Automatic XACML requests generation for policy testing

Antonia Bertolino, Said Daoudagh, Francesca Lonetti, Eda Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
via G. Moruzzi, 1 - 56124 Pisa, Italy
{firstname.lastname}@isti.cnr.it

***Abstract*—Access control policies are usually specified by the XACML language. However, policy definition could be an error prone process, because of the many constraints and rules that have to be specified. In order to increase the confidence on defined XACML policies, an accurate testing activity could be a valid solution. The typical policy testing is performed by deriving specific test cases, i.e. XACML requests, that are executed by means of a PDP implementation, so to evidence possible security lacks or problems. Thus the fault detection effectiveness of derived test suite is a fundamental property. To evaluate the performance of the applied test strategy and consequently of the test suite, a commonly adopted methodology is using mutation testing. In this paper, we propose two different methodologies for deriving XACML requests, that are defined independently from the policy under test. The proposals exploit the values of the XACML policy for better customizing the generated requests and providing a more effective test suite. The proposed methodologies have been compared in terms of their fault detection effectiveness by the application of mutation testing on a set of real policies.**

I. Introduction

XACML has become the de facto standard for specifying policies for access control decisions in many application domains such as Service Oriented Architectures (SOAs) and Peer-to-Peer (P2P) systems. Because of their key role the access control policies must be carefully designed and checked to avoid weaknesses and/or inaccuracies that could compromise either the correct data protection and management or let unauthorized access of data.

As for any other kind of systems the procedure applied for testing consists of: the definition or selection of a test strategy; the derivation of the test cases accordingly with the test strategy; the execution of the test cases on the system under test (SUT); the analysis of the obtained results. In this paper we consider as SUT the XACML policy specification, which is implemented in an engine called the PDP (Policy Decision Point). Thus the test cases are expressed in terms of XACML requests and used for probing the XACML policy, via PDP, and checking the PDP's responses against the expected ones.

Due to the complexity of XACML language, the policies definition and implementation is an error prone activity that represents a criticality in the definition of access control decision systems. Moreover the XACML requests generation could require a quite big effort to be manually managed, due to the many constraints to be satisfied.

So far the literature does not sufficiently pay attention on these problems, and few proposals for increasing the confidence on written policies are available.

For this, we implemented a set of specialized strategies, which systematically generate XACML requests by exploiting the syntax of the XACML request structure and the values defined in the XACML policy under test. We detail here two of them and we provide their comparison in term of fault detection effectiveness. The first presented strategy, called *Incremental XPT*, exploits the XACML Context Schema [1] which is the standard format of the XACML requests and a combination of policy values to better customize the derived requests.

In [2], we presented a preliminary version of this strategy, named in this paper *Preliminary XPT* (XML Partition Testing), and we provided an overview of the proposals focusing in particular on the performance of the Targen tool [3], one of the most complete tool for test cases derivation from XACML policy specification. Observing the obtained results we deduced that the effectiveness of the *Preliminary XPT* test suite was comparable with, or higher than, that provided by the test suites derived by Targen tool. Moreover, this previous study evidenced a set of issues useful for the development of a new strategy called *Incremental XPT* that we present in this paper.

The second testing strategy we propose here is called *Simple Combinatorial*. It implements a simple testing approach based on the number of possible combinations of the values of the subjects, resources, actions and environments of the XACML policy.

The two selected testing strategies are different in several aspects, but mainly in the higher structural variability of the requests obtained by the *Incremental XPT* compared with the simple structure of those obtained by the *Simple Combinatorial*. Experimental results are then necessary for defining guidelines for test strategy selection and application.

Summarizing, our contribution includes:

- the *Incremental XPT* that improves the *Preliminary XPT* aiming at reducing the number of derived requests;
- a new strategy called *Simple Combinatorial* and a related stop testing criterion;
- an improved procedure for test values generation;
- a preliminary empirical validation on real XACML poli-

cies, comparing the effectiveness of the two presented testing strategies.

The rest of this paper is structured as follows. Section II briefly presents the background while Section III details similar works. Section IV illustrates the motivations of the proposed approach. In Section V we present the new conceived *Incremental XPT* and *Simple Combinatorial* strategies. Section VI reports the experimental results. Finally, Section VII concludes the paper and gives related discussions.

II. Background

XACML [1] is a platform-independent XML-based language designed by the Organization for the Advancement of Structured Information Standards (OASIS). 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. At the root of all XACML policies there are the tags `<Policy>` or `<PolicySet>`. A policy set can contain other policies or policy sets.

A policy consists of a target, a set of rules and a rule combining algorithm. The target specifies the subjects (`<Subjects>` and `<Subject>`), resources (`<Resources>` and `<Resource>`), actions (`<Actions>` and `<Action>`) and environments (`<Environments>` and `<Environment>`) on which a policy can be applied.

Each `<Subject>`, `<Resource>`, `<Action>` and `<Environment>` contains two main attributes that are `<AttributeId>` and `<DataType>` and an `<AttributeValue>` that specifies the associated value.

If a request satisfies the target of the policy, then the set of rules of the policy is checked, else the policy is skipped.

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 a rule is a condition, that is a boolean function evaluated when the rule is applicable to a request. If the condition is evaluated to true, the result of the rule evaluation is the rule effect (*Permit* or *Deny*), otherwise a *NotApplicable* result is given. If an error occurs during the application of a policy to the request, *Indeterminate* is returned.

In a policy, more than one rule 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. For example, the *first-applicable* algorithm returns the effect of the first applicable rule or *NotApplicable* if no rule is applicable to the request. The *permit-overrides* algorithm specifies that *Permit* takes the precedence regardless of the result of evaluating any of the other rules in the combination, then it returns *Permit* if there is a rule that is evaluated to *Permit*, otherwise it returns *Deny* if there is at least a rule that is evaluated to *Deny* and all other rules are evaluated to *NotApplicable*. If there is an error in the evaluation of a rule with *Permit* effect and the other policy rules with *Permit* effect are not applicable, the *Indeterminate* result is given. Similarly, the *deny-overrides* algorithm returns *Deny* if there is a rule that is evaluated to *Deny* [1]. More policies in a policy set are combined according to the policy combining algorithm attribute, that has the same behavior as the rule combining algorithm. The access decision is given by considering all attribute and element values describing the subjects, resources, action and environment of an access request and comparing them with the attribute and element values of a policy.

We show in Listing 1 an example of a simplified XACML policy ruling library access. Its target (line 3) says that this policy applies to any subject, resource and action. This policy has a first rule *ruleA* (lines 4-29) with a target (lines 5-28) specifying that this rule applies only to the access requests of a "write" action of "http://library.com/record/book" and "documententry" resources. The effect of the second rule *ruleB* (lines 30-57) 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 2) is *permit-overrides*.

III. Related Work

Policy testing is a critical issue and the complexity of the XACML language specification 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.

Related work entails on one side analysis of policy specifications aimed at verifying that they properly reflect some intended properties, and on the other testing of the policy implementation (i.e. the PDP), to ensure that this conforms to the specifications.

Works in the first group use different verification techniques, such as model-checking [4] and SAT solvers [5]. Probably the most referred tool for policy analysis is Margrave [6], which represents policies as Multi-Terminal Binary Decision Diagrams (MTBDDs) and can answer queries about policy properties. MTBDDs represent the policy elements and the policy decision outcomes corresponding to their combinations.

Some existing approaches consider the policy values in the test cases derivation. In particular, [3] presents the Targen tool that 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. The set of strategies we propose here exploits the potentiality of the XACML Context schema defining the format of the test inputs, and/or applies combinatorial approaches to the policy values. In [2] a comparison between the *Preliminary XPT* methodology and the existing tool Targen has been performed in terms of fault-detection capability, and the results obtained showed that the *Preliminary XPT* has a similar or superior fault detection effectiveness, and yields a higher expressiveness, as it can generate requests showing higher structural variability.

[1] The behavior of *ordered-deny-overrides* and *ordered-permit-overrides* rule combining algorithm is identical to that of *deny-overrides* and *permit-overrides* except that the set of rules is evaluated in the order by which rules are listed in the policy.

```
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
 PolicyId="policyExample" RuleCombiningAlgId="permit-overrides">
 <Target/>
 <Rule RuleId="ruleA" Effect="Deny">
  <Target>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="anyURI-equal">
      <AttributeValue DataType="anyURI">http://library.com/record/book</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id1" DataType="anyURI"/>
     </ResourceMatch>
    </Resource>
    <Resource>
     <ResourceMatch MatchId="string-equal">
      <AttributeValue DataType="string">documententry</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id2" DataType="string"/>
     </ResourceMatch>
    </Resource>
   </Resources>
   <Actions>
    <Action>
     <ActionMatch MatchId="string-equal">
      <AttributeValue DataType="string">write</AttributeValue>
      <ActionAttributeDesignator AttributeId="action-id1" DataType="string"/>
     </ActionMatch>
    </Action>
   </Actions>
  </Target>
 </Rule>
 <Rule RuleId="ruleB" Effect="Permit">
  <Target>
   <Subjects>
    <Subject>
     <SubjectMatch MatchId="string-equal">
      <AttributeValue DataType="string">Julius</AttributeValue>
      <SubjectAttributeDesignator AttributeId="subject-id" DataType="string"/>
     </SubjectMatch>
    </Subject>
   </Subjects>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="anyURI-equal">
      <AttributeValue DataType="anyURI">http://library.com/record/journals/</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id3" DataType="anyURI"/>
     </ResourceMatch>
    </Resource>
   </Resources>
   <Actions>
    <Action>
     <ActionMatch MatchId="string-equal">
      <AttributeValue DataType="string">read</AttributeValue>
      <ActionAttributeDesignator AttributeId="action-id2" DataType="string"/>
     </ActionMatch>
    </Action>
   </Actions>
  </Target>
 </Rule>
</Policy>
```

Listing 1: A XACML policy

A different approach is provided by Cirg [7] that is able to exploit change-impact analysis for test cases generation starting from policies specification. In particular, it integrates the already cited Margrave tool [6] which performs change-impact analysis so to reach high policy structural coverage.

Other approaches for policy testing are based on representation of policy implied behavior by means of models [8], [9]. Usually these approaches provide methodologies or tools for automatically generating abstract test cases that have to be then refined into concrete requests for being executed.

IV. Motivation and Research Questions

The lesson learned from the comparison between the Targen tool and the *Preliminary XPT* [2] was useful for highlighting the peculiarities and the issues that could be exploited for improving the fault detection effectiveness of the *Preliminary XPT*. In particular, the most important feature of the *Preliminary XPT* was the structure variability of the derived requests: i.e., a request may include more than one subject, resource, action, or environment entity. This feature was especially important for testing policies or rules in which the access decision involves simultaneously more than one subject or resource or action or environment. However, the possibility of having a high variability in the structure of requests was also the main limitation of the *Preliminary XPT*.

As described in [2], the amount of requests that can be generated by varying the structure of the original XACML Context Schema was *MAXREQ_Preliminary* = 118098, which was extremely high for any policy specification. Even if the test strategy implemented some approaches for reducing and ordering the instances by keeping the fault detection capability, the final number of generated requests was still unmanageable.

From this experience we understood that an effective criterion for selecting the useful test suite to be executed was necessary. The criterion should guarantee a manageable number of test cases without decreasing the fault detection effectiveness. For this we worked into three directions:

- introduce a criterion for selecting the proper number of requests;
- derive a completely new test strategy, called the *Simple Combinatorial*, for the request generation that could satisfy the criterion and be the simplest as possible;
- improving the *Preliminary XPT* by reducing the overall amount of the derived requests keeping all the good peculiarities and advantages of it. So to differentiate, the new strategy is called *Incremental XPT*.

The *Simple Combinatorial* strategy focuses only on the values combination. The requests are generated so to cover all the possible combinations of the values of the subjects, resources, actions and environments of the policy. Because this testing strategy is the simplest to be conceived, the number of requests generated by the *Simple Combinatorial* approach could be used as a stopping criterion in the test cases generation for *Incremental XPT*, and considered for a fair comparison with the first one. In particular, for both methodologies we generate as many requests as necessary to cover the input domain of a XACML policy, i.e. the combinations of its

values. Then considering the defined stopping criterion, we compare the effectiveness of the *Simple Combinatorial* with that of the *Incremental XPT* by answering to the following research questions:

TSEff: Adopting the proposed stopping criterion, is the fault detection of the *Simple Combinatorial* strategy similar to that of the *Incremental XPT* one?

TSDecr: Is it possible to reduce the test suites maintaining the same level of fault detection?

TSIncr: Is it possible to increase the *Incremental XPT* fault detection?

Further details of the test strategies and their comparison are provided in the next sections.

V. Testing Strategies

We describe in Section V-A an improved version of the *Preliminary XPT*, called *Incremental XPT* while in Section V-B, we describe a new conceived testing strategy called *Simple Combinatorial*.

A. Incremental XPT

For aim of completeness in this section we summarize some of the steps of the *Preliminary XPT* strategy so to better highlight the main differences. The *Preliminary XPT*, as presented in [2], consists of three main steps: *(i)* intermediate-request generation; *(ii)* policy-under-test analysis; *(iii)* request values assignment. In this paper we focus on the modifications and improvements of the first two steps, while we summarize in Section V-A3 the procedure already used in the *Preliminary XPT* strategy for the requests values assignment.

1) Intermediate requests cardinality: In the *Preliminary XPT*, given the XACML Context Schema, a set of conforming XML instances is generated by applying a variant of the Category Partition (CP) method [10] and traditional boundary conditions. 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.

Combining the occurrence values assigned to each element, the set of intermediate instances are generated. In such manner, given the Context Schema, up to $3^Y * 2^Z$ intermediate instances can be derived, 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 2.0 Context Schema, only for the part concerning the requests specification, there are 10 elements with unbounded occurrence and 1 having [0,1] cardinality. The other elements have cardinality 1. Thus, in the *Preliminary XPT* varying the structure of the original XACML 2.0 Context Schema (only for the part concerning the requests specification) we generated a maximum number of *MAXREQ_Preliminary* = $3^{10} * 2^1$ = 118098 structurally different intermediate requests.

This huge number of generated intermediate requests is obviously unmanageable for testing purposes so we define a new criterion: use only one value for the `<AttributeValue>` element. For this we set the `minOccurs` and `maxOccurs` occurrences of the `<AttributeValue>` element and those of the contained `<Any>` element equal to one, generating consequently a modified XACML 2.0 Context Schema version. Moreover, we set to zero the `minOccurs` and `maxOccurs` occurrences of the `ResourceContent` element and those of the contained `<Any>` element because our test values generation does not deal with the XPath resolution. In this manner we obtained a high reduction on the total number of the intermediate requests because the modified schema version shows only six elements with unbounded cardinality and zero elements having [0,1] cardinality. Thus *Incremental XPT* strategy generates *MAXREQ_Incremental* = 3^6 = 729 structurally different intermediate requests, that is a more manageable number of test cases. In Figure 1 we show a sketch of the modified XACML 2.0 Context Schema where the elements occurrences have been set to (1,2,3)[2].

2) Entity definition: The set of intermediate requests derived by the *Incremental XPT* are general and do not contain values so to better customize them to a particular XACML policy a selection of the values of elements and attributes of the policy under test has to be performed.

In the *Preliminary XPT* we used for this purpose four values sets called *SubjectSet*, *ResourceSet*, *ActionSet* and *EnvironmentSet* respectively. Those sets were filled with the values of the `<AttributeValue>` elements and the values of the attributes named `<AttributeId>`, `<Datatype>`, `<Issuer>` and `<SubjectCategory>` referring to the `<Subjects>`, `<Resources>`, `<Actions>` and `<Environments>` elements of the policy respectively. These elements and attributes appear in the `<Target>` element of each `<Rule>`, `<Policy>` and `<PolicySet>` element included in the policy under test and in the `<Condition>` section (see Section II for details about XACML language). These attributes and elements values where then combined in order to obtain the entities. Specifically, a *subject entity* was defined as a combination of the values of elements and attributes of the *SubjectSet* set, and similarly the *resource entity*, the *action entity* and the *environment entity* are a combination of the values of the elements and attributes of the *ResourceSet*, *ActionSet*, and *EnvironmentSet* respectively. For robustness and negative testing purposes to each set of data (*SubjectSet*, *ResourceSet*, *ActionSet*, and *EnvironmentSet*) a random value for `<AttributeValue>` `<AttributeId>` and `<Datatype>` was added.

However combining all possible attributes and elements of each set has the main disadvantage of the high cardinality of the derived entity set without any guarantee of an increasing of fault detection effectiveness. A more accurate analysis of the results obtained in the experiment conducted with the *Preliminary XPT* evidenced that some of the combinations could be meaningless considering the rules and constraints expressed by the XACML functions. Specifically, the attributes

[2]1,2,3 are the values used in the current implementation of the *Preliminary XPT* presented in [2].

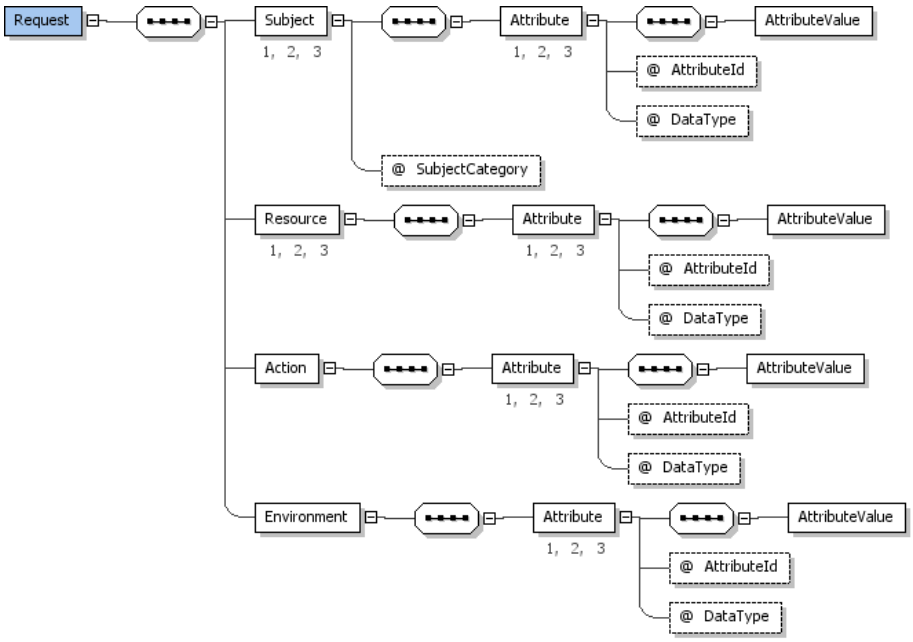

Fig. 1: Sketch of the modified XACML 2.0 Context Schema for intermediate requests derivation

<AttributeId> and <Datatype> are related to the value of the <AttributeValue> element and this relation is specified by a XACML function.

For example, in the policy of Listing 1, the <AttributeValue> element (line 15) that has the value equal to *documententry* is related to the <AttributeId> and <Datatype> equal to *resource-id2* and *string* respectively. The entity containing *documententry*, *resource-id2* and *string* is a meaningful resource entity. The requests containing all other combinations of *documententry* with other <AttributeId> and <Datatype> in the *ResourceSet* (for example with *resource-id1* and *anyURI*) are not applicable to the *ruleA* rule (lines 4-29) because the function *string-equal* (line 14) is not matched.

From this, our new idea for improving the *Preliminary XPT*: an entities generation approach that considers all policy functions and derives all combinations of <AttributeValue> <AttributeId> and <Datatype> that satisfy the defined XACML functions. In addition, we reduce also the number of random entities of each set by adding a random entity for each <Datatype> that has a random value for the <AttributeValue> element and one <AttributeId> value randomly chosen among entities containing the same <Datatype>. In this way we decrease the number of entities-for each set of data (*SubjectSet*, *ResourceSet*, *ActionSet* and *EnvironmentSet*). For the *ResourceSet* derived from the policy of Listing 1, the new introduced approach derives 5 entities against 48 of the *Preliminary XPT*. The applicable entities are 3 in both cases.

3) Request values assignment: Once derived the set of intermediate requests the values of elements and attributes of the policy under test are used so to obtain executable and meaningful final requests. We adopted for the *Incremental XPT* the same procedure of the *Preliminary XPT*. For aim of completeness we report in the rest of this section the main details.

The combinations of *subject entities*, *resource entities*, *action entities* and *environment entities* are used to fill the values of elements and attributes of the subject, resource, action and environment of an intermediate instance respectively. Specifically, depending on the number of intermediate requests to complete, the values entities are selected by applying an incremental combination approach [11], [9] so that all the possible combinations of the values of *subject entities*, *resource entities*, *action entities* and *environment entities* can be considered. The procedure adopted for generating the test requests takes one by one the combinations of entities values and completely fills the set of the intermediate requests according to the following considerations:

- derive all combinations of *subject entities*, *resource entities*, *action entities* and *environment entities* by: first, apply the pair-wise combination and we obtain the *PW* set; then, apply the three-wise combination and we obtain the *TW* set; finally, apply the four-wise combination and we obtain the *FW* set. These sets have the following inclusion property $PW \subseteq TW \subseteq FW$. For eliminating duplicated combinations we consider the following set of combinations: *PW* called *Pairwise*, $TW \setminus PW$ called *Threewise* and $FW \setminus (TW \cup PW)$ called *Fourwise*. We use these combinations to fill the values of elements and attributes of the subject, resource, action and environment of an intermediate instance respectively;
- if all the combinations of *subject entities*, *resource entities*, *action entities* and *environment entities* have been already used for filling a subset of intermediate requests, then start again by selecting the *subject entity*, *resource*

entity, *action entity* and *environment entity* combinations in the same order till the completion of the intermediate requests set;
- in the case in which the structure of an intermediate request requires more than one entity in the subject, resource, action and environment, then the necessary entities are randomly taken from the available ones avoiding duplicates.

More details about the values assignment procedure of the entities to the intermediate requests are in [2].

B. Simple Combinatorial Testing Strategy

In the *Simple Combinatorial* testing strategy a combinatorial approach is applied to the policy values. As in the *Incremental XPT*, this methodology requires the definition of the *SubjectSet*, *ResourceSet*, *ActionSet*, *EnvironmentSet*, *subject entity*, *resource entity*, *action entity* and *environment entity* as in Section V-A3 and V-A2.

In particular, as for the *Incremental XPT* we create the *PW* set, *TW* set and *FW* set with the inclusion property $PW \subseteq TW \subseteq FW$. Then for each combination included in the above sets, we generate a simple request containing the entities of that combination. The derived requests are first those obtained using the combinations of the *Pairwise* set, then those ones using the combinations of the *Threewise* set and finally those using the combinations of the *Fourwise* set. In this way, we try to generate a test suite guaranteeing a coverage first of all pairs, then of all triples and finally of all quadruples of values entities derived by the policy. The maximum number of requests derived by this strategy is equal to the cardinality of the *FW* set.

Thus the main difference between the instances generated with *Incremental XPT* and *Simple Combinatorial* testing strategies consists in the cardinality of subjects, resources, actions and environments considered in each generated request: in *Simple Combinatorial* testing strategy the cardinality of each group is almost 1.

The main advantage of the proposed strategy is that it is simple and achieves the coverage of the policy input domain represented by the policy values combinations.

VI. Experimental Results

In this section, we discuss about the effectiveness of the *Simple Combinatorial* strategy and the *Incremental XPT* in terms of fault-detection capability. The test suites of the two test strategies were applied by a tool called X-CREATE[12]. For the comparison we used (see first column of Table I): three policies presented in [3] (specifically demo-5, demo-11, demo-26); a set of real policies ruling a health care service (specifically read-patient, dashboard) and an university administration server implemented in the TAS3 project [13]. For aim of completeness, we report also some data about the structure of the mentioned policies. Specifically, the columns of Table I represent the number of rules, conditions, subjects, resources, actions and distinct functions within each policy. For testing the effectiveness of the proposed techniques we applied mutation analysis which is a standard technique to assess the quality of a test suite in terms of fault detection [14]. Specifically, by means of mutation operators, faults can be introduced into programs (in this case into the policies of Table I). Thus we generated the mutants set using the mutation operators for XACML policies indicated in [15]: for instance insert syntactic faults into the policy, rule, target and condition elements, changing logical constructs or emulate semantic faults and so on. Table II in the second column, shows the total amount of mutants obtained for the polices of Table I. The sets of mutants obtained have been used for answering the three Research Questions, labeled **TSEff**, **TSDecr**, **TSIncr**, proposed in Section IV. Experimental answers to these questions are described below.

RQ TSEff. Applying the *Simple Combinatorial* strategy we obtained the test suites for each of the policies considered. We report in the third column of Table II the cardinality of each test suite. In parallel, by using *Incremental XPT*, for each policy we generated the same number of requests as indicated by the *Simple Combinatorial* strategy, so to get a fair comparison. Finally, the test suites obtained by the *Simple Combinatorial* strategy and those derived by the *Incremental XPT* one have been executed on the associated policies and on their mutants. The same XACML policy implementation engine, the PDP (Policy Decision Point)[16], has been used in both the experiments to execute and validate the test suites of *Simple Combinatorial* and *Incremental XPT* against each policy and its mutants. Specifically for each test set, each request was executed on a policy and on each of its mutants. If the produced responses were different, then the mutant policy was considered killed by the request. For each policy we report in Table II the percentage of mutants killed using *Simple Combinatorial* strategy (4th column) and *Incremental XPT* one (8th column).

Observing the obtained results we could deduce that the effectiveness of the *Incremental XPT* test suites is generally higher than that provided by the test suites of the *Simple Combinatorial* strategy (see 4th column v.s. 8th column of Table II). Looking at the Table II, in two cases (*create-document* and *read-information-unit*) the fault detection of the *Simple Combinatorial* is higher than that of *Incremental XPT* (75% v.s. 45%, 73% v.s. 66%). This evidences that even a simple and easy-to-apply test strategy, as the *Simple Combinatorial*, can improve the testing activity and in some specific cases provide interesting results. This happens mainly when policies are simple and with few values so that the test cases provided by the *Simple Combinatorial* could cover quickly all the possible features implemented in the analyzed policies. However, when the complexity of the policies structure increases, Table II shows cases in which there is an incredible higher fault detection effectiveness of the *Incremental XPT* with respect to that of the *Simple Combinatorial* strategy. For instance, for *dashboard* policy the *Incremental XPT* gets 55.88% of mutants killed while the *Simple Combinatorial* only 2.94%.

RQ TSDecr. The next point we analyzed was the possibility of reducing the test suites of the two test strategies maintaining

TABLE I: POLICIES DATA

Policy	# Rule	# Cond	# Sub	# Res	# Act	# Funct
demo-5	3	2	2	3	2	4
demo-11	3	2	2	3	1	5
demo-26	2	1	1	3	1	4
student-application-1	2	0	5	2	2	2
student-application-2	2	0	11	2	2	2
create-document	3	2	1	2	1	3
read-document	4	3	2	4	1	3
delete-document	3	2	1	3	1	3
read-information-unit	2	1	0	2	1	2
read-patient	4	3	2	4	1	3
dashboard	6	5	3	3	0	4
university-admin-1	3	0	24	3	3	2
university-admin-2	3	0	24	3	3	2
university-admin-3	3	0	23	3	3	2

TABLE II: Mutant-kill ratios achieved by test suites of *Simple Combinatorial* and *Incremental XPT*

		Simple Combinatorial				**Incremental XPT**					
		TSEff		TSDecr		TSEff		TSDecr		TSIncr	
Policy	# Mut	# Req	Mut Kill %	# Req	Mut Kill %	# Req	Mut Kill %	# Req	Mut Kill %	# Req	Mut Kill %
demo-5	23	84	100 %	56	100 %	84	100 %	23	100 %	-	-
demo-11	22	40	95.45 %	13	95.45 %	40	100 %	25	100 %	-	-
demo-26	17	16	58.82 %	6	58.82 %	16	58.82 %	12	58.82 %	94	94.11 %
student-application-1	15	16	60 %	3	60 %	16	60 %	6	60 %	31	93.33 %
student-application-2	15	32	60 %	2	60 %	32	60 %	32	60 %	38	93.33 %
create-document	20	12	75 %	9	75 %	12	45 %	10	45 %	172	80 %
read-document	25	30	40 %	2	40 %	30	56 %	25	56 %	75	88 %
delete-document	20	16	60 %	3	60 %	16	65 %	16	65 %	96	85 %
read-information-unit	15	6	73 %	4	73 %	6	66 %	2	66 %	10	80 %
read-patient	25	30	40 %	2	40 %	30	56 %	25	56 %	50	88 %
dashboard	34	60	2.94 %	1	2.94 %	60	55.88 %	18	55.88 %	18	55.88
university-admin-1	20	80	50 %	2	50 %	80	65 %	43	65 %	211	75 %
university-admin-2	20	80	75 %	21	75 %	80	85 %	70	85 %	253	95 %
university-admin-3	20	76	50 %	2	50 %	76	75 %	43	75 %	43	75 %

the same level of fault detection, i.e. the values of columns 4 and 8 of Table II. This analysis was necessary to evaluate if the decision of executing as many test cases as the possible combinations derivable by the *Simple Combinatorial* strategy was an effective stopping criterion also for the *Incremental XPT*. Thus, wherever the test suite reduction was applicable, starting from the first request ahead, following the order in which the requests have been generated, we incrementally derived the score of mutants killed of the two test sets till we reached the values reported in the 4th and 8th columns of Table II. The cardinality of the subsets are in 5th and 9th columns of the Table II. As shown, for the *Simple Combinatorial* strategy the maximum reachable percentage is reached with a few number of requests for all policies, while for *Incremental XPT* for 7 policies over the 12 considered the maximum reachable percentage is reached with almost half of the available requests (or even much less in some cases). Thus the considered stopping criterion is a good upper bound assuring a good fault detection effectiveness with a manageable low number of requests.

RQ TSIncr. Last question we analyzed was the evaluation of the maximum reachable fault detection executing all the 729 available requests for the *Incremental XPT*. This let us to evaluate: the effectiveness of the proposed strategy; the loss in the fault detection effectiveness due to the application of the stopping criterion. Thus for all the available policies we executed the full pull of available requests; results obtained are in last column of the Table II. As shown in the table, the percentage of mutants killed could be increased a lot, reaching in some cases values more than 90%. This is a point in favor of the new conceived *Incremental XPT* and confirms that the reduction in the intermediate requests generation adopted did not impact on the effectiveness of the strategy with respect to the *Incremental XPT*. Then for each policy we calculated the minimum number of requests necessary for reaching the maximum fault detection effectiveness. Specifically, starting from the first request we calculated incrementally the fault detection till the highest value was reached. These values are reported in the 11th column of Table II. The analysis of these new data showed that in most of the cases the maximum reachable value of the fault detection effectiveness was reached adding not an extremely high number of requests to that established by the adopted stopping criterion. Moreover, in most of the cases the loss of fault detection effectiveness due to the application of the stopping criterion is around the 15%. This was another confirmation that the stopping criterion

adopted was quite good.

VII. Discussion and Conclusions

In this paper we proposed two different testing strategies for the automated generation of the test inputs for the XACML policy, focused on the coverage of the policy input domain. In particular, the former called *Incremental XPT*, exploits the XACML Context Schema [1] representing the format of the XACML requests and the combinations of policy values, the latter named *Simple Combinatorial* is based on the combination of values taken by the policy itself. We also proposed a comparison of the effectiveness of the proposed strategies considering as a stopping criterion the cardinality of all possible combinations of values of the policy.

The preliminary conclusions we can draw from this initial evaluation are:

- a greater fault detection percentage of the *Incremental XPT* due to the variability of the structures of the generated requests, which could result in an improved effectiveness of the derived test suite;
- it is possible to reduce the number of requests for both strategies keeping the same test effectiveness. That means that the introduced stopping criterion is a good upper bound. However, further criteria for test reduction could be conceived.

Note that, the percentage of mutants killed by the test suite derived by *Simple Combinatorial* strategy is the maximum reachable. As it was conceived, it is not possible to include additional test cases to the test suite and consequently to get higher value of fault detection. On the contrary, for the *Incremental XPT* it is possible to increase the test suite cardinality up to a maximum number of *MAXREQ_Incremental* = 729 requests as described in Section V-A1.

Of course such conclusions must be taken in light of the threats to validity of the performed experiment. The presented study is limited to a small set of real policies, then further experimentation would be required. We measured the fault detection effectiveness of *Simple Combinatorial* and *Incremental XPT* by referring the mutation operators of [15], and due to the limited dimension of the test suite we report the results of a single application of the proposed strategies run of X-CREATE. We need to collect average results of achieved mutation score over repeated executions of the experiment and consider further mutation operators than those of [15]. Moreover, in the entities definition procedure described in Section V-A2 we take into account only the main XACML functions, in the future we plan to address all the other functions. 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.

From the performed analysis we noticed an impact of the policy specification on the effectiveness of the derived test suite. Thus, we would like to investigate other methodologies for requests generation taking into account this. In particular, a limitation of *Simple Combinatorial* strategy was that it is not able to detect situations where the satisfiability of the policy rules depends simultaneously on the values of more than one entity. We would like to force by construction the requests derived by this strategy to contain all the possible combinations of more than one subject, resource, action and environment entity. As a future work, we plan to investigate about the comparison between *Incremental XPT* approach and this new test inputs derivation proposal in terms of fault detection effectiveness.

Acknowledgment

This work has been partially funded by EC FP7 under Grant Agreement N. 216287 (TAS^3 - Trusted Architecture for Securely Shared Services) project, by the Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS) FP7 Project contract n. 256980 and by EU FP7 IP n. 257178 CHOReOS (Large Scale Choreographies of the Future Internet) project.

References

[1] 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, February 1 Feb 2005.

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

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

[4] N. Zhang, M. Ryan, and D. Guelev, "Evaluating access control policies through model checking," in *Information Security*, ser. Lecture Notes in Computer Science, J. Zhou, J. Lopez, R. Deng, and F. Bao, Eds. Springer Berlin / Heidelberg, 2005, vol. 3650, pp. 446–460.

[5] G. Hughes and T. Bultan, "Automated verification of access control policies," *Technical Report 2004-22. Comp. Science Dep., Univ. of California, Santa Barbara, CA*, 2004.

[6] 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.

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

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

[10] 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.

[11] 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.

[12] X-CREATE, http://labsewiki.isti.cnr.it/labse/tools/xcreate/public/main.

[13] TAS3 Project, "Trusted Architecture for Securely Shared Services," http://www.tas3.eu/.

[14] 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.

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

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