A Toolchain for Model-Based Design and Testing of Access Control Systems

Said Daoudagh[1], Donia El Kateb[2], Francesca Lonetti[1], Eda Marchetti[1] and Tejeddine Mouelhi[3]
[1]*Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy*
[2]*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*
[3]*itrust consulting, Niederanven, Luxembourg*
{*said.daoudagh, francesca.lonetti, eda.marchetti*}*@isti.cnr.it, donia.elkateb@uni.lu, mouelhi@itrust.lu*

Keywords: XACML Language, Model-based-testing, Access control systems.

Abstract: In access control systems, aimed at regulating the accesses to protected data and resources, a critical component is the Policy Decision Point (PDP), which grants or denies the access according to the defined policies. Due to the complexity of the standard language, it is recommended to rely on model-driven approaches which allow to overcome difficulties in the XACML policy definition. We provide in this paper a toolchain that involves a model-driven approach to specify and generate XACML policies and also enables automated testing of the PDP component. We use XACML-based testing strategies for generating appropriate test cases which are able to validate the functional aspects, constraints, permissions and prohibitions of the PDP. An experimental assessment of the toolchain and its use on a realistic case study are also presented.

1 INTRODUCTION

Nowadays many IT systems are required to satisfy several properties orthogonal to the main functionalities, one of them is security. Resources, e.g., data, machines or services, could be sensitive and valuable, and hence a proper security support must be put in place to protect them against unauthorized, malicious, improper or erroneous usage. For ICT systems, authorization mechanisms provide the procedures for managing data and resources according to the security policy requirements.

Commonly adopted authorization systems, based on XACML (eXtensible Access Control Markup Language), rely on the same architectural model which includes in a simplified version three main components: a Policy Enforcement Point (PEP), a Policy Decision Point (PDP) and an Attribute Manager. PEP is an application dependent component which intercepts and transmits any incoming access request to the Policy Decision Point (PDP). The latter, using specific attribute values collected by the application dependent Attribute Manager, will grant or deny the access based on a set of established rules defining usage decision process. The critical core of the access control system is therefore the PDP which is an application independent component in charge of regulating the accesses to the protected data and resources.

Several PDP have been defined in scientific literature, and some implementations of this component, both academic, free and commercial, are currently available, such as the SUN's XACML engine (Sun Microsystems, 2006), named Sun PDP in the rest of the paper. However, even if these implementations could be ready-to-use solutions for the access control system development, in the last years the adoption of domain specific PDPs, able to not impact on the overall performance of IT systems and to be compliant with specific implementation constraints becomes a more common attitude. The use of either a ready-to-use solution or a customized version of the PDP component would require careful verification and testing process before its integration into the access control system: any fault or problem in the PDP may result in a critical failure of the overall security system. Unfortunately, PDP testing is per se a time and effort consuming activity that hardly matches the strict ICT market requirements.

To this purpose this paper presents a first attempt of a toolchain for automatic testing of the PDP component. The toolchain provides different facilities: i) a model-driven approach to specify and generate security policies; ii) a test case generation strategy and automated execution of the test suite on the target PDP; iii) a model-based oracle and an automated checker able to derive the expected results and com-

pare them with test results; iv) a set of predefined policies with the associated test cases and expected results. An experimental assessment of the toolchain and its use for testing a specific PDP implementation, the Sun PDP (Sun Microsystems, 2006), are also presented.

The remainder of this paper is structured as follows: Section 2 presents background about XACML language. In Section 3 we present the toolchain and its underlying functions, while a case study shows how to use the different components of the toolchain for testing Sun PDP in Section 4. Finally, Section 5 outlines related work whereas Section 6 concludes the paper also hinting at future work.

2 THE XACML LANGUAGE

XACML (OASIS, 2005) is a de-facto standardized specification language that defines access control policies and access control decision requests/responses in an XML format. An XACML policy defines the access control requirements of a protected system. An access control request triggers a policy evaluation and aims at accessing a protected resource. The PDP evaluates the request against the rules in the policy and returns the access response (Permit/-Deny/NotApplicable/Indeterminate) according to the specified XACML policy. An XACML policy can contain one or more *policy set* or *policy* elements. A *policy set* or *policy* includes a *target* (containing a set of subjects, a set of resources, a set of actions and finally a set of environments) and one or more *rules*. A *rule* contains a decision type (Permit or Deny) and a *target*. A rule contains a *condition* element, i.e., a boolean function that specifies constraints on the subjects, resources, actions and environments values so that if the *condition* evaluates to true, then the rule's decision type is returned. A *combining algorithm* is used to select which policy (*policy-combining algorithm*) or rule (*rule-combining algorithm*) has to be considered in case the request matches more than one policy (or rule). For instance, the *first-applicable* combining algorithm will select the first applicable policy (or rule). An access request contains subject, resource, action, and environment attributes. At the decision making time, the Policy Decision Point evaluates an access request against a policy, by comparing all the attributes in an access request against the attributes in all the *target* and *condition* elements of the *policy set*, *policy* and *rule* elements. If there is a match between the attributes of the request and those of the policy, the *effect* of a matching rule is returned, otherwise the *NotApplicable* decision is drawn.

3 TOOL CHAIN ARCHITECTURE

In this section we present the proposed toolchain, that consists of three main parts:
Model-driven Policy Design composed by:

- a modeling framework for specifying security requirements, i.e. a graphical security model aiming at simplifying the designing of security constraints (Ecore Model);
- an automated translation of the graphical security model into an XACML policy, so to avoid the common errors and problems due to the writing of XACML policies (Ecore2XACML);
- an automated model driven oracle for deriving the expected response of each request (Oracle).

Test Case Generation and Execution composed by:

- an automated tests generation according to different testing strategies to speed up and improve the testing process by reducing as much as possible time and effort due to test cases specification (X-CREATE);
- an automated execution of test cases on the SUT PDP i.e. a selected access control engine (Test Executor);
- an automated analysis of test results against the expected ones (Checker).

Test Archive which consists of a repository of predefined policies, test sets and expected responses (Test Archive).

Figure 1 schematizes the architecture of the proposed toolchain that can be used either for model-driven testing (steps from 1 to 8) or adopting predefined test sets (steps from 1' to 6'). More technical details about the components of the toolchain are provided in the following sections.

3.1 Model based specification

Figure 2 illustrates the security policy metamodel adopted in the proposed toolchain. The metamodel is defined using the following classes: i) *POLICYType*: it defines a set of element types *ElementType* and a set of rule types *RuleType*; ii) *Parameter*: it has a type which must belong to the *ElementType* of the *PolicyType*; iii) *RuleType*: it has a set of *Parameters* that are typed by *ElementTypes*.

The meta-model allows the definition of a generic security policy model and the definition of a security policy formalism according to this security model. The security policy metamodel can be instantiated into an XACML metamodel which targets the main

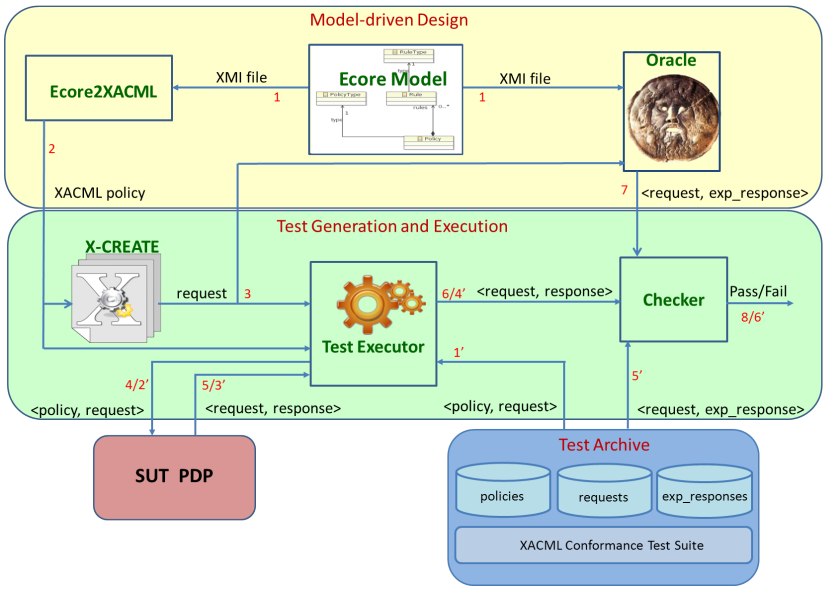

Figure 1: Toolchain

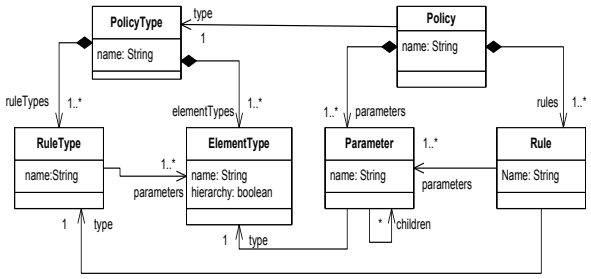

Figure 2: Security Policy Metamodel

specific elements and constraints of XACML language. Figure 3 presents the XACML metamodel, which is an instance of the security policy metamodel depicted in Figure 2. As shown in the Figure, there are five `RuleType` elements:

- PS_Comb: it is composed of a policySet (PS), which is the name of the policy set, and the policy combining algorithm (PSComb) associated to the policySet;
- P_Comb: it contains a policy name (Policy) and the associated rule combining algorithm (RuleComb). According to the XACML language specification the definition of at least one policy is mandatory;
- PS_Policy: it is composed of a policySet name (PS), and the policy name (Policy) that is contained in that policySet;
- P_Rule: it contains a policy name (Policy) and the rule name contained in that policy (RuleName);
- XACML_Rule: it contains a subject (Subject), an action (Action), a resource (Resource), an environment (Env), a decision (Decision) and a rule name (RuleName).

In order to define an XACML policy model, we need to create an instance of the XACML metamodel according to the following steps:

- define the policy structure, that is to create a list of all XACML policy elements: subjects, actions, resources, environments, policies, rules names, policy sets names, decision (predefined values: permit and deny), policy and rule combining algorithms (predefined values according to XACML standard);
- define a set of rules that is: 1) define a set of policy sets, policies, and policy combining algorithms; 2) define which policy set contains which policy/ies, and which policy contains which rule/s; 3) create XACML rules.

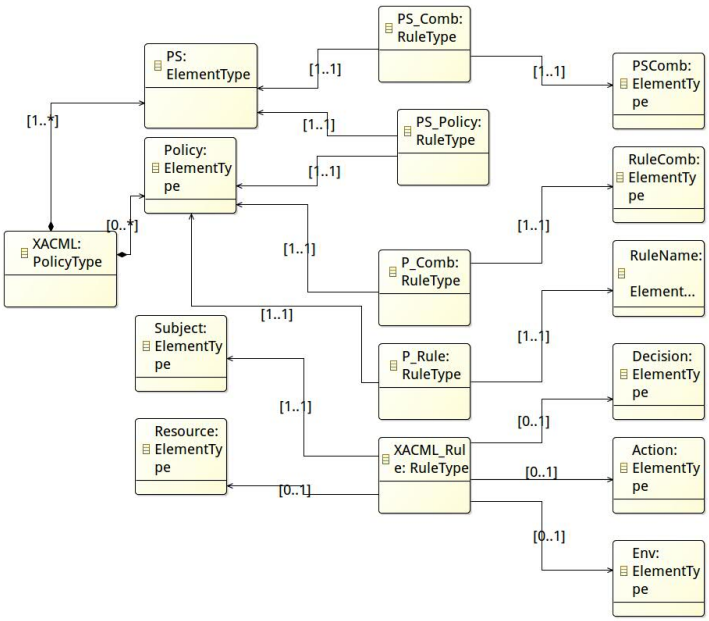

Figure 3: XACML Metamodel

The XACML metamodel is implemented as an Ecore model and comes with an associated editor allowing users to easily specify new XACML policy models. In Figure 4 an extract of the Library Management System (LMS) access control model (Pretschner et al., 2008) is presented. It instantiates the metamodel of Figure 3 specifying a rule allowing a student to borrow a book during the working days.

3.2 Model2XACML Transformation

This section describes the transformation algorithm implemented into the Ecore2XACML component of Figure 1 for deriving the XACML policy. This algorithm is implemented using the Kermeta language (Jézéquel et al., 2011).

First of all, the algorithm collects the metamodel rules having the type "PS_Policy". Based on these rules the algorithm creates the full list of policies contained in the global policy set. Then, the algorithm collects the combining algorithms for the policy set and its policies (using the rules having the rule type "PS_Comb" and "P_Comb" respectively). Once this step is performed, the algorithm generates the initial tags (policy set and policies). The XACML rules contained in each policy are built based on the metamodel rules having the type "P_Rule" (that links a rule with the policy containing it). Finally, for each rule, the subject, action, resource and environment are created using "XACML_Rule" and the corresponding rule is generated. As an example Table 1 reports the translation of PS_Comb and P_Comb into the XACML language.

It is important to note that some parts of the XACML code is generated by default. As shown in Table 1, the "xmlns" and "xmlns:xsi" have default values that are added automatically. Considering for instance the LMS access control model described in Section 3.1, the transformation algorithm derived the LMS XACML policy which contains a PolicySet, a Policy, 42 rules, 8 subjects, 3 resources, 10 actions and 3 environments [1].

3.3 Oracle

The oracle defined in our toolchain takes as input an XMI file that represents the Ecore model of the security policy and an XACML request. Subject, action, resource, environment attributes are extracted from the request using an attribute extraction tool. Based on the relevant attributes in the request and on the policy model, the oracle provides as an output a response for the input request. The algorithm that defines the oracle behavior is described in Algorithm 1. It takes as input an XACML request *req* and the ecore model of the XACML policy *mod* (line 1), extracts policy from the model (line 4) and creates a set of applicable rules (lines 5-9). A rule *R* is applicable to a request *req* if and only if the values of *Subject*, *Resource*,

[1] For space limitation we cannot provide here the source file.

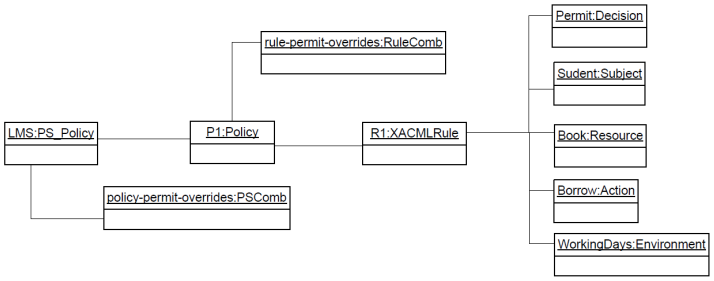

Figure 4: LMS model

Table 1: Model2XACML transformation example

Metamodel	**XACML**
PS_Comb: (LMSPolicySet, deny-overrides)	$< PolicySet xmlns = \text{“}urn : oasis : names : tc : xacml : 2.0 : policy : schema : os\text{”} xmlns : xsi = \text{“}http : //www.w3.org/2001/XMLSchema - instance\text{”} PolicySetId = \text{“}LMSPolicySet\text{”} PolicyCombiningAlgId = \text{“}urn : oasis : names : tc : xacml : 1.1 : policy - combining - algorithm : deny - overrides\text{”} >$
P_Comb: (BookPolicy, permit-overrrides)	$< PolicyPolicyId = \text{“}BookPolicy\text{”} RuleCombiningAlgId = \text{“}urn : oasis : names : tc : xacml : 1.1 : rule - combining - algorithm : permit - overrides\text{”} >$

Action and *Environment* in *R* are equal to those contained in the request *req*. If the set of applicable rules is empty the oracle decision is *NotApplicable* (lines 10-12), otherwise the Oracle takes a decision according to the *Rule Combining Algorithm* defined in the model. In particular, lines 13-21 determine the decision in case of the *rule-permit-overrides* algorithm; lines 22-30 define the oracle behavior in case of *rule-deny-overrides* algorithm and finally lines 31-34 calculate the decision in case of *rule-first-applicable* algorithm.

The following example illustrates the oracle reasoning: a request *R*, with the attributes values “Student”, “Borrow”, “Book”, “WorkingDays” related respectively to the *subject*, *action*, *resource* and *environment* attributes is evaluated against the following rule in the policy: i) **Name**: StudentRule; ii) **Type**: XACML RULE; iii) **Parameters**: Element R1, Element Permit, Element Student, Element Borrow, Element Book, Element WorkingsDays. The request *R* is evaluated in this case according to the decision element *Permit*.

Algorithm 1 Oracle Algorithm

```
1: Input: Request req, Model mod
2: Output: Decision ∈ {Permit, Deny, NotApplicable}
3: Set applicableRules := ∅
4: Policy P := mod.getPolicy()
5: for all Rule R ∈ P do
6:     if R.isApplicable(req) then
7:         applicableRules.add(R)
8:     end if
9: end for
10: if applicableRules = ∅ then
11:     Decision := NotApplicable
12: end if
13: if P.RuleCombiningAlgorithm = permit_overrides then
14:     if ∃ R ∈ applicableRules s.t. R.effect = Permit then
15:         Decision := Permit
16:     else
17:         if ∃ R ∈ applicableRules s.t. R.effect = Deny then
18:             Decision := Deny
19:         end if
20:     end if
21: end if
22: if P.RuleCombiningAlgorithm = deny_overrides then
23:     if ∃ R ∈ applicableRules s.t. R.effect = Deny then
24:         Decision := Deny
25:     else
26:         if ∃ R ∈ applicableRules s.t. R.effect = Permit then
27:             Decision := Permit
28:         end if
29:     end if
30: end if
31: if P.RuleCombiningAlgorithm = first_applicable then
32:     Rule R := applicableRules.getFirstRule()
33:     Decision := R.effect
34: end if
```

3.4 X-CREATE

The policy derived by the Ecore2XACML component is then used for the automatic generation of test cases (step 2 in Figure 1). In particular, among the available tools for test cases generation, in this toolchain we integrated X-CREATE (Bertolino et al., 2010; Bertolino et al., 2012a; Bertolino et al., 2013). Experimental results presented in (Bertolino et al., 2010; Bertolino et al., 2012a; Bertolino et al., 2013) showed that the fault detection effectiveness of X-CREATE test suites is similar or higher than that of comparable tools (like for instance Targen (Martin and Xie, 2006)).

In the toolchain implementation among the various X-CREATE proposals, we decided to use the

Multiple Combinatorial test strategy (Bertolino et al., 2013). As described in (Bertolino et al., 2013) for each policy this strategy generates four sets: the SubjectSet, ResourceSet, ActionSet, and EnvironmentSet, containing the values of elements and attributes of the subjects, resources, actions and environments respectively. The elements and attributes values in each set are then combined in order to obtain the subject, resource, action, environment entities. Specifically, a subject entity is defined as a combination of the values of elements and attributes of the SubjectSet set. Similarly the resource entity, the action entity and the environment entity represent combinations of the values of the elements and attributes of the ResourceSet, ActionSet, and EnvironmentSet respectively.

The XACML requests are then generated by combining the subject, resource, action and environment entities applying first a pair-wise, then a three-wise, and finally a four-wise combination, obtaining all possible combinations. In this process X-CREATE automatically eliminates the duplicated combinations. For more details we refer to (Bertolino et al., 2010; Bertolino et al., 2012a; Bertolino et al., 2013).

Considering the LMS XACML policy, derived as described in Section 3.2, 1800 XACML requests are generated. In the following an example of XACML request derived from LMS policy, expressed in JSON format.

```
{"XacmlRequest":{
  "Subject":["Student"],
  "Resource":["Book"],
  "Actions":["BorrowBook"],
  "Environment": ["WorkingDays","HolyDays"],
  }
}
```

This request denotes that a student wants to borrow a book during the working and holy days.

3.5 Test Executor

The Test Executor interacts directly with the SUT PDP either in case of model-driven testing, i.e. when the set of requests is derived by the X-CREATE component (steps 1, 2, 3 of Figure 1), or when a predefined test set is considered, (step 1' of Figure 1). In both cases the Test Executor takes as inputs the set of requests and the related policy and one by one it sends the requests to the SUT PDP together with the related XACML policy (steps 4 and 2' of Figure 1).

The Test Executor intercepts the obtained PDP response and derives the couple (request, response) useful to the Checker component for the final test verdict definition (step 6 and 3' of Figure 1).

3.6 Checker

The Checker defines a verdict for a given test case. As the Test Executor, it works either in case of model-driven testing, in this case the set of expected responses is generated by the Oracle component (step 7 of Figure 1), or when the predefined test set is considered, in this case the expected responses are taken from the database (step 5' of Figure 1). In both cases, the Checker takes as inputs the couple (request, response) obtained by the execution of the request on the SUT PDP (step 6 and 4' of Figure 1), and the couple (request, expected response) containing the correct PDP output. Thus for each request if the response is equal to the expected one then the Checker's verdict is Pass, otherwise is Fail (step 8 and 6' of Figure 1). The separation between the oracle and the checker in our model enables the checker to take as input any oracle response in the form of (request, expected response).

3.7 Test Archive

The last component of the proposed toolchain is the Test Archive which lets the possibility to test the SUT PDP by means of a predefined set of XACML policies. This kind of testing could be performed as an alternative (or in association) to the model-driven testing approach to improve the possibility to find faults in the SUT PDP.

The Test Archive is a repository of policies, each one with an associated set of requests and the corresponding expected responses. In particular the Test Archive integrates the XACML conformance test suite (OASIS Committee, 2005) and a set of real world policies (Bertolino et al., 2013). Details of this set of policies are summarized in Table 2, while the associated requests and expected responses provided by the X-CREATE tool are showed in Table 3.

As shown in Figure 1, the Test Archive interacts both with the Test Executor and the Checker. In particular the policies selected by the toolchain user are sent one by one to the Test executor with the associated set of requests (step 1' of Figure 1), then the associated couple (requests, expected responses) is sent to the Checker for computing the final verdict (step 5' of Figure 1).

4 EXPERIMENTAL RESULTS

In this section we describe the experimental results collected by a class of 14 master degree students in Computer Science, using the proposed toolchain

Table 2: Structure and description of XACML Policies

Policy	# PolicySet	# Policy	# Rule	# Cond	# SubAttDes	# ResAttDes	# ActAttDes	# EnvAttDes	# Funct
XACMLrules	3	7	17	10	2	35	10	0	4
create-document	0	1	3	2	1	2	1	0	3
delete-document	0	1	3	2	1	3	1	0	3
demo-11	0	1	3	2	2	5	1	0	5
demo-26	0	1	2	1	1	5	1	0	4
demo-5	0	1	3	2	2	5	2	0	4
read-document	0	1	4	3	2	4	1	0	3
read-information-unit	0	1	2	1	0	2	1	0	2
read-patient	0	1	4	3	2	4	1	0	3
student-application-1	0	1	2	0	7	2	2	0	2
student-application-2	0	1	2	0	19	2	2	0	2
university-admin-1	0	1	3	0	33	3	3	0	2
university-admin-2	0	1	3	0	32	3	3	0	2
university-admin-3	0	1	3	0	32	3	3	0	2

for testing a specific PDP implementation, the Sun PDP (Sun Microsystems, 2006).

Using test archive In this first experiment we asked the students to deploy the Sun PDP in the computers available in our students lab and to perform the testing of this engine using the predefined set of data available for the validation. Thus referring to Figure 1, the students followed the path labeled from 1' to 6'. Specifically, interacting with the toolchain each student selected either the execution of one or more elements from the set (policies, requests, exp_reponses) or to execute the conformance test suite. At this point the toochain starts the Sun PDP testing by: sending one by one the policy and the associated set of requests to the Test Executor, (step 1' of Figure 1); executing each request on the Sun PDP and collecting the associated response (step 2' and 3' of Figure 1); comparing the set of couples (requests, expected responses), (requests, responses) for computing the final verdict (step 4', 5' and 6' of Figure 1). In Table 3, for space limitation we report only the results collected for the real world policy set. In this case the execution of the requests on the Sun PDP does not evidence any inconsistency between the collected responses and the expected ones. This means that all the requests having as expected response Permit (third column) (Deny (fourth column), NotApp (fifth column), Indet (sixth column) respectively) in Table 3, have risen the same response when executed over the Sun PDP. Even if no failure have been discovered (mainly due to the overall good quality of the Sun PDP implementation), the experiment was a good benchmark for the proposed toolchain and the derived suggestions were useful for improving the usability of the user interface and the data collection of the toolchain.

Table 3: PDP Under Test results using *Multiple Combinatorial* strategy

		PDP Under Test			
Policy	**# Req**	# Permit	# Deny	# NotApp	# Indet
XACMLrules	17250	1035	16215	0	0
create-document	3	2	0	1	0
delete-document	7	3	0	3	1
demo-11	98	28	28	42	0
demo-26	7	1	3	3	0
demo-5	861	394	98	369	0
read-document	42	16	0	21	5
read-information-unit	3	1	0	1	1
read-patient	42	13	0	21	8
student-application-1	7	3	4	0	0
student-application-2	63	4	59	0	0
university-admin-1	1159	22	1137	0	0
university-admin-2	1159	176	983	0	0
university-admin-3	13908	88	4544	9276	0

Using Model-based Approach In this second experiment we asked the students to perform the testing of the Sun PDP by using the Library Management System (LMS) access control model (Pretschner et al., 2008) (see Section 3.1). Referring to Figure 1, the students followed the path labeled from 1 to 8 of the proposed toolchain. In particular, a set of 1800 XACML requests has been derived by using the tool X-CREATE from the LMS policy. The result of the requests execution was: 18 Deny, 24 Permit and 1758 NotApplicable. By using the Checker component, the students compared each couple (request, response) with the corresponding couple of (request, expected response). As in the previous experiment, all the verdicts got the *Pass* value and therefore no failure has been discovered.

5 RELATED WORK

This section addresses several existing initiatives in applying model-based approaches for security testing. A standard language for designing test models

is UML Testing Profile (UML-TP) (OMG, 2004) that represents a lightweight extension of UML with specific concepts (stereotypes) to support the design of testing artifacts. In (Lodderstedt et al., 2002), UML models have been used to generate security infrastructures through SecureUML language which is based on Role based Access Control Model (RBAC).

Model-based testing has been investigated for the XACML policy testing (Le Traon et al., 2007; Pretschner et al., 2008). The approaches proposed in (Le Traon et al., 2007; Pretschner et al., 2008) are based on the representation of policy implied behavior by means of models. Differently from these works model based testing in this paper addresses testing of the PDP engine. The authors of (Li et al., 2008) address testing of the XACML PDP by running different XACML implementations for the same test inputs and detecting not correctly implemented XACML functionalities when different outputs are observed. Differently from our proposal, this approach randomly generates requests for a given policy and requires more PDP implementations for providing an oracle facility by means of a voting mechanism. Our focus is to provide an integrated toolchain including test case generation as well as policy and oracle specification for the PDP testing. A different solution in the context of usage control is presented in (Bertolino et al., 2012b) where the authors provide a fault model and a test strategy able to highlight the problems, vulnerabilities and faults that could occur during the PDP implementation. This solution is specifically designed for PolPA language, then it cannot be used for XACML PDP testing.

6 DISCUSSION AND CONCLUSION

This paper presented a toolchain for testing an XACML PDP. The main facilities of the proposed toolchain are: a model-based specification of the XACML policy, a test suite generation and execution engine, and the oracle definition. Two different experiments confirmed the effectiveness of the toolchain for testing a real PDP engine (Sun PDP). Concerning the validity of the experiments, i.e. the amount of confidence on the reported results, an important key factor is the employed test set: we used X-CREATE for deriving test suites, but it is likely that other test sets may produce different results. In this paper, we have applied our model based approach for deriving LMS policy which includes 42 rules. Therefore, bigger policies need to be considered to guarantee the scalability of the proposed approach. For future work, we plan to extend the Test Archive including XACML policies having a large number of rules, consider different test suites and improve the toolchain to support the XACML 3.0.

ACKNOWLEDGEMENTS

The authors would like to thank Antonia Bertolino and Yves Le Traon for their suggestions and useful discussions.

REFERENCES

Bertolino, A., Daoudagh, S., Lonetti, F., and Marchetti, E. (2012a). Automatic XACML requests generation for policy testing. In *Proc. of SECTEST*, pages 842–849.

Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., and Mori, P. (2012b). Testing of PolPA Authorization Systems. In *Proc. of AST*, pages 8–14.

Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., and Schilders, L. (2013). Automated testing of extensible access control markup language-based access control systems. *IET Software*, 7(4):203–212.

Bertolino, A., Lonetti, F., and Marchetti, E. (2010). Systematic XACML Request Generation for Testing Purposes. In *Proc. of EUROMICRO (SEAA)*, pages 3–11.

Jézéquel, J.-M., Barais, O., and Fleurey, F. (2011). Model driven language engineering with kermeta. In *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer.

Le Traon, Y., Mouelhi, T., and Baudry, B. (2007). Testing security policies: going beyond functional testing. In *Proc. of ISSRE*, pages 93–102.

Li, N., Hwang, J., and Xie, T. (2008). Multiple-implementation testing for XACML implementations. In *Proc. of TAV-WEB*, pages 27–33.

Lodderstedt, T., Basin, D., and Doser, J. (2002). SecureUML: A UML-based modeling language for model-driven security. In *The Unified Modeling Language*, pages 426–441. Springer.

Martin, E. and Xie, T. (2006). Automated Test Generation for Access Control Policies. In *Supplemental Proc. of ISSRE*.

OASIS (1 Feb 2005). eXtensible Access Control Markup Language (XACML) Version 2.0.

OASIS Committee (2005). XACML Version 2.0 Conformance Tests.

OMG (2004). UML 2.0 Testing Profile Specification. `http://utp.omg.org/`.

Pretschner, A., Mouelhi, T., and Traon, Y. L. (2008). Model-based tests for access control policies. In *Proc. of ICST*, pages 338–347.

Sun Microsystems (2006). Sun's XACML Implementation.