the considered set of policies. Then we applied the *XACML smart coverage* selection approach to select from each test suite the proper reduced set. Table III second column shows the cardinality of the selected test suite. As shown in the last column of the table, for most cases (except *pluto*) the number of selected tests is quite low (less than 12%) and the cardinality of the reduced test suite remains manageable in terms of requests to be run and checked manually. This evidences a good performance of the *XACML smart coverage* selection approach purely reasoning in terms of test reduction. Finally, by means of the XACMUT tool, for each of the six policies we generated the respective set of mutants and used them for evaluating the test effectiveness of the various test suites. Table IV last column shows the cardinality of the mutants killed for each of the six policies.

To tackle RQ1, we compared the percentage of mutants killed by the reduced test suite derived using the *XACML smart coverage* selection approach with that killed by the overall test suite. It is out of the scope of this paper to evaluate the effectiveness of the test strategy used in this experiment; the objective is the evaluation of the capability of the *XACML smart coverage* selection approach to provide a fault detection effectiveness as close as possible to that of the overall test suite (whatever its effectiveness). The results are shown in Table IV. In particular, the third column (labeled Cov.-Based) reports the number of mutants killed by the *XACML smart coverage* selection and summarizes in brackets the percentage of fault detection effectiveness reached by the reduced test suite with respect to the complete one; the fifth column of the same table (labeled # Killed Mutants) reports the number of mutants killed by the complete test suite.

Table III
TEST REDUCTION OF THE COVERAGE BASED SELECTION

Policy	# SelectedTests	# Tests	% Selected Tests
LMS	42	720	6%
VMS	106	945	11%
ASMS	130	1760	7%
pluto	175	360	49%
iTrust	61	2835	2%
continue-a	169	1382	12%

Table IV
TEST SUITES EFFECTIVENESS IN TERMS OF MUTATION RESULTS

Pol.	Random	Cov.-Based	Opt. Score	# Killed Mutants
LMS	451 (20%)	1357 (62%)	1358 (62%)	2183
VMS	2771 (49%)	4031 (72%)	4076 (73%)	5550
ASMS	1522 (22%)	4771 (71%)	4809 (72%)	6649
pluto	7588 (51%)	13968 (94%)	13895 (94%)	14721
iTrust	1253 (10%)	11782 (98%)	11844 (99%)	11949
continue-a	757 (43%)	1333 (76%)	1685 (96%)	1741

Considering column Cov.-Based, the loss in fault detection (except LMS) varies from the 29% to the 2% , with an average value of 18%. Considering that, as reported in Table III, the selected test cases are on average the 15% of the overall test suite (for a reduction of 85%) the obtained average value of the loss of fault detection effectiveness of *XACML smart coverage* selection can be considered a valid compromise in view of the considerable test case reduction. Therefore the data collected in this experiment give a first positively reply to RQ1, i.e. *XACML smart coverage* can be considered a good approach for test selection in terms of fault detection effectiveness.

The second experiment focused on RQ2, i.e., we wanted to compare the performance in terms of fault detection effectiveness of *XACML smart coverage* with other selection approaches. In particular, we considered two different cases: the random selection of test cases (as a baseline), and on the other hand an optimal approach for test selection.

For the first case we randomly selected from the complete test suite, the same number of test cases of the *XACML smart coverage* selection. To prevent bias, we performed the random selection 10 times and computed the average number of killed mutants on the 10 runs. The results are shown in Table IV second column (labeled Random). Again, the first values are the number of mutants killed and in brackets the percentage of fault detection effectiveness of random selection with respect to that of the complete test suite. As shown in the table, the results of the *XACML smart coverage* selection (third column) outperform in all cases those obtained by random selection. For *iTrust* the result is even more evident because from 10% of Random it jumps to 98% of *XACML smart coverage*. These data positively reply to RQ2 for the first case: *XACML smart coverage* selection

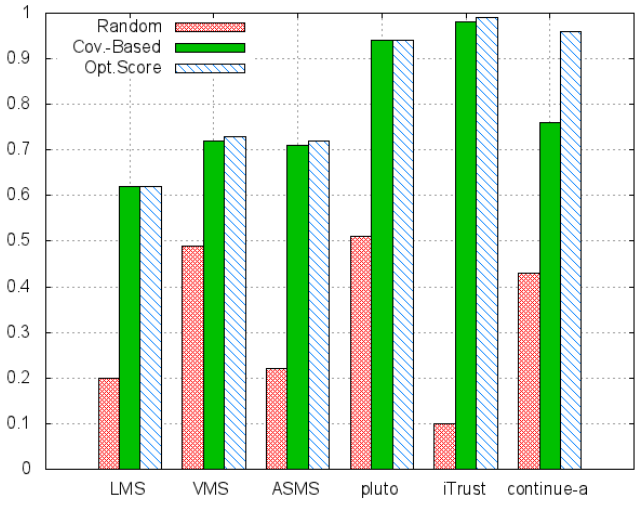

Figure 1. Mutation Analysis Results

can be considered a very good improvement with respect to random selection in terms of fault detection capability.

For the second case, we compared the loss in fault detection effectiveness of the *XACML smart coverage* selection approach with that of an optimal test cases selection technique. By applying a greedy algorithm, we selected the test cases yielding the highest number of mutants killed at each request (the mutation results) so to obtain a test suite, called *Opt. Score* in the table, having the same cardinality of that derived using the *XACML smart coverage* selection. Of course, because the optimal selection relies on the knowledge of the mutation score of each test case, the *Opt. Score* test suite has only experimental validity. It represents the upper bound of the fault detection effectiveness reachable by the *XACML smart coverage* selection. For the sake of consistency, as for random experiment, we performed the optimal test suite 10 times and computed the average number of killed mutants on the 10 runs. As shown in the table, for three of the considered policies, the fault detection effectiveness of the *XACML smart coverage* selection test suite has a loss of 1% with respect to that of the *Opt. Score* one. For *pluto* and *LMS*, the *XACML smart coverage* selection has the same performance than *Opt. Score* [4], while for the remaining *continue-a* the loss is equal to 20%. Considering RQ2, the results evidenced that the *XACML smart coverage* test selection is a good approximation of the optimal solution.

The results collected in this experiment, and sketched in Figure 1, positively reply to RQ1 and RQ2 and hint that the *XACML smart coverage* selection approach can be a valid methodology for reducing the tests suite while guaranteeing an acceptable loss in fault detection effectiveness.

[4]Note that the *Opt. Score* is a mean value computed over 10 runs.

V. Discussion

This section discusses the limitations of the current approaches and potential strategies for improving the selection strategy. In addition, it discusses the threats to validity.

A. Limitations and Improvements

According to the experiments, the coverage-based selection enables reaching relatively high mutation score (from 62% to 98%). This shows that for some policies, there are still a lot of mutants (38% in the worst case) that are not killed by the selected test cases. It is important to note that the remaining 38% mutants are not equivalent mutants because we only consider the mutants that are killed by the whole test suite. We need therefore to improve the selection approach in order to achieve better mutation scores and enable killing those remaining mutants. We are currently investigating other selection strategies that, as for *XACML smart coverage*, do not require running all the test cases before making the selection, but rely on the policy structure and on the requests content.

B. Threats to validity

The threats to external validity mainly relate to the fact that the six policies used in the experiments may not be representative of true practice. Therefore further experiments on a larger set of policies may be required to reduce these threats. In fact, several other real policies were available to us. However, they included a quite small set of tests (they had 2 or 3 rules only). In those cases, test selection did not make much sense because the policies are quite small and they could easily be checked manually. On the other hand, our six policies have quite different structures. Some have relatively few rules, while others have a large number of rules. For some policies the number of resources is bigger than the number of subjects (while for others it is not the case). Therefore, we are confident in the general relevance of the results. In addition, the random test case selection could produce very different results (it could be a very high or a very low mutation score), which would not give a correct view on the effectiveness of this approach. To address this threat, we repeated the process of random selection 10 times and computed the average mutation score. This enables us to evaluate correctly the effectiveness of the random test cases selection. Furthermore, our current version of the coverage criteria does not take into consideration the combining algorithms, which play an important role when it comes to selecting which rule applies in case of conflicts. Therefore, for some cases, it is important to consider the combining algorithm at policy and policy set level. We are aware of this limitation and plan to improve the coverage criterion by taking this into consideration. It is however unclear whether combining algorithms have an important impact on the quality of the selected test cases in terms of fault detection effectiveness. This issue should therefore be

investigated by using other policies having many conflicts. In addition, the effectiveness of the selection approaches was evaluated based on mutation analysis. As always when mutation is used, there is the issue whether the artificial faults represent or not real faults. However, in our approach, we have combined three different sets of mutation operators, which are implemented by the XACMUT tool [12]. We are confident in the quality of the mutation operators even though it would be interesting to perform a large empirical study to assess the quality of access control mutation operators.

VI. Related Work

The work presented in this paper spans over two main research directions: coverage criteria and test cases selection, and access control testing.

Coverage Criteria and Test Case Selection: The work in [17] provides a survey on test adequacy criteria presenting code coverage as a good criterion for test cases selection and test suite effectiveness evaluation. Many frameworks for test coverage measurement and analysis have been proposed dealing with different programming languages. The work closer to our proposal is [6] where authors provide a first coverage criterion for XACML policies defining three structural coverage metrics targeting XACML policies, rules and conditions respectively. As in our work, these coverage metrics are used for reducing test sets and the effects of test reduction in terms of fault detection are measured. With respect to the coverage approach proposed in [6], our proposal also addresses the policy set and does not require the policy execution and PDP instrumentation to be applied, then reducing the effort for coverage measurement. However, while the work in [6] provides structural coverage of the policy elements, our proposal relies on the Rule Target Set concept and the inclusion of the request values in that Rule Target Set, since according to the XACML language, only requests matching the Rule Target Set could provide an evaluation of that policy rule. Many proposals address test cases selection for regression systems. The work in [18] presents a survey of selection techniques able to identify the test cases that are relevant to some set of changes and addresses the emerging trends in the field. The work of [7] addresses three regression test selection techniques for security policies, based on mutation analysis, coverage analysis and recorded request evaluation respectively. They can be applied to XACML based systems and can reveal regression faults caused by policy changes, thus reducing the number of test cases. Differently from this work, our proposal does not target regression systems and does not require the execution of test cases against the security policy for selecting test cases, then reducing cost and time effort of the overall testing process. Code coverage criteria are also addressed by most of the techniques of test cases prioritization [19] [20] [21], with the aim to reorder test cases so that those tests that have a higher priority are executed before the ones having a lower priority [22], [23]. Differently from these works, the target of our proposal is not test cases prioritization but test cases selection, i.e. to choose a subset of test cases from the overall test suite.

Testing Access Control Systems: Automated test cases generation solutions have been proposed for testing either the XACML policy or the PDP implementation. The Targen tool [2] generates test inputs using combinatorial coverage of the truth values of independent clauses of XACML policy values. This approach has been proven to be more effective than random generation strategy in terms of structural coverage of the policy and fault detection capability [2]. The already mentioned X-CREATE tool [15], [3], [16] provides different strategies based on combinatorial approaches of the subject, resource, action and environment values taken from the XACML policy for deriving the access requests. A comparison of X-CREATE test generation strategies in terms of fault detection effectiveness is presented in [3], [16]. Among the X-CREATE generation strategies we used in this paper *Simple Combinatorial*. This strategy is easy-to-apply and at the same time able to reach the coverage of the policy input domain represented by the policy values combinations. More detail about this strategy were presented in Section IV-B2. Other test cases generation strategies [8] deal with combinatorial approaches of the elements of the model (role names, permission names, context names). Such approaches automatically derive abstract test cases that have to be then refined into concrete XACML requests for being executed on a PDP. A different approach is provided by Cirg [24] that is able to exploit change-impact analysis for test cases generation starting from policy specification.

To evaluate the effectiveness of the generated test suites, mutation analysis has been applied on access control policies [13], [14], [12]. The work of [13] has been the first attempt to define a fault model for access control policies and a set of mutation operators manipulating the predicates and logical constructs of target and condition elements of an XACML policy. The authors of [14] extend the mutation operators in [13], focusing on the use of a metamodel that allows to simulate the faults in the security models independently from the used role-based formalism (R-BAC or OrBAC). Finally, the work in [12] includes and enhances the mutation operators of [13] and [14] providing the XACMUT tool adopted in this paper.

VII. Conclusion and Future Work

In this paper, we have presented a new approach that aims at selecting access control test cases for testing XACML policies. This new approach relies on an *XACML coverage* criterion. The approach is efficient in terms of test case reduction, while reaching high mutation killing scores. In fact, the approach reduces drastically the test suite (an average of 15% of tests are selected). According to the experiments that

we performed, the proposed coverage-based test selection approach performs much better than random selection and its mutation score is close to the optimal one.

Although high mutation scores are already reached by the selected test sets, for future work we are currently investigating new approaches to further improve the test selection criteria to increase the mutation score and kill the remaining mutants. In fact, in some cases, it can be important to select more test cases to reach higher mutation scores (specially when more testing resources are available). The current proposal does not provide support for improving the mutation score beyond the coverage criteria.

In addition, as future work, we plan to perform an empirical study to compare our approach to other similar approaches [25], [26]. In addition, other approaches (like for instance [24], [27], [28]), which are more generic than ours could be also taken into consideration and compared to our proposed approach.

ACKNOWLEDGMENT

This work has been partially funded by the EC FP7 Network of Excellence NESSoS No. 256980.

REFERENCES

[1] OASIS, "extensible access control markup language (xacml) version 2.0," February 1 Feb 2005.

[2] E. Martin and T. Xie, "Automated Test Generation for Access Control Policies," in *Supplemental Proc. of 17th International Symposium on Software Reliability Engineering (ISSRE)*, November 2006.

[3] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, "Automatic XACML requests generation for policy testing," in *Proc. of The Third International Workshop on Security Testing*, 2012, pp. 842–849.

[4] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[5] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong., "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proc. of International Conference on Software Maintenance*, 1998, pp. 34–43.

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

[7] J. Hwang, T. Xie, D. El Kateb, T. Mouelhi, and Y. Le Traon, "Selection of regression system tests for security policy evolution," in *Proc. of the 27th International Conference on Automated Software Engineering*. ACM, 2012, pp. 266–269.

[8] A. Pretschner, T. Mouelhi, and Y. L. Traon, "Model-based tests for access control policies," in *Proc. of First International Conference on Software Testing, Verification (ICST)*, 2008, pp. 338–347.

[9] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proc. of IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 1999, pp. 179–188.

[10] K. Maly, M. Zubair, M. Nelson, X. Liu, H. Anan, J. Gao, J. Tang, and Y. Zhao, "Archon - a digital library that federates physics collections."

[11] Realsearch Group at NCSU, "iTrust: Role-Based Healthcare," http://agile.csc.ncsu.edu/iTrust/wiki/doku.php.

[12] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti., "XACMUT: XACML 2.0 Mutants Generator," in *Proc. of 8th International Workshop on Mutation Analysis*, 2013, pp. 28–33.

[13] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proc. of 16th International Conference on World Wide Web (WWW)*, pp. 667–676.

[14] T. Mouelhi, F. Fleurey, and B. Baudry, "A generic metamodel for security policies mutation," in *Proc. of Software Testing Verification and Validation Workshop (ICSTW)*, 2008, pp. 278–286.

[15] A. Bertolino, F. Lonetti, and E. Marchetti, "Systematic XACML Request Generation for Testing Purposes," in *Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, 2010, pp. 3–11.

[16] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, and L. Schilders, "Automated testing of extensible access control markup language-based access control systems," *IET Software*, vol. 7, no. 4, pp. 203–212, 2013.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[18] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430

[19] A. Kaur and S. Goyal, "A genetic algorithm for regression test case prioritization using code coverage," *International journal on computer science and engineering*, vol. 3, no. 5, pp. 1839–1847, 2011.

[20] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proc. of 14th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2003, pp. 442–453.

[21] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proc. of the 2006 International Symposium on Software Testing and Analysis*. ACM, 2006, pp. 1–12.

[22] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 102–112, 2000.

[23] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[24] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *Proc. of the Third International Workshop on Software Engineering for Secure Systems*, 2007, pp. 5–12.

[25] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proc. of the 13th ACM symposium on Access control models and technologies*, 2008, pp. 195–204.

[26] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance checking of access control policies specified in XACML," in *31st Annual IEEE International Computer Software and Applications Conference*, vol. 2, 2007, pp. 275–280.

[27] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff, "An approach to modular and testable security models of real-world health-care applications," in *Proc. of the 16th ACM Symposium on Access Control Models and Technologies*, 2011, pp. 133–142.

[28] V. C. Hu, D. R. Kuhn, and T. Xie, "Property verification for generic access control models," in *Proc. of IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, vol. 2, 2008, pp. 243–250.