Noname manuscript No.
(will be inserted by the editor)

XACMET: XACML Testing & Modeling

An Automated Model-based Testing Solution for Access Control Systems

Said Daoudagh · Francesca Lonetti · Eda Marchetti

Received: date / Accepted: date

Abstract In the context of access control systems, testing activity is among the most adopted means to assure that sensible information or resources are correctly accessed.

In XACML-based access control systems, incoming access requests are transmitted to the Policy Decision Point (PDP) that grants or denies the access based on the defined XACML policies. The criticality of PDP component requires an intensive testing activity consisting in probing such component with a set of requests and checking whether its responses grant or deny the requested access as specified in the policy. Existing approaches for improving manual derivation of test requests such as combinatorial ones do not consider policy functions semantics and do not provide a verdict oracle.

In this paper, we introduce XACMET, a novel approach for systematic generation of XACML requests as well as automated model-based oracle derivation. The main features of XACMET are: i) it defines a typed graph, called the XAC-Graph, that models the XACML policy evaluation; ii) it derives a set of test requests via full-path coverage of this graph; iii) it derives automatically the expected verdict of a specific request execution by executing the corresponding path in such graph; iv) it allows to measure coverage assessment of a given test suite. Our validation of the XACMET prototype implementation confirms the effectiveness of the proposed approach.

S. Daoudagh
ISTI-CNR, University of Pisa
E-mail: said.daoudagh@isti.cnr.it

F. Lonetti
ISTI-CNR
E-mail: francesca.lonetti@isti.cnr.it

E. Marchetti
ISTI-CNR
E-mail: eda.marchetti@isti.cnr.it

Keywords Access Control · Testing · Request Generation · Automated Oracle Derivation

1 Introduction

Security is a primary concern in modern pervasive and interconnected distributed systems. An important security aspect is constituted by *access control policies*, which specify which *subjects* can access which *resources* under which *conditions*. They are usually written using the eXtensible Access Control Markup Language (XACML) [28], an XML-based standard language proposed by OASIS, and rely on a specific architecture: incoming access requests are transmitted to the Policy Decision Point (PDP) that grants or denies the access based on the defined XACML policies. The criticality of the PDP component, as explained in [14], imposes an accurate testing activity that mainly consists of probing the PDP with a set of XACML requests and checking its responses against the expected decisions.

In literature, there are different proposals for automating PDP testing, including: mutation [25], coverage [26], random, combinatorial [23,5] and model-based [35] techniques.

Strictly speaking, the first two approaches provide an adequacy criterion for deciding on the thoroughness of the executed set of test requests, which have to be somehow generated. Combinatorial approaches to PDP testing seem convenient because they can automatically derive all combinations of policy parameters up to a given strength, and have been shown to outperform random testing in terms of fault-detection capability [23]. However, combinatorial approaches for PDP testing present some drawbacks:

- lack of oracle: for the generated requests, the expected PDP decision is not provided. This is an important limitation, especially when test suites are large and manual inspection of results is unfeasible. Li et al. [19] proposed to implement a PDP automated oracle through voting, i.e. to locally or remotely access more than one PDP engine and collect their responses for the same request. The most frequent decision value is considered the correct one. Although effective, this solution has a high computation and implementation cost and could not be applied in low energy consuming environments. Other proposals, for instance [10], strictly bind the oracle definition to the proposed test generation approach and do not provide generic solutions able to evaluate any kind of requests.
- shallow analysis: as we discuss in more detail in Section 3.1,existing approaches cannot take into account neither the semantics of the XACML functions, nor how the rules are combined.
- large number of test requests: as with any combinatorial approach, the number of requests generated can rapidly grow. Due to the increasing size and complexity of XACML policies, we believe that the huge number of derived requests in perspective could undermine the applicability of combinatorial approaches. Usually, heuristics are advised to limit the number

of generated requests. However, the risk exists that the applied reduction may decrease test effectiveness.

To overcome the above limitations, in this paper, we introduce XACMET (XACML Modeling & Testing), a novel model-based approach to support the automation of XACML-based testing [1]. Intuitively, XACMET builds from the XACML specification a typed graph, called the XAC-Graph, representing the XACML policy evaluation. Therefore, with respect to the existing solutions, such graph can be exploited for several purposes:

- defining a new test case generation strategy able to reduce the size of the test suite guaranteeing the same test suite effectiveness of the most adopted existing approaches. In particular, concerning the number of test requests, in our approach this is tied to the complexity of policy structure. Thus, even though we cannot *a priori* guarantee a low number, by construction all requests we generate correspond to different user's access modes.
- the automatic derivation of an XACML oracle based on the evaluation paths of the XAC-Graph. Since the XAC-Graph represents the expected verdict corresponding to each test request, we can automatically derive the test oracle. A preliminary version of the oracle derivation approach has been presented in [3].
- measuring the coverage of test requests in terms of the paths executed on the XAC-Graph. It can also help in deriving an adequate reduction or prioritization of the set of test requests such that all paths are executed at least once.

To the best of our knowledge, the test case generation as well as the oracle derivation are the most novel features supported by XACMET. In particular, concerning the oracle derivation, the proposed approach is the first completely automated model-based oracle for XACML-based PDP testing. XACMET represents an alternative approach for oracle derivation with respect to the well-known voting mechanism presented in [19]. We refer to [9] for the other features of XACMET.

In this paper, we also show the effectiveness of the proposed approach by empirical validation on real policies.

In summary, the contributions of this paper include: i) the definition of the XAC-Graph for modeling XACML policies; ii) a new strategy for test case generation; iii) an automated XACML oracle based on the evaluation paths of the XAC-Graph; iv) an automated methodology for measuring the coverage of test requests on the different policy paths, and v) an empirical evaluation of the XACMET approach against commonly adopted testing solutions.

The rest of this paper is structured as follows. Section 2 briefly introduces XACML. Section 3 overviews the basic idea behind the XACMET while a formal definition of the approach is provided in Section 4. Section 5 reports the validation of the proposal. Finally, Section 6 puts our work in context of related work and Section 7 draws conclusions.

[1] The tool is available at http://labsedc-wiki.isti.cnr.it/labsedc/tools/xacmet/public/main

2 XACML

XACML[2] [28] is a platform-independent XML-based standard language for the specification of access control policies. Briefly, an XACML policy has a tree structure whose main elements are: PolicySet, Policy, Rule, Target and Condition. The PolicySet includes one or more policies. A Policy contains a Target and one or more rules. The Target specifies a set of constraints on attributes of a given request. The Rule specifies a Target and a Condition containing one or more boolean functions. If a request satisfies the target of the policy, then the set of rules of the policy is checked, else the policy is skipped. If the Condition evaluates to true, then the Rule's Effect (a value of Permit or Deny) is returned, otherwise a NotApplicable decision is formulated (Indeterminate is returned in case of errors). More policies in a policy set and more rules in a policy may be applicable to a given request. The PolicyCombiningAlgorithm and the RuleCombiningAlgorithm define how to combine the results from multiple policies and rules respectively in order to derive a single access result.

For example, the *first-applicable* rule combining algorithm returns the effect of the first applicable rule or *NotApplicable* if no rule is applicable to the request. The *deny-overrides* algorithm specifies that *Deny* takes the precedence regardless of the result of evaluating any of the other rules in the combination, then it returns *Deny* if there is a rule that is evaluated to *Deny*, otherwise it returns *Permit* if there is at least a rule that is evaluated to *Permit* and all other rules are evaluated to *NotApplicable*. Similarly, the *permit-overrides* algorithm returns *Permit* if there is a rule that is evaluated to *Permit*.

```
1  <Policy PolicyId="Pol_Ex" RuleCombiningAlgId="deny-overrides">
2   <Target/>
3   <Rule RuleId="rule1" Effect="Deny">
4    <Target>
5     <Resources><Resource><ResourceMatch MatchId="string-equal">
6       <AttributeValue DataType="string">book</AttributeValue>
7       <ResourceAttributeDesignator AttributeId="resource-id" DataType="
            string"/>
8      </ResourceMatch></Resource>
9      <Resource><ResourceMatch MatchId="string-equal">
10      <AttributeValue DataType="string">document</AttributeValue>
11      <ResourceAttributeDesignator AttributeId="resource-id" DataType="
            string"/>
12     </ResourceMatch></Resource>
13     <Resource><ResourceMatch MatchId="string-equal">
14      <AttributeValue DataType="string">documententry</AttributeValue>
15      <ResourceAttributeDesignator AttributeId="resource-id" DataType="
            string"/>
16     </ResourceMatch></Resource></Resources>
17    <Actions><Action><ActionMatch MatchId="string-equal">
18      <AttributeValue DataType="string">write</AttributeValue>
19      <ActionAttributeDesignator AttributeId="action-id" DataType="string"/>
20     </ActionMatch></Action></Actions>
21   </Target>
22   <Condition><Apply FunctionId="string-is-in">
23     <Apply FunctionId="string-one-and-only">
24      <ResourceAttributeDesignator AttributeId="resource-id" DataType="
            string"/>
25     </Apply>
```

[2] The current implementation of the presented approach is compliant with XACML 2.0 but it can be easily extended to address the functionalities of XACML 3.0.

```
26      <SubjectAttributeDesignator AttributeId="subject-id1" DataType="string"
            />
27     </Apply>
28    </Condition>
29   </Rule>
30   <Rule RuleId="ruleB" Effect="Permit">
31    <Target>
32     <Subjects><Subject><SubjectMatch MatchId="string-equal">
33       <AttributeValue DataType="string">Julius</AttributeValue>
34       <SubjectAttributeDesignator AttributeId="subject-id" DataType="string"
             />
35     </SubjectMatch></Subject></Subjects>
36     <Resources><Resource><ResourceMatch MatchId="string-equal">
37       <AttributeValue DataType="string">journals</AttributeValue>
38       <ResourceAttributeDesignator AttributeId="resource-id" DataType="
             string"/>
39     </ResourceMatch></Resource></Resources>
40     <Actions><Action><ActionMatch MatchId="string-equal">
41       <AttributeValue DataType="string">read</AttributeValue>
42       <ActionAttributeDesignator AttributeId="action-id" DataType="string"/>
43     </ActionMatch></Action></Actions>
44    </Target>
45   </Rule>
46  </Policy>
```

Listing 1 An XACML policy

We show in Listing 1 an example of a simplified XACML policy ruling library access. Its target (line 2) says that this policy applies to any subject, resource and action. This policy has a first rule, *ruleA* (lines 3-29), with a target (lines 4-21) specifying that this rule applies only to the access requests of a "write" action of "book", "document" and "documententry" resources. The rule condition will be evaluated true when the request resource value is contained into the set of request subject values. The effect of the second rule *ruleB* (lines 30-45) is *Permit* when the subject is "Julius", the action is "read", and the resource is "journals". The rule combining algorithm of the policy (line 1) is *deny-overrides*.

3 XACMET: Motivations and Ideas

In this section, by using as an example the policy of Listing 1, we provide the main motivations and ideas behind the XACMET approach (which is formally defined in Section 4). In particular, in the following subsections we focus on the three main features provided by the proposal: test case generation, oracle derivation and path coverage measure.

3.1 Test Case Generation

Considering in particular the test case generation, for aim of simplicity, we compare XACMET approach with the widespread combinatorial proposals. We do not consider other more sophisticated approaches which have been proposed, that apply for example change-impact analysis [24] or model-based testing techniques (e.g., [35,13]), as these are not completely automatic as

XACMET. Indeed to be applied they require the intervention of the policy tester, either to spot ad-hoc changes or to define the most suitable test criteria [14].

The main idea under the XACMET test case generation approach relies on the semantics of XACML functions. Indeed, some XACML rules to be satisfied may require that detailed conditions are met in the request.

For instance, considering the policy of Listing 1, the condition of *ruleA* consists of the combination of two functions, namely: *(i)* only one resource must be specified into the request (for the *string-one-and-only* function semantics) and *(ii)* the resource value must be contained into the set of the subject values (for the *string-is-in* function semantics). To test this rule, a request should satisfy both constraints and the rule target, i.e. the resource value in the request needs to be defined into the resources of the target of the rule.

Random or combinatorial test policy approaches are shallow with respect to the semantics of the XACML functions specified in the target and condition elements. They only consider the parameters values, therefore, a request generated by those approaches could satisfy very specific conditions such as this only by chance.

The XACMET approach in contrast derives requests to expressly satisfy all constraints specified into all target and condition elements of a policy by construction, in completely automatic way. Specifically, Listing 2 shows the request generated by XACMET to satisfy the referred rule.

```
1  <Request xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
2   <Subject>
3    <Attribute AttributeId="subject-id1" DataType="string">
4     <AttributeValue>documententry</AttributeValue>
5    </Attribute>
6   </Subject>
7   <Resource>
8    <Attribute AttributeId="resource-id" DataType="string">
9     <AttributeValue>documententry</AttributeValue>
10   </Attribute>
11  </Resource>
12  <Action>
13   <Attribute AttributeId="action-id" DataType="string">
14    <AttributeValue>write</AttributeValue>
15   </Attribute>
16  </Action>
17  <Environment/>
18 </Request>
```

Listing 2 An XACML request

Another important aspect considered during the test case generation is the evaluation of the rule combining algorithms. Indeed, the result of the evaluation of an XACML policy strictly depends on the rule combining algorithm, that prioritizes rules evaluation.

For instance, let us consider the policy of Listing 1 where the intended rule combining algorithm is *Deny-overrides* (line 1), and let us assume a faulty implementation yielding instead *Permit-overrides*. With reference to the two rules *ruleA* and *ruleB*, the possible situations for an incoming test request are:

1. it satisfies *ruleA* but not *ruleB* :

   - the faulty policy returns *Deny*, i.e., the effect of the first rule, because the second one is not applicable;
   - the correct policy returns *Deny*, because it takes the precedence regardless of the result of the second rule;
2. it satisfies *ruleB* but not *ruleA*:
   - the faulty policy returns *Permit*, i.e., the effect of the second rule, because it takes the precedence regardless of the result of the first rule;
   - the correct policy returns *Permit*, i.e., the effect of the second rule, because the first rule is not applicable;
3. it does not satisfy neither rule:
   - both policies return *NotApplicable*;
4. it satisfies both rules:
   - the faulty policy returns *Permit* as in 2); i.e., the effect of the second rule, because it takes the precedence regardless of the result of the first rule;
   - the correct policy returns *Deny* as in 1). because it takes the precedence regardless of the result of the second rule.

Therefore, the fault corresponding to the rule combining algorithm would be detected only by a request fulfilling the fourth case. However, neither random nor combinatorial approaches focus on the application of the rule combining algorithm during the test request generation.

The XACMET approach that expressly takes into account the application of the rule combining algorithm during the request generation, systematically exercises all possible combinations of rule evaluations and thus guarantees to detect this kind of fault.

3.2 PDP Oracle

The second feature provided by the XACMET approach is the derivation of a PDP oracle. Usually, random and combinatorial solutions can automatically generate the test requests, but do not provide their expected responses. XACMET by contrast derives together with each test request its expected verdict, therefore alleviating the costs and risks of manual result inspection [14].

By referring to the example policy of Listing 1, in this section we explain the underling idea of the approach used for the oracle definition. We refer to Section 4.1 for formal details.

Given a generic request, the result of the evaluation of an XACML policy with that request strictly depends on: the request values, the policy constraints as well as the combining algorithm that prioritizes the evaluation of the policy rules as explained in Section 3.1. Specifically, we define an *evaluation path* as a sequence of policy elements that are exercised by the request during the evaluation of an XACML policy and the verdict associated to that request. Thus, the general idea of the XACMET approach is to derive all possible evaluation paths from the policy specification and order them according to the rule combining algorithm. For instance, let us consider the policy of Listing

1, having as elements, the rules *ruleA* and *ruleB* and *deny-overrides* as the combining algorithm (line 1), the possible evaluation paths are:

1. *ruleA* evaluated to true and *ruleB* evaluated to false: the associated verdict is *Deny*, i.e., the effect of the first rule;
2. *ruleA* evaluated to false and *ruleB* evaluated to true: the associated verdict is *Permit*, i.e., the effect of the second rule;
3. *ruleA* and *ruleB* both evaluated to false: the associated verdict is *NotApplicable*;
4. *ruleA* and *ruleB* both evaluated to true: the associated verdict is *Deny*, because it takes the precedence regardless of the result of the second rule.

This set of paths is ordered according to the semantics of the rule combining algorithm, and then according to the verdict associated to each path. For instance, in case of *deny-overrides* combining algorithm, first the paths having *Deny* are evaluated, then those having *Permit* and finally those having *NotApplicable*. For paths having the same verdict, the evaluation order of the paths is based on their length, namely the shortest path takes the precedence. For the policy of Listing 1, the order of the evaluated paths is (1), (4), (2) and (3).

The ordered set of paths is then used for the requests evaluation and the verdicts association. For each request, the first path for which all the path constraints are satisfied by the request values is identified and the final verdict associated to the request is derived.

3.3 Measuring Path Coverage

XACMET also provides the possibility of measuring the path coverage of a generic set of requests. Indeed, each of the evaluation paths represents the set of constraints that should be satisfied by some specific request values so to reach the final verdict. This information can be useful to improve the policy itself and avoid possible security flaws.

For instance, considering the ordered set of paths of policy of Listing 1, listed in Section 3.2, a request asking to `write` a `documententry` does not match paths (2), (3) and (4), but covers (satisfies) path (1) since it satisfies only *ruleA*, so it reaches a path coverage equal to 25%.

In literature, coverage metrics are the most used approaches for evaluating the quality of a test suite [33]. In the access control context, among the different coverage criteria, the path coverage is one of the hardest to be adopted; identifying all the possible paths of an XACML policy has the same complexity of the definition of a policy evaluation engine. In case of XACMET, the paths identification is the basis of the XACMET proposal itself and consequently the path coverage is the easiest coverage metrics to be adopted.

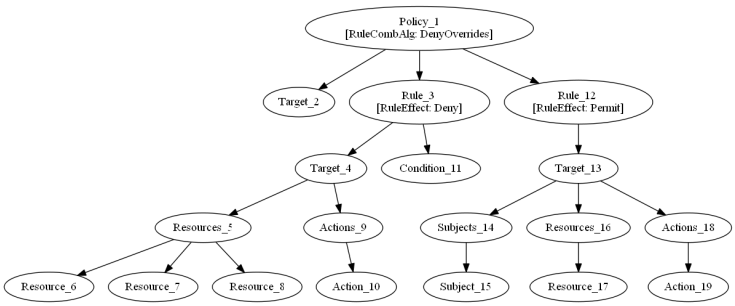

Fig. 1 XAC-Tree. Label T_P means node of type T and parameter P. The attributes are within square brackets.

4 Formal Definitions

In this section, we first provide some formal definitions used for representing the policy into a XAC-Graph (Section 4.1), then in Section 4.2 we provide the procedures used for visiting this graph, finally in Section 4.3 and Section 4.4 the algorithms for test cases generation and oracle derivation are presented respectively.

4.1 XAC-Graph

In this section, we provide some formal definitions related to the XAC-Graph model.

With reference to the XACML policy, as an XML document it can be represented as a tree, called the XAC-Tree. In particular, the following concepts can be used:

- *Contained:* Element i is contained within element j if i is between the start-tag and the end-tag of j.
- *Parent:* Element i is the parent of element j when j is contained within i and i is exactly one level above j.
- *Sibling*: The siblings in an XML document are the elements that are on a same level of the tree and have the same parent. In particular, given parent i of elements j and k, j is left (right) sibling of k if j is contained just before (after) k within element i.

The XAC-Tree derivation exploits the parent relationship of the XACML policy and uses the following sets of types and values:

- T_V = {Policy, Target Rule, Subjects, Subject, Resources, Resource, Actions, Action, Environments, Environment};
- T_{V_a} = {RuleAlgorithm, Effect, NotApplicable};
- T_{V_v} ={ReturnPermit, ReturnDeny, ReturnNotApplicable};
- `RCA`= {FirstApplicable, DenyOverrides, PermitOverrides};

– RE = {Permit, Deny}.

Formally, the parent relationship, called XAC-TreeParent, is defined as:

Definition 1 (XAC-TreeParent) *Given a tree T=(V, E, Root) in which every vertex v in V has an associated type $t_v \in T_V$: Element i ∈* V *is* XAC-TreeParent *of j ∈ V if i is parent of j in the XACML policy document.*

From this, the definition of the XAC-Tree as in the following:

Definition 2 (XAC-Tree) *Given an XACML policy, the XAC-Tree is a labeled and typed tree (V, E, Root) where*

- *V is a set of vertices such that every $v \in V$ has type $t_v \in T_V$;*
- *Each vertex v in V has parameter i with i = 1,.., n;*
- *E is a set of edges (i,j) such that i,j ⊆ V and i is* XAC-TreeParent *of j;*
- *Root is a vertex v in V with t_{Root} = Policy;*
- *Root has an attribute called* RuleCombAlg ∈ *RCA;*
- *Each vertex v ∈ V with t_v = Rule has an attribute called* EffectRule ∈ *RE.*

Figure 1 shows the XAC-Tree associated to the policy of Listing 1. In particular, *ruleA* becomes the node `Rule_3` into the XAC-Tree. In this case, 3 is the suffix of the node and the EffectRule attribute of the node `Rule_3` is set to Deny as specified in the Effect of *ruleA*. Moreover, there is a XAC-TreeParent relation between `Policy_1` and `Rule_3` nodes because *ruleA* is contained within `policyExample` in Listing 1.

The representation of the XACML policy is then used to derive a model of the XACML evaluation. For this, a parent relationships, called XAC-GraphParent, is defined as in the following:

Definition 3 (XAC-GraphParent) *Given a graph G= (V_g,E_g,Entry) and XT = (V,E,Root) a XAC-Tree, where Entry = Root and $V_g = V \bigcup V_a^\star \bigcup V_v^\star$ with (i) V_a is a set of vertices such that every* v *∈ V_a has type $t_v \in T_{V_a}$; (ii) V_v is a set of vertices such that every* v *∈ V_v has type $t_v \in T_{V_v}$.*

Element i ∈ V_g is in a XAC-GraphParent *relation with j ∈ V_g if:*

1. *i is left sibling of j in XT, $t_j \neq$ Rule and i has not children in XT;*
2. *i is leaf in XT and ∃ k ∈ V_g such that k is XAC-TreeParent of i and k is left sibling of j in XT and $t_j \neq$ Rule;*
3. *j has no left sibling in XT and i is XAC-TreeParent of j in XT;*
4. *t_j ∈ {Effect, NotApplicable}, i is leaf in XT, $t_i \neq$ Target, ∃ k ∈ V such that t_k=Rule and i is the rightmost leaf of the subtree rooted in k in XT;*
5. *t_i ∈ {Effect, NotApplicable} and t_j = RuleAlgorithm;*
6. *t_i= RuleAlgorithm, t_j = ReturnPermit if ∃ k∈ V, t_k=Rule such that the value of the attribute EffectRule of k is Permit;*
7. *t_i= RuleAlgorithm, t_j = ReturnDeny if ∃ k∈ V, t_k=Rule such that the value of the attribute EffectRule of k is Deny;*
8. *t_i= RuleAlgorithm and t_j = Rule, and if k ∈ V is the left sibling of j in XT than $t_k \neq$ Target.*

Finally, a labelled and typed graph, called the XAC-Graph, is defined as in the following:

Definition 4 (XAC-Graph) *Let XT = (V,E,Root) be a XAC-Tree, a policy graph (XAC-Graph) is a graph (V_g,E_g,Entry) where*

- *Entry = Root;*
- $V_g = V \bigcup V_a^\star \bigcup V_v^\star$
- *V_a is a set of vertices such that every $v \in V_a$ has type $t_v \in T_{V_a}$;*
- *V_v is a set of vertices such that every $v \in V_v$ has type $t_v \in T_{V_v}$;*
- *Each vertex v in V_a with t_v = Effect has an attribute called* EffectValue $\in$ *RE;*
- *Each vertex v in V_a has parameter i with i = 1,.., n;*
- *The vertex v in V_a with t_v = RuleAlgorithm is unique and has an attribute called* Algorithm *of value equal to the value of the attribute RuleCombAlg of the Root in XT;*
- *E is a set of edges (i,j) such that i,j $\sqsubseteq$ V_g and i is XAC-GraphParent of j;*
- *Each vertex j, j, k $\in$ V_g with t_j = Effect, i XAC-GraphParent for the point 4 of the definition 3 and t_k=Rule the value of the attribute EffectValue of j is equal to value of the attribute EffectRule of k.*

Figure 2 shows the XAC-Graph of the XAC-Tree of Figure 1. In particular, considering the conditions of Definition 3, in XAC-Graph there exists a XAC-GraphParent relation between:

- `Target_2` and `Rule_3` due to Condition 1;
- `Resource_7` and `Actions_9` due to Condition 2;
- `Policy_1` and `Target_2` due to Condition 3;
- `Condition_11` and `NotApplicable_3` due to Condition 4;
- `Condition_11` and `Effect_3` due to Condition 4;
- `Effect_3` and `RuleAlgorithm` due to Condition 5. Note that `Policy1` and `RuleAlgorithm` have the same algorithm value;
- `RuleAlgorithm` and `ReturnPermit` due to Condition 6;
- `RuleAlgorithm` and `ReturnDeny` due to Condition 7;
- `RuleAlgorithm` and `Rule_12` due to Condition 8.

The XAC-Graph could basically be derived applying a depth-first search approach to the XAC-Tree. For the sake of simplicity, we show the XACMET approach applied to a policy rooted in the Policy element. Definitions 1, 2, 3, 4, can be easily extended to also consider the *PolicySet* and the XACMET approach can be used also for deriving the XAC-Graph considering a policy rooted in the *PolicySet* node.

4.2 Visiting the XAC-Graph

In this section we detail the procedure adopted for visiting the XAC-Graph related to an XACML policy in order to derive both the set of test requests

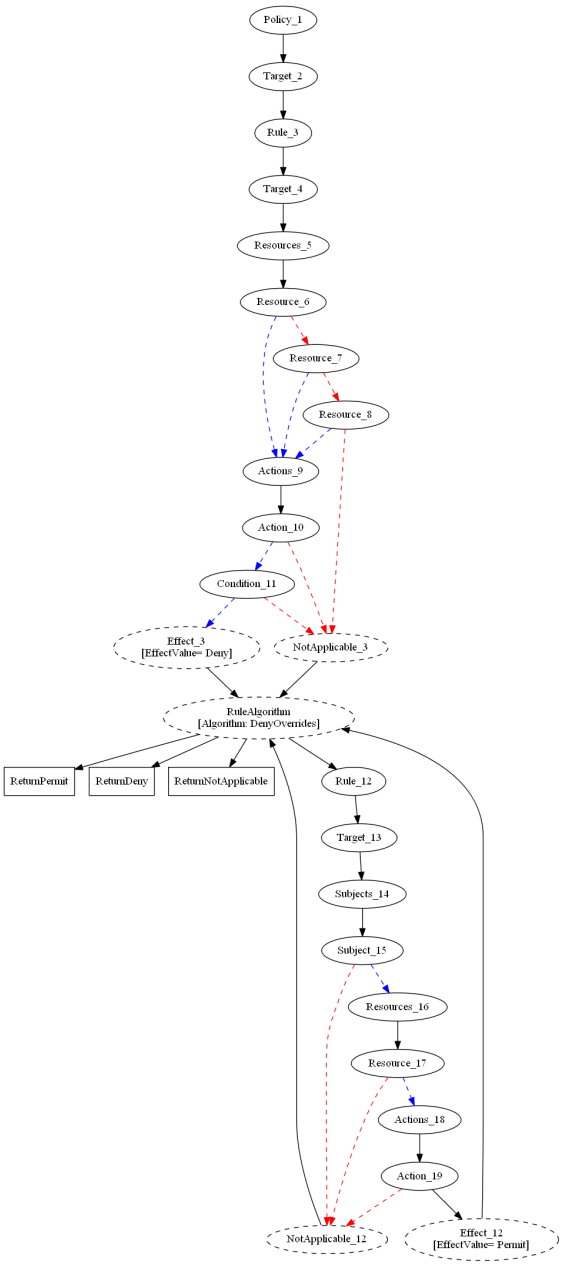

Fig. 2 XAC-Graph. Label T_P means node of type T and parameter P. The attributes are within square brackets.

that allow for the full coverage of the paths of XAC-Graph and the derivation of the oracle verdicts.

In both cases, the process adopted is divided into two main steps: *coloring* and *unfolding* as detailed in the following sections.

4.2.1 Coloring XAC-Graph

During the coloring step, the concept of *Forward Node* is adopted. In particular, given the XAC-Graph G = (V_g , E_g, Entry) for each node i $\in V_g$ it is possible to identity the Forward Node $FN(i) \in V_g$ as the set of nodes j such that i is a XAC-GraphParent of j. Consequently, it is possible to define for each node i $\in V_g$ the Forward Star $FS(i) \in E_g$ as the set of edges (i, j) where j $\in FN(i)$.

Thus, given a XAC-Graph for each node b and c $\in V_g$, with $t_b \in$ {Subject, Resource, Action, Environment}, the cardinality of $FN(b) = 2$, and $t_c \in$ {Subject, Resource, Action, Environment, NotApplicable}, the coloring process marks each edge (b,c)$\in FN(b)$: with red dashed line if $t_b = t_c$ or $t_c =$ NotApplicable; with blue dotted line otherwise.

In practice, the red dashed edges represent a successful evaluation of the node b.

4.2.2 Unfolding XAC-Graph

During the unfolding process the paths are obtained by visiting the XAC-Graph from the Entry node to each node in V_v. The cycles are due to the presence of the node typed `Rule_Algorithm`. In XACMET, the order of the paths strictly depends on the order in which the rules are evaluated, which in turn is guided by the FirstApplicable, DenyOverrides, PermitOverrides algorithms.

Thus let P a path of k nodes on the XAC-Graph and the h the last included one, with t_h =Rule_Algorithm. If the value of Algorithm attribute of h is equal to:

FirstApplicable and the node k-1 has type:

- Effect, and EffectValue = Deny (Permit), then the next node has type ReturnDeny (ReturnPermit);
- NotApplicable, then the next node v has type Rule iff v is not already included in P (ReturnNotApplicable otherwise).

DenyOverrides and the node k-1 has type

- Effect and EffectValue = Deny, then the next node has type ReturnDeny;
- Effect and EffectValue = Permit, then the next node v has type Rule iff v is not already included in P (ReturnPermit otherwise);
- NotApplicable, then the next node v has type Rule iff v is not already included in P;

– NotApplicable and each node v with Rule $\in P$ the next node has type ReturnPermit, if $\exists$ a node p $\in P$ with t_p=Effect (ReturnNotApplicable otherwise).

PermitOverrides and the node k-1 has type

– Effect and EffectValue = Permit, then the next node has type ReturnPermit;
– Effect and EffectValue = Deny, then the next node v has type Rule iff v is not already included in P (ReturnDeny otherwise);
– NotApplicable, then the next node v has type Rule iff v is not already included in P;
– NotApplicable and each node v with Rule $\in P$ the next node has type ReturnDeny, if $\exists$ a node p $\in P$ with t_p=Effect (ReturnNotApplicable otherwise).

In Figure 3, we show a path of XAC-Graph.

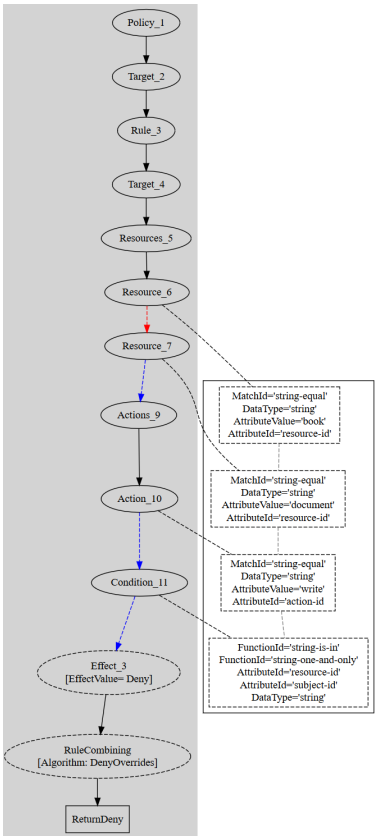

Fig. 3 A path of XAC-Graph. The boxes connected to the nodes contain the functions and values.

In the next two subsections we describe XACMET framework from the implementation point of view and we present two algorithms that implement two features provided by the proposed framework: the generation of XACML requests and the derivation of the oracle, i.e. the virtual PDP.

4.3 Generation of XACML Test Requests

In this section, we describe the algorithm conceived for the XACML requests generation purpose. Intuitively, for each path identified during the unfolding step, the generation of requests is based on the data associated to each node. In particular, given P a path of k nodes for each node i, the functions and the values collected are translated into the properties and constraints. If the outgoing edge is a blue dotted line (red dashed line), the set of values satisfying (not satisfying) the properties are identified using a constraint satisfaction approach. For all the nodes in a path P, the collected set of values are integrated and the values for test requests generation successively extracted.

Note that, as the various sets are combined, possible inconsistences between the selected values can be detected, which hint at the potential presence of unfeasible paths in the XAC-Graph. This peculiarity can be useful to highlight issues in the policy specifications and to improve the expressiveness of the policy itself.

In the remaining of this section, we present the algorithm for the generation of XACML requests. This algorithm, named *XACML Requests Generation* and detailed in Algorithm 1, receives the XACML policy as input and returns a set of XACML requests. Informally, the algorithm generates an XACML request for each feasible XAC-Path by satisfying all the constraints encoded in that XAC-Path.

Firstly, the algorithm derives a XAC-Tree by applying the concepts of XAC-TreeParent and XAC-Tree definition defined in Section 4.1 (Algorithm 1, line 4). Secondly, staring from the XAC-Tree, it derives the corresponding XAC-Graph (Algorithm 1, line 5) by applying the XAC-GraphParent relationship (Definition 3) and the XAC-Graph notion (Definition 4). Thirdly, it derives an ordered set of XAC-Paths by applying the Coloring XAC-Graph and Unfolding XAC-Graph procedures introduced in Section 4.2. In this step, (Algorithm 1, line 6) it also takes into account the combining algorithm defined in the XACML policy.

Note that, not all XAC-Paths guarantee the generation of an XACML request, this is due to the presence of two or more contradictory constraints within the same path. The idea underlined the algorithm is to transform the computation of a XAC-Path into a constraint satisfaction problem and to use constraint solving techniques to generate an XACML request (Algorithm 1, lines 8-10).

A Constraint Satisfaction Problem (CSP) consists of a set of variables and a set of constraints over the values of these variables [1]. Each variable has a domain of possible values. The constraints define a set of restrictions over

the possible variables values with respect to the already assigned ones. An assignment is a state of a CSP, in which some or all the variables have an assigned value. An assignment is called *consistent*, if the assigned values of the variables do not violate any constraint. It is called *complete*, if a value is assigned to every variable. An assignment that is consistent and complete is called a *solution*.

Among the existing techniques for solving CSPs [1], in the current implementation, we adopt the JaCoP solver [16] (Algorithm 1, line 10) to create a CSP instance (Algorithm 1, line 8), determine the feasibility of XAC-Path and consequently generate XACML requests (Algorithm 1, line 10). Indeed, JaCoP is a general purpose constraint solving library for Java which provides various ready to use variable and constraint types to define CSP instances and can be easily integrated in Java-based applications.

Finally, if a solution exists for the created CSP instance, the XAC-Path is considered feasible and the XACML request is derived by that solution (Algorithm 1, lines 11-15).

For instance, by referring to the first constraint of the XACML policy reported in Listing 1 (lines 5 - 7), namely *resource-id* = *book*, Algorithm 1 defines a model with two variables and one constraint. The first variable named *resourceIdVar* has the following domain: {article, book, journal} [3], the second variable named *bookVar* has the following domain: {book}, while the constraint over these variables is: *resourceIdVar* = *bookVar*. One of the possible requests derived by the solution of this constraint is presented in Listing 2.

Algorithm 1 XACML Requests Generation

```
1: input: XACMLPolicyP
2: output: XR                                   ▷ A set of XACML requests
3: XR ← {}
4: XacTree ← createXacTree(P)
5: XacGraph ← createXacGraph(XacTree)
6: XacPaths ← createXacPaths(XacGraph)
7: Foreach p_i ∈ XacPaths do
8:     JM ← JaCoPConstraintModel(p_i)
9:  ▷ Create a Java Constraint Model from the node of p_i containing XACML constraints
10:    xacRequest ← JaCoPConstraintSolver.solve(JM)
11:    if xacRequest ≠ NULL then
12:        XR.add(xacRequest)
13:    else
14:        infeasiblePath.add(p_i)
15:    end if
16: end for
17: return XR
```

[3] Note that whereas *book* is a resource value taken from the policy, *article* and *journal* are random values assigned by the algorithm.

4.4 Oracle Derivation

In this section, we describe the algorithm conceived for the oracle derivation. Informally, starting from an XACML policy, the algorithm derives a set of ordered XAC-Paths, each containing an expected result. Therefore, for a given request it is able to identify the first path satisfied by that request and then associates to it an expected result. Algorithm 2 takes as input an XACML policy P and a set of XACML requests $XR = \{xr_1, \ldots, xr_n\}$ and returns a set of expected results for these requests.

Firstly, it derives a set of XAC-Paths $XacPaths = \{xp_1, \ldots xp_k\}$, using: i) the concepts of XAC-TreeParent and XAC-Tree (Algorithm 2, line 4); ii) the XAC-GraphParent relationship and the XAC-Graph notion (Algorithm 2, line 5); iii) the Coloring XAC-Graph and Unfolding XAC-Graph procedures (Algorithm 2, line 6).

Then, for each request xr_i and for each XAC-Path xp_j, the algorithm derives a constraint model (Algorithm 2, line 9). This model takes into account both the constraints encoded within the constrained nodes of the path xp_j and the values contained in the request xr_i (Algorithm 2, line 9). This model represents a CSP instance. If a solution exists for this instance, the request xr_i satisfies the XAC-Path xp_j, and then the associated verdict is returned (Algorithm 2, lines 9-15).

For instance, by referring to the XACML policy reported in Listing 1, the XAC-Path of Figure 3 and a request having *resource* and *subject* values equal to *document* and *action* value equal to *write*, the algorithm firstly derives a constraint model including: i) the constraints of the boxes in the right side of Figure 3; and ii) the values of the request. Then, the algorithm solves the constraint model (Algorithm 2, line 9-11) and derives the final verdict for the request that in this case is *Deny*.

Algorithm 2 XACML Oracle

```
1: input: P, XR                    ▷ A set XACML Policy P and a set of XACML requests XR
2: output: ER                                          ▷ A set of expected decisions
3: ER ← {}
4: XacTree ← createXacTree(P)
5: XacGraph ← createXacGraph(XacTree)
6: XacPaths ← createXacPaths(XacGraph)
7: Foreach xr_i ∈ SR do
8:     Foreach p_j ∈ XacPaths do
9:         JM ← JaCoPConstraintModel(p_j, xr_i)
10:                ▷ Create a Java Constraint Model from the node of p_j and values in xr_i
11:        result ← JaCoPConstraintSolver.solve(JM)
12:        if result = TRUE then
13:            xr_i.setDecision(p_j.DECISION)
14:            ER.add(p_j.DECISION)
15:            continue
16:        end if
17:    end for
18: end for
19: return ER
```

Table 1 Java Constraint Model

Variables	
Variable	Domain
bookVar	{book}
documentVar	{document}
resourceIdVar	{document}
actionIdVar	{write}
writeVar	{write}
subjectIdVar	{document}
Constraints	
Constraint Name	Constraint
Resource_6	resourceIdVar $\neq$ bookVar
Resource_7	resourceIdVar = documentVar
Action_10	actionIdVar = writeVar
Condition_11	resourceIdVar = subjectIdVar

5 Experiment

In this section, we provide experimental results related to the different three features of XACMET: test case generation, oracle derivation and path coverage measure.

5.1 XACMET *vs* X-CREATE

The objective of the study presented in this section is to assess the effectiveness of XACMET test case generation strategy in terms of fault-detection capability. In particular, we provide in this section a comparison between the tool X-CREATE [5] and XACMET.

Several common approaches for generating XACML requests are based on combinatorial strategies, as surveyed in Section 6. Among them, we selected X-CREATE as a baseline for comparison of results, because it has been proven to perform better than other combinatorial request generation techniques [7]. In particular, we use the Multiple Combinatorial test strategy implemented in this tool for deriving the test suites used to empirically validate the effectiveness of XACMET test cases generation approach.

The Multiple Combinatorial strategy relies on combinatorial approach of the policy values. Specifically, four data sets called SubjectSet, ResourceSet, ActionSet and EnvironmentSet are defined and filled with the values and the attributes of *Subjects*, *Resources*, *Actions* and *Environments* elements of the policy respectively. Then, for each set, all possible subsets are derived. The test requests are then generated by combining the subject, resource, action and environment subsets using Pairwise, Threewise, Fourwise combinatorial approaches. We refer to [5] for a more detailed description.

We considered a set of real-world XACML policies taken form real contexts and European projects as summarized in Table 2. The considered poli-

cies were: i) the Fedora (Flexible Extensible Digital Object Repository Architecture) XACML policies (demo-5, demo-11 and demo-26) [4] and other six policies released in the context of the TAS3 European project [5]. In particular, the columns of Table 2 represent the number of rules, conditions, subjects, resources, actions and distinct functions within each policy.

To measure the fault-detection effectiveness, we applied mutation analysis, which in software testing is a standard technique to asses a test suite [30]. A set of java based mutation operators is selected and applied directly to the code of the PDP to derive mutants. A test suite is then executed on these mutants to measure its effectiveness. Specifically, we consider two available sets of java based mutation operators implemented in the *mu*Java tool [22]: the traditional-method level mutation operators and the class-level ones. The former introduce faults at the level of methods by seeding faults directly into internal code statements while the latter are specific to object-oriented features of Java. Table 3 shows the operator name and a brief description of the selected set of mutation operators. We refer to [11] for a detailed description of the mutants generation approach.

Table 2 XACML Policies Subjects.

Xaml Policy	Functionality					
	# Rule	# Cond	# Sub	# Res	# Act	# Funct
2_73020419964_2	6	5	3	3	0	4
create-document-policy	3	2	1	2	1	3
demo-5	3	2	2	3	2	4
demo-11	3	2	2	3	1	5
demo-26	2	1	1	3	1	4
read-document-policy	4	3	2	4	1	3
read-informationunit-policy	2	1	0	2	1	2
read-patient-policy	4	3	2	4	1	3
Xacml-Nottingham-Policy-1	3	0	24	3	3	2

Considering the distribution of mutants in Sun PDP evaluation engine, for each of the Java Package, Table 4 reports the cardinality of the mutants set (fourth column) and their distributions at class-level and method-level (second and third column). As reported in the last row, the 76% (i.e. 6085 over 8030 mutants) of the mutants generated are derived by the application of the method-level mutation operators.

Table 5, in the second and third columns, reports the test suite effectiveness of X-CREATE and XACMET respectively, defined in terms of percentage of mutated PDPs killed. As can be evidenced by the effectiveness values for each of the nine policies, there is no substantial difference from the point of

[4] Fedora Commons Repository Software. http://fedora-commons.org.

[5] Trusted Architecture for Securely Shared Services. http://www.tas3.eu.

Table 3 Java Mutation operators

Mutation Operator	Description
	class-level mutation operators
EAM	Accessor method change
EMM	Modifier method change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IPC	Explicit call to a parent's constructor deletion
ISI	super keyword insertion
JDC	Java-supported default constructor deletion
JID	Member variable initialization deletion
JSD	static modifier deletion
JSI	static modifier insertion
JTD	this keyword deletion
JTI	this keyword insertion
OAC	Arguments of overloading method call change
OMR	Overloading method contents replace
PCC	Cast type change
PCI	Type cast operator insertion
PNC	new method call with child class type
PRV	Reference assignment with other comparable variable
	method-level mutation operators
AODS	Delete short-cut arithmetic operators
AODU	Delete basic unary arithmetic operators
AOIS	Insert short-cut arithmetic operators
AOIU	Insert basic unary arithmetic operators
AORB	Replace basic binary arith. op. with other one
AORS	Replace short-cut arith. op. with other one
ASRS	Short-Cut Assignment Operator Replacement
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
COR	Conditional Operator Replacement
LOI	Logical Operator Insertion
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement

view of effectiveness between the two test strategies. Only for the first policy (*2_73020419964_2*) a better effectiveness of XACMET test suite is evident due to the particular structure of this policy.

Both the strategies have in general a similar behaviour and they are able to kill less than 20% of the mutated PDPs. By a deeper analysis, we highlighted that these low values of fault detection effectiveness are due to two main reasons: i) the adopted XACML policies are not a complete representation of the XACML policy population but they define the few functions, data types, XACML elements that are currently used in the practice so to focus as much

Table 4 Number of Mutants By Mutations Operator Level and By Java Package.

Java Package	Mutation Operator Level		All
	class-level	method-level	
com.sun.xacml	685	885	1570
com.sun.xacml.attr	431	2943	3374
com.sun.xacml.combine	73	277	350
com.sun.xacml.cond	344	1747	2091
com.sun.xacml.ctx	282	174	456
com.sun.xacml.finder	122	43	165
com.sun.xacml.finder.impl	8	16	24
All	1945	6085	8030

Table 5 XACMET *vs* X-CREATE

Xacml Policies Subjects	Effectiveness		Size	
	X-CREATE	XACMET	X-CREATE	XACMET
2_73020419964_2	2,14	13,57	60	9
create-document-policy	15,30	14,93	16	5
demo-11	8,78	8,89	40	13
demo-26	7,67	8,97	16	8
demo-5	9,18	9,14	84	16
read-document-policy	19,88	19,43	30	6
read-informationunit-policy	8,59	8,91	6	4
read-patient-policy	7,68	8,79	30	6
Xacml-Nottingham-Policy-1	19,65	19,65	16	18

as possible the testing activity on the most critical aspects; ii) the mutation operators are java based operators and do not consider the peculiarities of the XACML language and therefore could not be targeted by the XACML requests used as a tests input.

It is important to remark that in order to correctly compute the effectiveness, the number of distinct killed mutants by each strategy and on each policy has been considered while equivalent mutants have been identified and excluded in the computation of the test suite effectiveness.

Table 5, in the fourth and fifth columns, reports for each of the nine XACML policies, the size of test suites generated by each of the two test strategies. Except for *Xacml-Nottingham-Policy-1* policy, XACMET strategy generates smaller test suites with respect to those generated by Multiple combinatorial. Namely, the size of XACMET test suites is about 70% less than that of Multiple Combinatorial ones, but reaching the same quality in terms of errors found or mutants killed.

5.2 Oracle Validation

We conducted an empirical evaluation of the XACMET oracle. The evaluation includes two phases: in the former we assessed the compliance of the XACMET

oracle with the XACML specification given by the conformance test suite; in the latter we assessed the compliance of the XACMET oracle with respect to a voting PDP mechanism, considering both the policies of conformance test suite and some real policies.

5.2.1 XACMET Oracle vs *Conformance Test Verdict*

In this first phase, we considered the tests of the XACML 2.0 Conformance Tests V0.4 [29]. Each test consists of three elements: an XACML policy, an XACML request, and an XACML response representing the expected access decision associated to that request. We focused on the subset of tests implementing the mandatory functionalities and specifically on the following groups of tests: i) IIA: tests exercising attribute referencing; ii) IIB: tests exercising target matching; iii) IIC: test exercising function evaluation; and finally iv) IID: tests exercising combining algorithms. For each group, we selected a subset of tests specifying only the functionalities implemented in XACMET. In particular, we excluded by IIA group the tests referring to *Indeterminate* values, in IIC group we considered only the most used arithmetic and equality functions (for instance *type-equal*) while for IID group we considered the combining algorithms presented in Section 4.2.2.

In Table 6 (column 1), we show for each group the percentage of considered tests. Moreover, in rows from 4 to 7, the structure of the considered XACML policies belonging to the four groups is described in terms of cardinality of policies, rules, conditions, subjects, resources, actions and functions (total number of functions and distinct functions) respectively. Table 6 (column 9) shows the number of XACML requests for the four groups of tests. To validate the XACMET oracle, we applied the XACMET approach to the policies of the conformance tests. Specifically, for each test case we derived starting from the XACML policy, the associated XAC-Graph and an ordered set of paths as described in Section 4.1. Then, we evaluated the XACML request belonging to the test case, against the obtained set of paths, we identified the first covered path and derived the verdict associated to that path. Finally, we compared this verdict with the decision value specified in the response belonging to the test case. For all the tests of the conformance, we obtained that the XACMET verdict coincides with the expected access decision. This improves the confidence in the effectiveness of the XACMET approach for the XACML functionalities specified in the conformance tests.

5.2.2 XACMET Oracle vs *PDP Voting Mechanism*

The goal of this second phase was to evaluate the XACMET oracle with respect to other competing automated oracles. We referred to the already mentioned black box approach of multiple implementations testing presented in [19]. Hence we derived an automated majority oracle by running a request on three implementations of an XACML based PDP and then using a majority voting to derive the expected XACML response associated to that re-

Table 6 XACMET Oracle *vs* Conformance Test Verdict

XACML Policy	Functionality							# XACML Request	
	# Policy	# Rule	# Cond	# Sub	# Res	# Act	# Funct (distinct)	I Study	II Study
Conformance Test Suite XACML Policies									
IIA (90%)	18	18	12	18	8	16	112 (12)	18	68
IIB (100%)	53	53	6	51	50	98	410 (7)	53	254
IIC (10%)	22	22	22	18	3	1	102 (19)	22	31
IID (17%)	5	13	7	13	-	-	60 (5)	5	19
Real-world XACML Policies									
2_730204199642_2	1	6	5	3	3	0	4	-	9
create-document-policy	1	3	2	1	2	1	3	-	5
demo-5	1	3	2	2	3	2	4	-	16
demo-11	1	3	2	2	3	1	5	-	13
demo-26	1	2	1	1	3	1	4	-	8
read-document-policy	1	4	3	2	4	1	3	-	6
read-informationunit-policy	1	2	1	0	2	1	2	-	4
read-patient-policy	1	4	3	2	4	1	3	-	6
Xacml-Nottingham-Policy-1	1	3	0	24	3	3	2	-	18

quest. Specifically, we used a pool of three XACML PDPs, namely Sun PDP [6], Herasaf PDP [7] and Balana PDP [8].

In this second phase, we used XACMET to derive, starting from a XACML policy, a set of XACML requests and the associated verdict as described in Section 4.3 and Section 4.4 respectively. For each policy, Table 6 (column 10) shows the cardinality of the derived XACML requests. Each policy and the derived set of requests were executed on the PDPs pool, we observed whether the different PDPs produced the same responses and took as oracle value the majority output. For each request we compared the XACMET oracle with the majority output of the PDPs pool. We observed that for all requests the XACMET oracle coincided with the automated majority oracle. This enhances the confidence of the effectiveness of the XACMET oracle considering also the functionalities of real policies. However, as side effect of this experiment we obtained an enhancement of the requests of the conformance tests. As showed in Table 6 (column 10), this enhancement is evident mostly for IIB policies group (201 additional requests). This is due to the more complex structure of the policies of this group as evidenced also by the higher number of elements (mainly actions and functions) of these policies.

5.3 Using XACMET for Path Coverage Measurement

In this section, we report the coverage measures obtained by X-CREATE test suites. The analysis has been performed using the paths derived by XACMET approach, classified according to their final decision. In particular, Table 7 reports for each policy the number of different paths, having as final decision *Permit*, *Deny*, *NotApplicable* (columns 2, 3 and 4) and the total path cardinality (column 5) respectively. Note that, the cardinality of *NotApplicable* is equal

[6] Sun PDP is available at: `http://sunxacml.sourceforge.net`.

[7] Herasaf PDP is available at: `https://bitbucket.org/herasaf/herasaf-xacml-core`.

[8] Balana PDP is available at: `https://github.com/wso2/balana`.

to one only when a default rule is explicitly defined, none of the other policies rule is satisfied and the target of the policy contains just one constraint.

By construction, the test suites derived by XACMET approach, according to the algorithm presented in Section 4.3, cover all the paths of Table 7 with a number of requests equal to the cardinality of the paths. Therefore, in this experiment we do not report the coverage measures obtained by the XACMET test suites since they always reach the 100% of path coverage for each decision type.

Table 8 reports the distribution of the X-CREATE test requests on the different paths of Table 7. For instance, considering *2_73020419964_2* policy (second row), all the 60 generated requests cover the single *NotApplicable* path since they are not able to satisfy the policy target. For this policy, the paths having *Permit* and *Deny* decisions are never covered. In this case, even if X-CREATE is able to generate an high number of requests (60), their variety is not enough to guarantee a satisfactory coverage of the policy paths. On the contrary, XACMET approach with only 9 requests, of which 6, 2, 1 requests covering *Permit*, *Deny* and *NotApplicable* decisions respectively, is able to cover all the paths decisions with a 100% of coverage. The percentage of path coverage for each policy is reported in Table 9. In particular, for *2_73020419964_2* policy, the coverage percentage is: 0% (*Permit*), 0% (*Deny*) and 100% (*NotApplicable*) respectively for a total amount of 11%.

In case of *demo-26*, the X-CREATE requets are able to cover only some of the paths with *Permit* and *NotApplicable* decisions. In particular, X-CREATE generates 4 requests able to cover 4 of the 5 different paths with *Permit* decision and 12 requests able to cover the unique path with *NotApplicable* decisions. As reported in Table 9 (line 6), the coverage percentage is: 80% (*Permit*), 0% (*Deny*) and 100% (*NotApplicable*) respectively for a total amount of 63%. On the contrary, XACMET approach with only 8 requests (5, 2, 1 requests covering *Permit*, *Deny* and *NotApplicable* decisions respectively) is able to cover all the paths decisions with a 100% of coverage.

Only in case of *demo-5* and *demo-11*, the requests are distributed over all the paths decisions. However, for *demo-5* X-CREATE generates 17 requests able to cover 7 of the 10 different paths with *Permit* decision, 4 requests covering 4 of the 5 paths having *Deny* decision and 63 requests able to cover the unique path with *NotApplicable* decision. As reported in Table 9 (line 4), the coverage percentages are: 70% (*Permit*), 80% (*Deny*) and 100% (*NotApplicable*) respectively, for a total amount of 75%.

For *demo-11* X-CREATE generates 8 requests able to cover all the 5 paths with Permit decision, 2 requests covering the 7 paths having Deny decision and 30 requests able to cover the unique path with NotApplicable decision. As reported in Table 9 (line 5) the coverage percentages are: 100% (*Permit*), 29% (*Deny*) and 100% (*NotApplicable*) respectively, for a total amount of 62%.

The analysis of the results reported before confirms that combinatorial approaches for test cases generation never reach the 100% of total coverage. This is because they are not able to satisfy the very specific policy conditions; in-

deed they leverage on policy values combinations and do not take into account the policy structure.

As a side effect, the coverage measures provided by XACMET approach as well as the mapping of the requests over the different decision types could be exploited for a twofold purpose: i) either for improving the coverage of a given test suite by adding ad hoc requests; ii) or to reduce the redundancy inside a test suite to keep the same coverage percentage with a less number of test cases.

Table 7 XACMET Decisions per Type.

XACML Policy	Decisions			
	# Permit	# Deny	# NotApplicable	# Tot
2_73020419964_2	6	2	1	9
create-document-policy	2	1	2	5
demo-5	10	5	1	16
demo-11	5	7	1	13
demo-26	5	2	1	8
read-document-policy	3	1	2	6
read-informationunit-policy	1	1	2	4
read-patient-policy	3	1	2	6
Xacml-Nottingham-Policy-1	6	5	7	18

Table 8 X-CREATE Decisions per Type.

XACML Policy	Decisions			
	# Permit	# Deny	# NotApplicable	# Tot
2_73020419964_2	0	0	60	60
create-document-policy	1	0	11	12
demo-5	17	4	63	84
demo-11	8	2	30	40
demo-26	4	0	12	16
read-document-policy	0	0	30	30
read-informationunit-policy	0	0	6	6
read-patient-policy	0	0	30	30
Xacml-Nottingham-Policy-1	0	3	13	16

6 Related Work

The work presented in this paper spans over the following research directions:

Analysis and modeling of policy specification Available proposals include different verification techniques [38], such as model-checking [39] or SAT solvers [34].

Table 9 XAC-Paths Coverage.

XACML Policy	Decisions			
	% Permit	% Deny	% NotApplicable	% Tot.
2_73020419964_2	0 %	0 %	100 %	11 %
create-document-policy	50 %	0 %	100 %	60 %
demo-5	70 %	80 %	100 %	75 %
demo-11	100 %	29 %	100 %	62 %
demo-26	80 %	0 %	100 %	63 %
read-document-policy	0 %	0 %	100 %	33 %
read-informationunit-policy	0 %	0 %	100 %	50 %
read-patient-policy	0 %	0 %	100 %	33 %
Xacml-Nottingham-Policy-1	0 %	60 %	100 %	56 %

Well-known analysis and verification tools for access control policies are: i) Margrave [12], which represents policies as Multi-Terminal Binary Decision Diagrams (MTBDDs) and can answer queries about policy properties; and ii) ACPT (Access Control Policy Testing) tool [15] that transform policies into finite state machines and represent static, dynamic and historic constraints into Computational Tree Logic. The capabilities and performances of such tools are analytically evaluated in [17].

Differently from the above approaches, XACMET models the expected behaviour of the evaluation of a given XACML policy as a labeled graph and guarantees the full path coverage of such graph. Moreover, our proposed XAC-Graph model is richer since it also represents the rule combining algorithm, the functions, and the associated conditions. The authors of [31] provide an optimized approach for XACML policies modeling based on tree structures aimed at fast searching and evaluation of applicable rules. Differently from our proposal, the main focus of this work is on performance optimization more than on oracle derivation.

Test cases and oracle derivation Considering the automated test cases generation, solutions have been proposed for testing either the XACML policy or the PDP implementation [4,5]. Among them, the most referred ones such as X-CREATE and the Targen tools [23,5] use combinatorial approaches for test cases generation. Specifically, the X-CREATE tool and the Targen tool [7,5]. generate test inputs using combinatorial approaches of the XACML policies values and the truth values of independent clauses of policy values, respectively. However, combinatorial approaches are shallow with respect to policy semantics.

Model-based testing has already been widely investigated for policy testing, e.g. [32,35]. Such approaches provide methodologies or tools for automatically generating access control test models from functional models and access control rules. A different approach is provided by Cirg [18,13] that is able to exploit change-impact analysis for test cases generation starting from policy specification. An overview of verification and testing methods for access control

policies and models is provided in [13]. The key original aspect of our approach is in the XAC-Graph model which we derive, which is richer in expressiveness than other proposed models, and can provide directly the evaluation paths including a verdict associated to a request. Other approaches leverage existing symbolic execution techniques for generating test cases. Specifically, in [20], first the access control policy under test is converted into semantically equivalent C Code Representation (CCR). Then, the CCR is symbolically executed to generate test inputs.

About the automated oracle, notwithstanding the huge interest devoted to this topic, reducing the human activity in the evaluation of the testing results is still an issue [2]. The automated oracle derivation is a key aspect in the context of XACML systems and testers need usually to manually verify the XACML responses. The few available solutions mainly deal with model based approaches. Specifically, the authors of [10] provide an integrated toolchain including test case generation as well as policy and oracle specification for the PDP testing. Other proposals such as [8] address the use of monitoring facilities for the assessment of the run-time execution of XACML policies. Differently from the above approaches, the main benefits of XACMET deal with the derivation of an XACML verdict for each XACML request.

Coverage assessment Coverage assessment is an important feature to focus the testing activity on the generation or selection of the test cases that cover the most important elements and/or policy constructs [33]. Considering coverage assessment of a test suite with respect to an XACML policy, in literature there are few works facing this problem. Seminal works as [27] and [6] present some coverage criteria for XACML policy. Specifically, in the former, the authors define three structural coverage metrics targeting XACML policies, rules and conditions respectively and use them for reducing test sets and measure the effects of test reduction in terms of fault detection. In the latter, the authors also address the policy set and do not require the policy execution and PDP instrumentation. More recently, in [40] the authors extend the introduced concept by presenting a combinatorial testing approach based on data flow coverage, in [37] the test execution information is used to determine which policy element is faulty while in [36] a family of coverage criteria for XACML policies is formalised and used for test cases generation. The authors of [21] propose an access control policy infrastructure, based on an external monitoring facility, for enabling the coverage measurement of XACML policies and evaluation of different testing strategies. Differently from existing approaches, in this paper, we leverage the XAC-Graph definition to derive a set of paths and measure the coverage of test requests in terms of the paths executed on the XAC-Graph.

7 Conclusion

We have introduced a novel model-based approach to automatic generation of XACML requests as well as automatic oracle derivation for testing pol-

icy evaluation engines. We have defined the XAC-Graph, which is a labelled typed graph representing how a policy should be evaluated taking into account the defined rule combining algorithm and the semantics of the functions. The XACMET approach fully automatically derives a set of XACML requests as well as a verdict for each executed XACML request by considering the set of evaluation paths derived from the obtained graph. We have illustrated the approach on an example policy and provided experimental results evidencing: i) the effectiveness of our test generation strategy with respect to existing combinatorial test generation approaches; ii) the effectiveness of our automated oracle proposal with respect to the oracle provided in the XACML conformance tests and an automated oracle implemented as a voting mechanism; iii) the ability of XACMET to perform path coverage measurement of a given test suite.

We recognize that the presented results are limited to a small set of policies. Therefore even though they are real world policies which have been obtained from different sources, the observed results may not have external validity and further experimentation would be required. We measured the fault detection effectiveness of XCREATE and XACMET by referring the mutation operators of [22]. This may threaten internal validity, that could be mitigated by additional mutation operators able to exploit the numerous other functions that may be specified in the policy. Except for possible faults in our own tool implementation, for the rest all steps have been carried out in automated way, so we do not see other potential internal threats.

In the future, we plan to extend our test generation strategy and automated oracle in order to consider more functionalities of the XACML language, such as different XACML functions and the *PolicySet* element. We also plan to investigate the adoption of the coverage information obtained by XACMET for prioritization or minimization of a given test suite. The XACMET approach will also be extended to be compliant with the last XACML 3.0 standard version. Future work will also include further experimentation of XACMET, and its comparison with other model-based approaches.

References

1. Apt, K.: Principles of constraint programming. Cambridge university press (2003)
2. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE transactions on software engineering **41**(5), 507–525 (2015)
3. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E.: An automated model-based test oracle for access control systems. In: Proceedings of the 13th International Workshop on Automation of Software Test, AST '18, pp. 2–8. ACM, New York, NY, USA (2018). DOI 10.1145/3194733.3194743. URL `http://doi.acm.org/10.1145/3194733.3194743`
4. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., Mori, P.: Testing of polpa-based usage control systems. Software Quality Journal **22**(2), 241–271 (2014)
5. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Schilders, L.: Automated testing of extensible access control markup language-based access control systems. IET Software **7**(4), 203–212 (2013)

6. Bertolino, A., Le Traon, Y., Lonetti, F., Marchetti, E., Mouelhi, T.: Coverage-based test cases selection for xacml policies. In: Proceedings of ICST Workshops, pp. 12–21 (2014)
7. Bertolino, A., Lonetti, F., Marchetti, E.: Systematic XACML Request Generation for Testing Purposes. In: Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 3 –11 (2010)
8. Calabrò, A., Lonetti, F., Marchetti, E.: Access control policy coverage assessment through monitoring. In: Proc. of TELERISE, pp. 373–383 (2017)
9. Daoudagh, S.: A Data Warehouse and a Framework for the Validation and Testing of Access Control Systems. Master's thesis, Department of Computer Science, University of Pisa, Italy (2017)
10. Daoudagh, S., El Kateb, D., Lonetti, F., Marchetti, E., Mouelhi, T.: A toolchain for model-based design and testing of access control systems. In: Proc.of MODELSWARD, pp. 411–418. IEEE (2015)
11. Daoudagh, S., Lonetti, F., Marchetti, E.: Assessment of access control systems using mutation testing. In: Proceedings of the First International Workshop on TEchnical and LEgal aspects of data pRIvacy, pp. 8–13. IEEE Press (2015)
12. Fisler, K., Krishnamurthi, S., Meyerovich, L., Tschantz, M.: Verification and change-impact analysis of access-control policies. In: Proc. of ICSE, pp. 196–205 (2005)
13. Hu, V.C., Kuhn, R., Yaga, D.: Verification and test methods for access control policies/models. NIST Special Publication **800**, 192 (2017)
14. Hwang, J., Martin, E., Xie, T., Hu, V.C.: Policy-based testing. In: Encyclopedia of Software Engineering, pp. 673–683. Taylor & Francis (2011)
15. Hwang, J., Xie, T., Hu, V., Altunay, M.: Acpt: A tool for modeling and verifying access control policies. In: Proc. of International Symposium on Policies for Distributed Systems and Networks (POLICY), pp. 40–43 (2010)
16. Kuchcinski, K., Szymanek, R.: Jacop-java constraint programming solver. In: CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming (2013)
17. Li, A., Li, Q., Hu, V.C., Di, J.: Evaluating the capability and performance of access control policy verification tools. In: Proc. of MILCOM, pp. 366–371 (2015)
18. Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability **23**(8), 613–646 (2013)
19. Li, N., Hwang, J., Xie, T.: Multiple-implementation testing for XACML implementations. In: Proc. of TAV-WEB, pp. 27–33 (2008)
20. Li, Y., Li, Y., Wang, L., Chen, G.: Automatic xacml requests generation for testing access control policies. In: SEKE, pp. 217–222 (2014)
21. Lonetti, F., Marchetti, E.: On-line tracing of xacml-based policy coverage criteria. IET Software (2018)
22. Ma, Y.S., Offutt, J., Kwon, Y.R.: Mujava: a mutation system for java. In: Proceedings of the 28th international conference on Software engineering, pp. 827–830. ACM (2006)
23. Martin, E., Xie, T.: Automated test generation for access control policies. In: Supplemental Proc. of ISSRE (2006)
24. Martin, E., Xie, T.: Automated test generation for access control policies via change-impact analysis. In: Proc. of Third International Workshop on Software Engineering for Secure Systems (SESS), pp. 5–12 (2007)
25. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proc. of WWW, pp. 667–676 (2007)
26. Martin, E., Xie, T., Yu, T.: Defining and measuring policy coverage in testing access control policies. In: Proc. of ICICS, pp. 139–158 (2006)
27. Martin, E., Xie, T., Yu, T.: Defining and measuring policy coverage in testing access control policies. In: International Conference on Information and Communications Security, pp. 139–158. Springer (2006)
28. OASIS: eXtensible Access Control Markup Language (XACML) Version 2.0. `http://docs.oasis-open.org/xacml/2.0/access\_control-xacml-2.0-core-spec-os.pdf` (2005)
29. OASIS: XACML 2.0 Conformance Tests v0.4. `https://www.oasis-open.org/committees/document.php?document_id=14846` (2005)

30. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. Advances in Computers (2017)
31. Pina Ros, S., Lischka, M., Gómez Mármol, F.: Graph-based xacml evaluation. In: Proc. of the 17th ACM symposium on Access Control Models and Technologies, pp. 83–92 (2012)
32. Pretschner, A., Mouelhi, T., Le Traon, Y.: Model-based tests for access control policies. In: Proc. of ICST, pp. 338–347 (2008)
33. Shahid, M., Ibrahim, S., Mahrin, M.N.: A study on test coverage in software testing. Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia (2011)
34. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Analysis of xacml policies with smt. In: Proc. of International Conference on Principles of Security and Trust, pp. 115–134. Springer (2015)
35. Xu, D., Kent, M., Thomas, L., Mouelhi, T., Le Traon, Y.: Automated model-based testing of role-based access control using predicate/transition nets. IEEE Transactions on Computers **64**(9), 2490–2505 (2015)
36. Xu, D., Shrestha, R., Shen, N.: Automated coverage-based testing of xacml policies. In: Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, pp. 3–14. ACM (2018)
37. Xu, D., Wang, Z., Peng, S., Shen, N.: Automated fault localization of xacml policies. In: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, SACMAT '16, pp. 137–147. ACM, New York, NY, USA (2016). DOI 10.1145/2914642.2914653. URL `http://doi.acm.org/10.1145/2914642.2914653`
38. Xu, D., Zhang, Y.: Specification and analysis of attribute-based access control policies: An overview. In: Proc. of Eighth International Conference on Software Security and Reliability-Companion (SERE-C), pp. 41–49. IEEE (2014)
39. Zhang, N., Ryan, M., Guelev, D.: Evaluating access control policies through model checking. In: Information Security, *Lecture Notes in Computer Science*, vol. 3650, pp. 446–460 (2005)
40. Zhang, Y., Zhang, B.: A new testing method for xacml 3.0 policy based on abac and data flow. In: 2017 13th IEEE International Conference on Control Automation (ICCA), pp. 160–164 (2017). DOI 10.1109/ICCA.2017.8003052