Systematic XACML request generation for testing purposes

Antonia Bertolino, Francesca Lonetti, Eda Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
via G. Moruzzi, 1 - 56124 Pisa, Italy
{antonia.bertolino, francesca.lonetti, eda.marchetti}@isti.cnr.it

***Abstract*—A widely adopted security mechanism is the specification of access control policies by means of the XACML language. In this paper, we propose a framework, called X-CREATE, for the systematic generation of test inputs (XACML requests). Differently from existing tools, X-CREATE exploits the XACML Context Schema. In particular, the tool applies a XML-based methodology (XPT) to systematically produce a set of intermediate instances, covering the XACML Context Schema. Moreover, for request generation, X-CREATE applies a procedure for parsing the policy under test and assigning values to the generated intermediate instances. The aim of the proposed framework is twofold: testing of policy evaluation engines and testing of access control policies. The experimental results show that the fault detection effectiveness of X-CREATE is similar or higher than that of existing approaches.**

***Keywords*-XACML; Test suite generation; Policy testing.**

I. Introduction

An important aspect in the security of modern information management systems is the control of accesses. Data and resources must be protected against unauthorized, malicious or improper usage or modification. For this purpose, a widely adopted security mechanism is the specification of access control policies by means of policy languages such as the eXtensible Access Control Markup Language (XACML) [1]. The policies rule various aspects such as: the level of confidentiality of data, the procedures for managing data and resources, the classification of resources and data into category sets with different access controls. An important software component of the access control systems is the Policy Decision Point (PDP) that evaluates the requests against the access control policies.

Due to the huge amount of information and resources to be managed and ruled in policies, their definition, implementation, modification and maintenance are very critical activities for policy developers. The risks grow in the case of complex, distributed and large systems, where multiple policies have to be managed. Hence, to prevent security problems a rigorous and accurate verification and testing process must be adopted.

Policies, or more general metamodels of security policies, provide a model that specifies the access scheme for the various actors accessing the resources of the system. This model has been successfully exploited for model-based testing (e.g., [2]). However, for the purpose of test case generation a second important model composes the access control mechanism: the standard format representing all the possible compliant requests. The potential of this second model has not been yet fully exploited.

In a XACML access control system every incoming request must be conforming to a specific XML Schema called the Context Schema [1] [1]. Once for all and independently from any specified policy this schema describes the overall structure of the accepted input requests for the PDP. Thus this XML Schema is a model formally describing what constitutes an agreed valid input. The XACML instances, formatted according to the rules of the referred Context Schema, represent the conforming requests, i.e. allowed naming and structure of data for access requests.

To the best of our knowledge, there are not access control policy testing methodologies that exploit this interesting and important model of data input. Thus our proposal here is to combine the potential of the Context Schema in describing input data of the requests in open and standard form, with a method for the systematic generation of policy test suites.

Our proposed method, called X-CREATE (XaCml REquests derivAtion for TEsting), is not in contrast or alternative with the existing policy testing approaches, rather it suggests a further source of test case generation, so far unexplored.

In particular our contribution includes:

- A test strategy for generating compliant XACML requests from the XACML Context Schema. We cover all interesting combinations of the schema by adopting a systematic black-box criterion [3].
- A universally valid generic test suite, derived by our proposed test strategy, which is customizable to any specific policy. The test suite provides a skeleton of XML instances for every possible request structure derivable by the schema of the requests.
- An algorithm for customizing the test suite skeletons to the peculiarities of a specific policy. As a result a ready-

[1]For readability purposes in this paper we refer to the XACML 1.0 Specification Set and not to the XACML 2.0 Specification Set. However, we identically applied the proposed methodologies and techniques to the 2.0 version.

to-use set of requests and responses is systematically constructed.
- A preliminary empirical validation on a case study comparing X-CREATE against an existing competitor tool.

The paper is structured as follows: in the next section, we provide a basic description of the XACML language. In Section III and IV related work and an overview of our approach are presented respectively. Then we describe the X-CREATE framework in Section V. A discussion about the effectiveness of the proposed approach is in Section VI and general conclusions are drawn in Section VII.

II. XACML Specification

XACML [1] is a platform-independent XML based standard language designed by the Organization for the Advancement of Structured Information Standards (OASIS).

In this section we provide an overview of the XACML features for the access control decisions. At the root of all XACML policies is a *Policy* or a *PolicySet*. A *PolicySet* can contain other *Policies* or *PolicySets*.

A *Policy* consists of a *Target*, a set of *Rules* and a *Rule combining algorithm*. The *Target* specifies the *Subjects*, the *Resources* and the *Actions* on which a policy can be applied. If a request satisfies the target of the policy, then the set of rules of the policy is checked, otherwise the policy is skipped without examining its rules. A *Rule* is the basic element of a policy. It is composed by a *Target*, that is similar to the policy target and specifies the constraints of the requests to which the rule is applicable. The heart of most rules is a *Condition* that is a boolean function evaluated when the rule is applicable to a request. The result of the condition evaluation is the rule effect (*Permit* or *Deny*) if the condition is evaluated to be true, *NotApplicable* otherwise. If an error occurs during the application of a policy to the request, *Indeterminate* is returned as decision. More than one rule in a policy may be applicable to a given request. The *rule combining algorithm* specifies the approach to be adopted to compute the decision result of a policy containing rules with conflicting effects. The access decision is given by considering all attribute values describing the subjects, the resource, and the action of an access request and comparing them with the attribute values of a policy.

We show in Listing 1 an example of a simplified XACML policy ruling library access. Its target (lines 5-9) says that this policy applies to any subject, resource and action. This policy has a first rule (lines 10-54) with a target (lines 12-36) specifying that this rule applies only to the access requests of a “write” action of the “http://library.com/record” resource. The rule condition will deny access when the requester subject role is not “researcher”, “professor”, or “staff”. The effect of the second rule (lines 55-92) is “Permit” when the subject is “Julius”, the action is “read”, and the resource is “http://library.com/record/journals”. The rule combining algorithm of the policy (line 4) determines to return the result of the evaluation of the first applicable rule.

The XACML Technical Committee provides a Policy Schema for the validation of XACML policies and a Context Schema for the validation of XACML requests and responses. Figure 1 sketches the structure of the schema and the occurrences of elements only for the part concerning the requests derivation.

```
1  <Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
2   PolicyId="PolicyExample"
3   RuleCombiningAlgId="rule-combining-algorithm:first-
       applicable">
4   <Target>
5    <Subjects><AnySubject/></Subjects>
6    <Resources><AnyResource/></Resources>
7    <Actions><AnyAction/></Actions>
8   </Target>
9   <Rule
10   RuleId="PolicyExample:rule1" Effect="Deny">
11   <Target>
12    <Subjects><AnySubject/></Subjects>
13    <Resources>
14     <Resource>
15      <ResourceMatch
16       MatchId="anyURI-equal">
17       <AttributeValue
18        DataType="anyURI">http://library.com/record</
            AttributeValue>
19       <ResourceAttributeDesignator
20        AttributeId="resource-id"  DataType="anyURI"/>
21      </ResourceMatch>
22     </Resource>
23    </Resources>
24    <Actions>
25     <Action>
26      <ActionMatch
27       MatchId="string-equal">
28       <AttributeValue
29        DataType="string">write</AttributeValue>
30       <ActionAttributeDesignator
31        AttributeId="action-id" DataType="string"/>
32      </ActionMatch>
33     </Action>
34    </Actions>
35   </Target>
36   <Condition
37    FunctionId="function:not">
38    <Apply
39     FunctionId="string-at-least-one-member-of">
40     <SubjectAttributeDesignator
41      AttributeId="role" MustBePresent="false" DataType="
          string"/>
42     <Apply
43      FunctionId="string-bag">
44      <AttributeValue
45       DataType="string">researcher</AttributeValue>
46      <AttributeValue
47       DataType="string">professor</AttributeValue>
48      <AttributeValue
49       DataType="string">staff</AttributeValue>
50     </Apply>
51    </Apply>
52   </Condition>
53  </Rule>
54  <Rule
55   RuleId="PolicyExample:rule2" Effect="Permit">
56   <Target>
57    <Subjects>
58     <Subject>
59      <SubjectMatch
60       MatchId="string-equal">
61       <AttributeValue
62        DataType="string">Julius</AttributeValue>
63       <SubjectAttributeDesignator
64        AttributeId="subject-id" DataType="string"/>
65      </SubjectMatch>
```

```
66       </Subject>
67      </Subjects>
68      <Resources>
69       <Resource>
70        <ResourceMatch
71         MatchId="anyURI-equal">
72         <AttributeValue
73          DataType="anyURI">http://library.com/record/
              journals</AttributeValue>
74         <ResourceAttributeDesignator
75          AttributeId="resource-id" DataType="anyURI"/>
76        </ResourceMatch>
77       </Resource>
78      </Resources>
79      <Actions>
80       <Action>
81        <ActionMatch
82         MatchId="string-equal">
83         <AttributeValue
84          DataType="string">read</AttributeValue>
85         <ActionAttributeDesignator
86          AttributeId="action-id" DataType="string"/>
87        </ActionMatch>
88       </Action>
89      </Actions>
90     </Target>
91    </Rule>
92   </Policy>
```

Listing 1. An example of XACML policy

III. Related Work

The natural, but clearly not cost-effective, approach for testing access control policies is to manually derive a set of test inputs that correspond to requests. However the complexity of the policy prevents the manual specification of a set of test cases capable of covering all the possible interesting critical situations or faults. This implies the need of automated test cases generation. Current approaches can be categorized into:

Synthesis-based. In this case, available properties and specifications can be used to automatically synthesize input values useful for checking the policy itself. A proposal of generic approaches for test synthesis is for instance [4] while Cirg [5] is one of the few specific for XACML.

Model-based. Usually these approaches are based on either the representation of policy implied behavior by means of models [6], [7] or the role based access control models [8], [2]. These approaches provide methodologies and tools for automatically generate abstract test cases so to verify functional aspects and vulnerabilities and discover policy specification or implementation faults.

Conformance. The goal is to verify the mapping between the policy reference model and its specification [9], [10]. In particular, only with reference to XACML 1.0 and 1.1, the XACML Technical Committee has provided a Conformance Tests suite [1][2] for verifying the PDP correct behavior.

However, all the available proposals are based on the policy. To the best of our knowledge, none of the available methodologies analyzes the domain of conforming requests. In this paper, we are the first to take this orthogonal direction: focus the testing activity not on the policy point-of-view but also on the perspective of the correct domain of the requests.

[2]Also a DRAFT XACML 2.0 Conformance Tests suite is available.

IV. Approach Overview

Our approach focuses on the XACML Context Schema (see Figure 1). The solution we propose is to derive a test suite of conforming requests from this standard schema. The peculiarity of our proposal is that since there exists one unique XACML Context Schema, that is independent from the policy specification, a generic conformance test suite of requests can be derived once and for all, as explained in detail in section V. Each request in this generic test suite is a general structure of a valid XACML request instance. These structurally different requests (or an opportunely chosen subset thereof) can be instantiated from time to time according to the policy specification considered, by suitably varying the input values and in practice originating an arbitrary number of final request instances. These requests can be used for two different testing purposes:

Testing the policy implementation. In this scenario we assume that the system under test (SUT) is the PDP implementing a policy specification that is considered correct. The actors of this scenario are: a Test Generator which takes as input the XACML Context Schema and the policy specification for generating a set of XACML requests, the PDP, and an oracle (e.g. as that in [11]) that is able to recognize if the PDP response is correct and provide accordingly a verdict (pass/fail).

Testing policy specification. In this scenario we consider a situation in which a policy developer wants to check the correctness of a policy specification. The actors of this scenario include again a Test Generator, a PDP that provides the responses and the policy developer who provides the policy specification and the oracle verdicts.

V. Framework

In this section, we outline the framework of X-CREATE for generating a set of XACML requests starting from the XACML Context Schema. The framework consists of three main components: A. intermediate-request generation; B. policy-under-test analysis; C. request values assignment.

A. Intermediate-request generation

Given the XACML Context Schema, we apply the XML-based Partition Testing (XPT) approach proposed in [12]. This approach generates conforming XML instances from a XML Schema by applying a variant of the well-known Category Partition (CP) method [3] and traditional boundary condition. In particular, the occurrences declared for each element in the schema are analyzed and, applying a boundary condition strategy, the border values (`minOccurs` and `maxOccurs`) to be considered for the instances generation are derived.

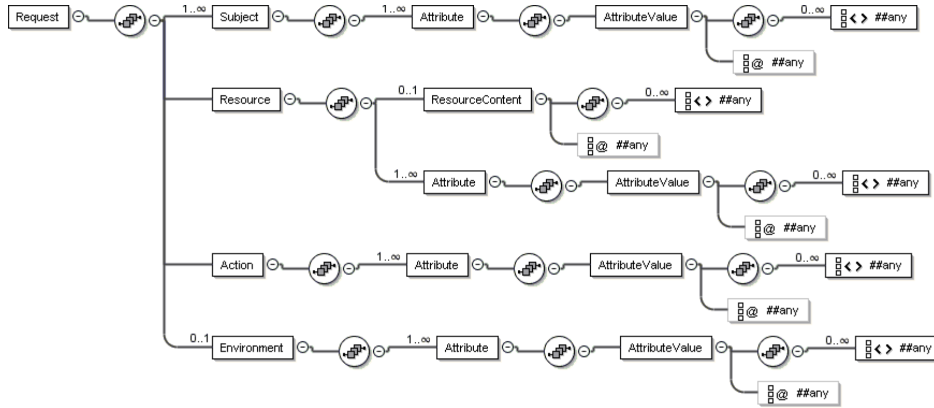

Figure 1. Sketch of XACML 1.0 Context Schema for requests derivation

Combining the occurrence values assigned to each element, XPT derives a set of intermediate instances. The final instances are derived from these intermediate ones by assigning values to the various elements. This methodology has been already implemented into a tool called TAXI [12][13].

Given the XACML Context Schema, by the tool TAXI we can derive $3^Y * 2^Z$ intermediate instances, where Y is the number of schema elements with unbounded cardinality, and Z is the number of elements having [0,1] cardinality. In particular, in the XACML Context Schema, only for the part concerning the requests specification (see Figure 1), there are 10 elements with unbounded occurrence and 2 having [0,1] cardinality. The other elements have cardinality 1. Thus, the application of TAXI to the XACML Context Schema, (only for the part concerning the requests specification) will generate a maximum number of $3^{10} * 2^2 = 236196$ structurally different intermediate requests. These (or an opportunely selected subset) will be then filled with the appropriate input values so to generate the final requests. Giving random values to these intermediate requests, an infinite number of final requests can be generated.

Note that, the user can choose the generation, by the TAXI tool interface, of a number of intermediate requests lower than 236196. Once the number of intermediate requests to be generated is fixed, TAXI will automatically select a corresponding number of intermediate instances, by applying the well-known pair-wise approach [14], that has been shown to be effective in picking a test subset with good fault detection capability.

Listings 2 and 3 are samples of two intermediate generated requests. In particular, the former is obtained by setting the resource *attribute* element occurrence to 2 and all the other occurrences to the `minOccur` value, the latter is obtained by setting all occurrences to the `minOccur` value. Note that in both examples the values for elements and attributes are not yet specified.

```
1  <Request xmlns="urn:oasis:names:tc:xacml:1.0:context">
2   <Subject>
3    <Attribute AttributeId="" DataType="">
4     <AttributeValue/>
5    </Attribute>
6   </Subject>
7   <Resource>
8    <Attribute AttributeId="" DataType="">
9     <AttributeValue/>
10   </Attribute>
11   <Attribute AttributeId="" DataType="">
12    <AttributeValue/>
13   </Attribute>
14  </Resource>
15 <Action>
16   <Attribute AttributeId="" DataType="">
17    <AttributeValue/>
18   </Attribute>
19  </Action>
20 </Request>
```

Listing 2. Intermediate request 1

```
1  <Request
2   xmlns="urn:oasis:names:tc:xacml:1.0:context">
3   <Subject>
4    <Attribute AttributeId="" DataType="">
5     <AttributeValue/>
6    </Attribute>
7   </Subject>
8   <Resource>
9    <Attribute AttributeId="" DataType="">
10    <AttributeValue/>
11   </Attribute>
12  </Resource>
13  <Action>
14   <Attribute AttributeId="" DataType="">
15    <AttributeValue/>
16   </Attribute>
17  </Action>
18 </Request>
```

Listing 3. Intermediate request 2

B. Policy-under-test analysis

The intermediate requests (derived as explained in Section V-A) must be then filled with the values taken from the policy under test for elements and attributes, so to obtain executable and meaningful final requests. A simply random value assignment would not be sufficient for exercising all the policy functionalities, thus the necessity of a component for the policy analysis.

Specifically the steps performed by this component are:

- Define four values sets: SubjectSet, ResourceSet, ActionSet and EnvironmentSet respectively.
- Look for the matching attributes and elements that appear in the *target* element of each *rule*, *policy* and *policySet* element included in the policy under test. They are defined in the *SubjectMatch*, *ResourceMatch*, and *ActionMatch* sections of the *Subjects*, *Resources*, and *Actions* respectively.
- Take the values of the *AttributeValue* elements and the values of the attributes named *AttributeId*, *Datatype*, *Issuer* and *SubjectCategory* of the SubjectMatch, *ResourceMatch*, and *ActionMatch* and put them in the SubjectSet, ResourceSet, ActionSet respectively.
- For each policy rule, take all values of the *AttributeValue* elements and all values of attributes named *AttributeId*, *Datatype* and *Issuer* (optional) of the *condition* section and put them in the SubjectSet, ResourceSet, ActionSet, and EnvironmentSet, if they refer to the *SubjectAttributeDesignator*, *ResourceAttributeDesignator*, *ActionAttributeDesignator* or *EnvironmentAttributeDesignator* elements respectively.

In Table I we show the result of the application of the policy-under-test analysis component to the policy of Listing 1. Note that in this case EnvironmentSet is empty and no values are associated to the *Issuer* and *SubjectCategory* attributes.

For robustness and negative testing purposes the policy analyzer component adds to each set of data (SubjectSet, ResourceSet, ActionSet, and EnvironmentSet) random values for elements and attributes.

C. Request values assignment

This component has the role of filling the intermediate requests with the values of the SubjectSet, ResourceSet, ActionSet, and EnvironmentSet sets. Note that, the number of requests to be filled can be established by the user (see Section V-A).

In particular we define a *subject entity* 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* as a combination of the values of the elements and attributes of the ResourceSet, ActionSet, and EnvironmentSet respectively. According to these definitions the values assignment is performed with the following steps:

- Derive the set of *subject entities*, *resource entities*, *action entities* and *environment entities*.
- Generate the *ValuesSet* set with the combinations of a *subject entity*, a *resource entity*, an *action entity* and an *environment entity* following a combinatorial approach. In particular, in the *ValuesSet* set the values are inserted applying first pair-wise combination [14], [2]. Then, if there are more intermediate requests to be filled, a three-wise combination is applied. Finally, if there are other intermediate requests not yet filled, applying the 4-wise, generate all the possible combinations of *subject entity*, *resource entity*, *action entity* and *environment entity* and insert them in the *ValuesSet* set.
- Take the values from the *ValuesSet* set one by one and use them for filling the intermediate requests as described in the following:
  - Note that each value of the *ValuesSet* set includes a *subject entity*, *resource entity*, *action entity* and *environment entity*. Use the values of this *subject entity*, *resource entity*, *action entity* and *environment entity* to fill the values of elements and attributes of the *subject*, *resource*, *action* and *environment* of an intermediate instance respectively.
  - If the *ValuesSet* set has been entirely used for filling the intermediate requests, then take again the *ValuesSet* values in the same order for filling the remaining intermediate requests.
  - If there are further elements and attributes in the *subjects*, *resource*, *action* and *environment* of an intermediate request, not yet filled with the values of the *subject entity*, *resource entity*, *action entity* and *environment entity* taken from the *ValuesSet* set, then fill them randomly picking values of elements and attributes from the SubjectSet, ResourceSet, ActionSet and EnvironmentSet respectively, excluding the *subject entities*, *resource entities*, *action entities* and *environment entities* already assigned.

In Listings 4 and 5 we show the final requests obtained by filling the intermediate requests of Listings 2 and 3 respectively with the *subject entities*, *resource entities* and *action entities* obtained from the values of the SubjectSet, ResourceSet, ActionSet and EnvironmentSet of Table I considering also the random values. Note that the value (datastream:id) of the *AttributeId* attribute of one *Attribute* element of the *Resource* section of the Listing 4 is a random value included in the ResourceSet by the policy analyzer

Table I
ANALYSIS OF THE POLICY PRESENTED IN LISTING 1

AttributeId	DataType	AttributeValue
	SubjectSet	
role	string	professor
		researcher
		staff
subject-id	string	Julius
	ResourceSet	
resource-id	anyURI	http://library.com/record
resource-id	anyURI	http://library.com/record/journals
	ActionSet	
action-id	string	read
action-id	string	write

component described in Section V-B.

```
<Request
 xmlns="urn:oasis:names:tc:xacml:1.0:context">
 <Subject>
  <Attribute
   AttributeId="subject-id"
   DataType="string">
   <AttributeValue>Julius</AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
  <Attribute
   AttributeId="resource-id"
   DataType="anyURI">
   <AttributeValue>http://library.com/record/journals</
        AttributeValue>
  </Attribute>
  <Attribute
   AttributeId="datastream:id"
   DataType="anyURI">
   <AttributeValue>http://library.com/record</
        AttributeValue>
  </Attribute>
 </Resource>
 <Action>
  <Attribute
   AttributeId="action-id"
   DataType="string">
   <AttributeValue>read</AttributeValue>
  </Attribute>
 </Action>
</Request>
```

Listing 4. Final request 1

```
<Request
 xmlns="urn:oasis:names:tc:xacml:1.0:context">
 <Subject>
  <Attribute
   AttributeId="subject-id"
   DataType="string">
   <AttributeValue>Julius</AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
  <Attribute
   AttributeId="resource-id"
   DataType="anyURI">
   <AttributeValue>http://library.com/record</
        AttributeValue>
  </Attribute>
 </Resource>
 <Action>
  <Attribute
   AttributeId="action-id"
   DataType="string">
   <AttributeValue>write</AttributeValue>
  </Attribute>
 </Action>
</Request>
```

Listing 5. Final request 2

VI. An empirical evaluation

In this section, we discuss about the effectiveness of the test suite generated by X-CREATE. In particular, we provide a comparison between the tool Targen presented in [15], and X-CREATE in terms of fault-detection capability. Even if Targen does not consider the schema of requests, it can be considered the most similar tool to X-CREATE to be taken as a baseline for comparison, and it has been proven to perform better than a random generation technique [15].

Table II
Index of mutation operators

PTT	Policy Target True
PTF	Policy Target False
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
CRC	Change Rule Combining Algorithm
CRE	Change Rule Effect

Targen derives the set of requests satisfying all the possible combinations of truth values of the attribute id-value pairs found in the subject, resource, and action sections of each target included in the policy under test. We refer to [15] for more details.

For the comparison we used (see column 1 in Table III), three policies presented in [15] (specifically *demo-5, demo-11, demo-26*) and the policy presented in Listing 1, called in the rest of this section *PolicyExample*.

We applied mutation analysis which is a standard technique to assess the quality of a test suite in terms of fault detection [16]. Mutation has been applied here to introduce faults into the policies.

We generated the mutants set considering the mutation operators for XACML policies indicated in [17]. For the sake of completeness we report in Table II the set of used mutation operators. In particular, PTT, PTF, RTT, RTF, RCT and RCF emulate syntactic faults into the policy and rule target elements, and into the condition elements, producing an evaluation of the predicates included in those elements to always True or False. CRC and CRE, changing logical constructs of XACML policies, emulate semantic faults. For a more detailed description of the mutant operators we refer to [17].

We applied the mutation operators of Table II to the policies *demo-5*, *demo-11*, *demo-26*, and *PolicyExample*, obtaining (see second column in Table III): 23, 22, and 17 mutants for *demo-5*, *demo-11* and *demo-26* respectively [3], and 12 mutants for the *PolicyExample*. The sets of mutants obtained have been used for answering the two Research Questions:

- TSEff Is the test suite derived by X-CREATE more effective than that derived by Targen?
- TSIncr Is X-CREATE provided capability to vary test request number and structure useful to increase effectiveness?

Experimental answers to these questions are described below.

RQ TSEff. Applying the methodology described in [15] and implemented in Targen, we derived, to the best of our understanding, a set of requests for the *PolicyExample*

[3] These mutants numbers correspond to those of the Table 4 (second column) of [17] for the corresponding policies.

Table III
MUTANT-KILL RATIOS ACHIEVED BY TEST SUITES OF TARGEN AND X-CREATE

		Targen		**X-CREATE**			
				TSEff		TSIncr	
policy	# Mut	# Req	Mut Kill %	# Req	Mut Kill %	# Req	Mut Kill %
demo-5	23	20	78.95 %	20	86.96 %	35	95.65 %
demo-11	22	16	77.78 %	16	81.82 %	18	95.45 %
demo-26	17	16	78.57 %	16	94.11 %	9	94.11 %
PolicyExample	12	12	75 %	12	75 %	14	83.33 %

policy. Note that, we only derived by Targen the test suite for the *PolicyExample* because for the *demo-5*, *demo-11* and *demo-26* we referred to [15] both for the cardinality of the test suites and the percentage of mutants killed.

In parallel, by using X-CREATE, we generated the same number of requests generated by the Targen tool for each policy, so to get a fair comparison [4].

Finally, the test suite obtained by Targen [5] and those derived by X-CREATE [6] have been executed on the associated policies, which has been taken as the golden reference policies, and on their mutants. The same XACML implementation [18] has been used to validate the test suites of Targen and X-CREATE against each policy and its mutants. Each request in the requests set is executed on a policy and on each of its mutants, if the produced responses are different, then the mutant policy is killed by the request.

For each policy under test, Table III reports the number of requests executed and the percentage of mutants killed using the Targen test suite (3rd and 4th columns), and using X-CREATE (5th and 6th columns).

We did not perform a qualitative analysis of the mutants killed by the test suites derived by Targen and X-CREATE but we relied on the percentage of mutants killed for *demo-5*, *demo-11* and *demo-26*, presented in [15] for a mutant killing ratio comparison. Observing the obtained results we can deduce that the effectiveness of our test suites is comparable with, or higher than, that provided by the test suites derived by Targen tool.

In particular, an advantage of our approach is that it considers also the attribute id-value pairs found in the rule condition in addition to the those found in each policy and rule target.

RQ TSIncr. According to the description of the Targen tool provided in [15], a finite number of requests can be generated. This number is equal to all possible combinations of truth values of the attribute id-value pairs found in the subjects, resources, and actions of each target included in the policy.

[4]Note that, by X-CREATE it is possible to generate a user defined number of requests by deriving a defined number of intermediate instances by the TAXI tool (see Section V-A) and populating them with policy values as described in Section V-B and Section V-C.

[5]for *PolicyExample*

[6]for *demo-5*, *demo-11*, *demo-26* and *PolicyExample*

X-CREATE does not have such a limitation by providing a higher variability in the number of generated requests. As described in Section V, the number of structurally different requests derived by X-CREATE is up-bounded by the application of the XPT methodology to the Context Schema.

Then, we increased one at a time the number of intermediate instances derived by TAXI. We filled the new derived intermediate instance with a value not already assigned of the *ValuesSet* corresponding to the policy as described in Section V-C and evaluated the fault detection capability of the obtained test suite. In this way, we augmented the test sets associated to the four policies in the previous experiment, until the maximum percentage of mutants killed was achieved.

We report in the last two columns of Table III the final obtained results. Note that, the last column shows the maximum reachable percentage of mutants killed. The 100% was not feasible because in the set of mutants for *demo-5*, *demo-11* and *demo-26* there was an equivalent mutant while for *PolicyExample* there were two. In particular, for *demo-5*, *demo-11* and *demo-26* the equivalent mutants are those obtained by changing (CRC mutation operator) the existing rule combining algorithm with the *deny-overrides* algorithm. For *PolicyExample* the two equivalent mutants are those obtained by changing (CRC mutation operator) the existing rule combining algorithm with the *deny-overrides* and *permit-overrides* algorithms respectively.

The column before last represents the minimum number of requests needed for achieving the higher number of mutants killed for each policy. Note that for *demo-26* this number is smaller than the requests used before. In this case, because the maximum percentage was already reached (94.11% in the 6th column of Table III), we decremented the original set of requests till the percentage of fault detection effectiveness was guaranteed. Thus 9 requests instead of 16 were required.

The preliminary conclusions we can draw from this initial evaluation of X-CREATE are:

- Considering the mutant operators of [17], we observed a comparable or greater fault detection effectiveness than existing approaches, i.e., Targen.
- X-CREATE provides arbitrary variability in number of

generated requests, in their structure, and their values, which can result in improved effectiveness.

Of course such conclusions must be taken in light of the threats to validity of the performed experiment. The first point must be mitigated against the limited scope of the case study: we only compared X-CREATE to Targen, and only for few and small policies. We need to make larger experiments to generalize the statement. Moreover, we considered the mutation operators of [17], because these are those referred to by Targen evaluation. Different results might be observed using different mutation operators, e.g. those in [8]. Concerning internal validity, we referred, where available, to the results in [15] without repeating their experiment. Due to the limited dimension of the test suite we report the results on a single run of X-CREATE. In the future we plan to study larger case studies and collect average results of achieved mutation score over repeated executions of the experiment.

VII. Conclusions

Testing of access control systems is a critical activity for security assurance. Current approaches, which are generally based on the policy specification, do not exploit the XACML Context Schema, which establishes the rules to which access requests should conform. We propose here the testing framework called X-CREATE which exploits this schema. The framework consists of three main components: an intermediate-request generator, which is based on the XML Partition Testing (XPT) approach for request structures generation; a policy analyzer which selects the input values from the policy; and a values manager, which distributes the input values to the request structures. We get a generic conformance test suite, for all the XACML policies, that can be characterized as desired with specific policy values. We have performed a comparison between X-CREATE and the existing tool Targen in terms of fault-detection capability, and the results obtained show that X-CREATE has a similar or superior fault detection effectiveness, and yields a higher expressiveness, as it can generate requests showing higher structural variability. In future, we plan to assess X-CREATE application to the two envisaged testing scenarios: Testing the policy implementation correctness; Testing the policy specification correctness.

Acknowledgments

This work has been partially funded by EC FP7 under Grant Agreement N. 216287 (TAS[3] - Trusted Architecture for Securely Shared Services).

References

[1] OASIS, "eXtensible Access Control Markup Language (XACML) Version 1.0," http://www.oasis-open.org/committees/xacml/, February 2003.

[2] A. Pretschner, T. Mouelhi, and Y. Le Traon, "Model-based tests for access control policies," in *Proc. of ICST*, 2008, pp. 338–347.

[3] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[4] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. of ICSE*, 2005, pp. 196–205.

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

[6] K. Li, L. Mounier, and R. Groz, "Test generation from security policies specified in or-BAC," in *Proc. of COMPSAC*, 2007, pp. 255–260.

[7] W. Mallouli, J. Orset, A. Cavalli, N. Cuppens, and F. Cuppens, "A formal approach for testing security rules," in *Proc. of ACMT*, 2007, p. 132.

[8] Y. Traon, T. Mouelhi, and B. Baudry, "Testing security policies: going beyond functional testing," in *Proc. of ISSRE*, 2007, pp. 93–102.

[9] A. Masood, A. Ghafoor, and A. Mathur, "Test Generation for Access Control Systems that Employ RBAC Policies," SERC-TR-283, Purdue University, Tech. Rep., 2005.

[10] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, "Test-Driven Assessment of Access Control in Legacy Applications," in *Proc. of ICST*, 2008, pp. 238–247.

[11] N. Li, J. Hwang, and T. Xie, "Multiple-implementation testing for XACML implementations," in *Proc. of TAV-WEB*, 2008, pp. 27–33.

[12] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Automatic test data generation for XML schema-based partition testing," in *Proc. of AST*, May 2007.

[13] ——, "TAXI – A Tool for XML-Based Testing," in *Proc. of ICSE Companion*, 2007, pp. 53–54.

[14] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. on Soft. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

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

[16] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[17] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proc. of WWW*, May 2007, pp. 667–676.

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