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

Testing of PolPA-based Usage Control Systems

**Antonia Bertolino · Said Daoudagh ·
Francesca Lonetti · Eda Marchetti ·
Fabio Martinelli · Paolo Mori**

Received: date / Accepted: date

Abstract The implementation of an authorization system is a critical and error-prone activity that requires a careful verification and testing process. As a matter of fact, errors in the authorization system code could grant accesses that should instead be denied, thus jeopardizing the security of the protected system. In this paper, we address the testing of the implementation of the Policy Decision Point (PDP) within the PolPA authorization system that enables history-based and usage-based control of accesses. Accordingly, we propose two testing strategies specifically conceived for validating the history-based access control and the usage control functionalities of the PolPA PDP. The former is based on a fault model able to highlight the problems and vulnerabilities that could occur during the PDP implementation. The latter combines the standard technique for conditions coverage with a methodology for simulating the continuous control of the PDP during the runtime execution. Both strategies are implemented within a testing framework supporting the automatic generation and execution of security test suites. Results produced by the application of this testing framework to a real case study are presented.

Keywords Authorization Systems · PolPA Language · Usage Control ·
History-based Access Control · Testing · Test Cases Generation

---

A. Bertolino and S. Daoudagh and F. Lonetti and E. Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
via G. Moruzzi, 1 - 56124 Pisa, Italy
E-mail: {firstname.lastname}@isti.cnr.it

F. Martinelli and P. Mori
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
via G. Moruzzi 1 - 56124 Pisa, Italy
E-mail: {firstname.lastname}@iit.cnr.it

1 Introduction

Security is a crucial aspect of modern pervasive ICT systems. The managed resources, e.g., data, machines or services, can be sensitive and valuable, and hence proper means must be put in place to protect them against unauthorized, malicious, improper or erroneous usage. To this purpose, authorization systems enable the specification of security policies that rule various protection aspects such as: the level of confidentiality of data, the procedures for managing data and resources, the classification of resources into category sets with different security requirements. Several authorization system models have been defined in scientific literature, and some implementations of these systems, both academic and commercial, are currently available, such as the SUN's XACML engine[1] or the WSO2's Balana one[2].

Among the several existing systems, in this paper we focus on the PolPA authorization system, which has been defined in [14]. PolPA exploits a process algebra-based security policy language that supports history-based control and Usage Control (UCON). Hence, the PolPA authorization system can control the sequence of security relevant actions performed by the user so to prevent the execution of an action when a policy is not satisfied. Moreover, it allows policy makers to express conditions that control the usage of resources, i.e., these conditions are continuously evaluated all along the access time, so that an access right can be revoked as soon as they are violated. The PolPA authorization system has been successfully adopted in several scenarios, such as the Grid [14], mobile devices [7], and Next Generation Networks [9].

From an architectural point of view (as further described in the next section), the PolPA authorization system includes several components, such as: the Policy Enforcement Point (PEP), which enforces the policy decisions, the Policy Decision Point (PDP), which performs the decision process to determine whether (according to the defined security policies and to the current values of the attributes of the users, resources and environment) an access should be granted or denied, the Policy Information Point (PIP), which is in charge of retrieving the attributes of users, resources, and environment, and the Policy Administration Point (PAP), which supports the editing and the storing of the different policies.

All the above components are critical from a security point of view and would require careful verification and testing. In particular, this paper focuses on testing the PDP. In traditional access control systems, the PDP implementation is already a difficult and crucial activity for developers since it is always under the threat of unintended points of vulnerability, or of missing or misrepresented access policies. The PolPA PDP is even more complex because, besides the support for traditional access control, it also includes the history-based and the usage control functionalities. Hence, its implementation is more error-prone and subject to vulnerabilities.

[1] http://sunxacml.sourceforge.net/

[2] http://xacmlinfo.com/category/balana

To prevent potential flaws, a rigorous and accurate verification and testing process must be adopted. Available solutions for testing PDPs are usually focused on standard access control models and more popular access control languages, such as OrBAC, RBAC or XACML, and cannot be easily transferred into the UCON environment. To the best of our knowledge, there are no suitable testing technologies for validating the PDP implementation of continuous control of the user accesses. Hence, in this paper we propose a testing framework customized for the PolPA language, specifically designed to deal with history-based UCON security policies.

Thus, focusing on the PDP that evaluates PolPA security policies described in [14], we present:

- a fault model that highlights the problems, vulnerabilities and faults that could occur when the PDP evaluates the access history;
- a history-based test case generator for the automatic generation of a test suite that covers the fault model. The test case generation relies on an original domain specific testing methodology proposed in this paper;
- a test generator for continuous policy enforcement specifically conceived for addressing the dynamic runtime behavior of the PDP required for prompt access revocation;
- an automated oracle able to determine whether a test has passed or failed for the continuous policy evaluation.

This paper refines and expands our preliminary proposal for a PolPA testing framework presented in [4], where the tested PDP only supports history-based security policies. Here, instead, we propose the complete process covering both history-based and continuous policy enforcement testing. The process has been automated and applied to an illustrative case study.

The rest of this paper is structured as follows. Section 2 briefly introduces the PolPA authorization systems. Section 3 motivates the proposed approach. In Section 4 we present the proposed testing framework and then detail its components in Sections 5, 6, 7, 8, the fault model, the test case generator, the test driver and the test oracle respectively. In Section 9 we provide results of the application of the proposed testing framework to a real PDP implementation and we discuss about the test cases effectiveness. Section 10 puts our work in context of related work and Section 11 concludes the paper.

2 PolPA Authorization System

PolPA is a process algebra-based language for specifying history-based security policies according to the Usage Control (UCON) model [14]. The UCON model [20, 22] is an extension of the traditional access control models that, besides authorizations, introduces new factors in the decision process, namely: obligations, conditions, and *mutable* attributes. Mutable attributes are paired with subjects and objects, and their values are updated as a consequence of the decision process. Hence, the attributes that have been evaluated by the

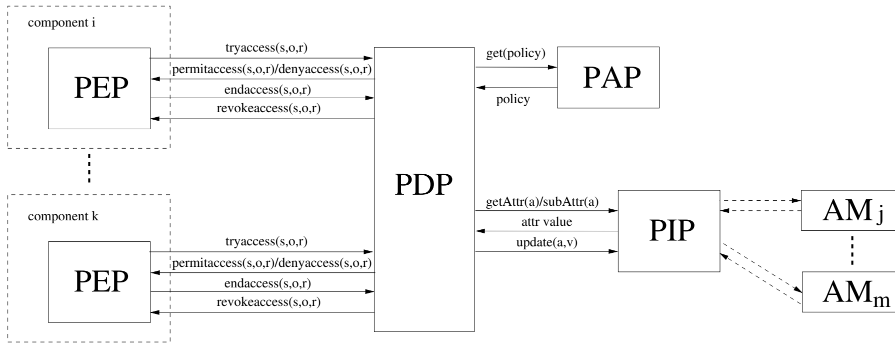

Fig. 1 Authorization System Architecture

security policy to initially grant an access to a resource could change their values while the access is in progress in such a way that the access right does not hold anymore. In this case, the access should be interrupted to preserve the system security. For this reason, UCON policies specify whether a decision factor must be evaluated before and/or during the usage of the resource (*continuous policy enforcement*).

The PolPA security policy exploits some composition operators to define the allowed behavior, i.e., the order in which security relevant actions can be performed. Roughly speaking, these operators allow to represent a sequence of actions, the alternative choice among a set of actions, the parallel execution of a set of actions, and the iterative or replicated execution of actions. For example, two or more actions must be executed in the same order they appear in the policy if they are composed through a *seq* operator. Two or more actions can be executed alternatively or in parallel if they are related to an *or* or *par* composition operator, respectively. Moreover, PolPA allows to specify some predicates involving action's parameters and attributes of the user, of the resource and of the environment that need to be satisfied in order to proceed with the action execution. For a more detailed description of PolPA language refer to [14].

The architecture of the authorization system that enforces PolPA policies, as most common authorization systems, is based on a Policy Enforcement Point (PEP), a Policy Decision Point (PDP), a Policy Information Point (PIP) and a Policy Administration Point (PAP), as shown in Figure 1.

The PEPs should be integrated in the software components that implement security relevant actions to intercept their execution. When the user tries to execute a security relevant action, the PEP sends to the PDP the $tryaccess(s, o, r)$ command, where s is a string that represents the name of the user, e.g., a X.509 Distinguished name such as "C=IT, L=Pisa, O=CNR-IIT, CN=paolo mori", o is the string that represents the name of the resource, e.g., a web service such as "http://node3.iit.cnr.it/axis2/services/MyTestService" and r represents the actions along with their parameters, e.g., the operations exposed in the web service interface.

For the sake of simplicity, in the following we will use some short placeholders to represent users' and resources' names, such as U and V or R1 and R2, respectively, and to represent actions, such as A(x_1,x_2), B(x_3), C(x_4,x_5) and D(x_6).

The PEP allows the execution of the action only after a positive response from the PDP, represented by the $permitaccess(s, o, r)$ command. Once an action has been permitted, the PEP should be able to detect when it terminates to issue the $endaccess(s, o, r)$ command to the PDP. The PDP is the component of the architecture that performs the usage decision process. The PDP, at first, gets the security policy from the repository managed by the PAP, and builds its internal data structures for the policy representation. When the PDP receives the $tryaccess(s, o, r)$ command from a PEP, it checks the requested action against the security policy. To this aim, the PDP collects fresh attribute values from the PIP. Consequently, either the $permitaccess(s, o, r)$ command is sent to the PEP, that executes the action, or the PDP returns $denyaccess(s, o, r)$ to the PEP, that enforces it by skipping the execution of the access. Since it must keep track of the actions that are in progress, the PDP is also invoked by the PEP when an action that was in progress terminates, with the $endaccess(s, o, r)$ command. In fact, the PDP is always active because, if required by the policy, the PDP continuously evaluates a set of given authorizations, conditions and obligations while an action is in progress. It could request the PEP to terminate this access through the $revokeaccess(s, o, r)$ command. Again, the PDP exploits a subscription mechanism to collect updated attribute values from the PIP. This is a main novelty of the UCON model with respect to prior access control work, where the PDP is usually only passive, i.e., it simply answers to the access requests received from PEPs. To enforce the $revokeaccess(s, o, r)$ command, the PEP should be able to interrupt an action that is in progress.

Table 1 shows a very simple example of PolPA policy. In this example we have two resources, represented by R1 and R2, and four actions, A, B, C, and D.

The policy defines a sequence of actions where action A on resource R1 must be firstly executed, then either action B or C can be executed on resource R2, and finally action D can be executed on resource R1. Lines 1-4 of the policy regulate the execution of action A on resource R1, imposing a control on the value of parameter x_1 (line 2), and requiring that action A terminates (line 3) before allowing the execution of the other actions. Line 1 does not impose any constraint on the identity of the user that executes the action, which is represented by the variable *user_id*. Lines 5-21 allow the alternative execution of action B or action C on resource R2, due to the *or* composition operator in line 12. Action B can be executed only if the value of the attribute role related to the user that required the execution of the action is equal to "admin", as stated in line 6 of the policy. Instead, for executing action C, line 14 imposes a constraint on the parameter x_4 of the action itself. In both cases, the *endaccess* commands (lines 7 and 15) are combined through an *or* operator with the *revokeaccess* commands (lines 9 and 18), in order

Table 1 Example of PolPA security policy

```
( tryaccess(user_id, R1, A(x1,x2)).                                             1
 [(sequal(x1,"val1"))].permitaccess(user_id, R1, A(x1,x2)).                     2
 endaccess(user_id, R1, A(x1,x2))                                               3
).                                                                              4
( ( ( tryaccess(user_id, R2, B(x3)).                                            5
    [(sequal(attr(role,user_id),"admin"))].permitaccess(user_id, R2, B(x3)).    6
    ( endaccess(user_id, R2, B(x3))                                             7
     or                                                                         8
      ([(lessthan(attr(reputation,user_id),85))].revokeaccess(user_id, R2, B(x3)))   9
     )                                                                          10
   )                                                                            11
   or                                                                           12
   ( tryaccess(user_id, R2, C(x4,x5)).                                          13
    [(iequal((x4,"val2"))].permitaccess(user_id, R2, C(x4,x5)).                 14
    ( endaccess(user_id, R2, C(x4,x5))                                          15
     or                                                                         16
      ([((lessthan(attr(reputation,user_id),75))and(morethan(attr(workload,R2),70)))].  17
      revokeaccess(user_id, R2, C(x4,x5)))                                      18
    )                                                                           19
   )                                                                            20
  ).                                                                            21
 ( tryaccess(user_id, R1, D(x6)).                                               22
  [(iequal(x6,"val3"))].permitaccess(user_id, R1, A(x6)).                       23
  endaccess(user_id, R1, D(x6))                                                 24
 )                                                                              25
)                                                                               26
```

to implement continuous control. Hence, the policy states that either these actions finish normally, and the *endaccess* commands are issued, or they are interrupted by the authorization system because the predicates that precede the *revokeaccess* commands (i.e., the ones in line 9 and line 17) are verified. In particular, the predicate in line 17 exploits the user attribute reputation, which is an integer that ranges from 0 to 100, and the resource attribute workload, which is an integer that ranges from 0 to 100 as well. When the execution of the action C is in progress on resource R2, it could be interrupted by the *revokeaccess* command in line 18 of the policy because the value of the user reputation is too low (below 75) and the workload of the resource R2 is too high (above 70). Finally, lines 22-25 allow the execution of action D on resource R1, again imposing a constraint on the value of action's parameter x_6.

3 Motivations and Key Ideas

The PDP is a key component of the PolPA authorization system since it is responsible for the decision process, i.e., it determines whether an access can be granted or not. Any error in the PDP implementation could alter the PDP decision, such as authorizing accesses that should be forbidden, or not interrupting accesses whose execution rights are no longer valid, as well as denying accesses that should instead be authorized, or erroneously revoking an ongoing access. Hence, any error could have a serious impact on the decision process and consequently on the *integrity* and on the *availability* of the protected system.

We focus on the PDP implementation, because the other components of the authorization system (as shown in the previous section) are typically dependent on the specific scenario where the authorization system has been embedded. On the contrary, the PDP is scenario independent, because it interacts only with the other components of the authorization system, i.e., it does not interact directly with entities that are specific to a scenario.

An important aspect in test cases generation is clearly identifying the target of the test cases, i.e., the problems and the weaknesses that the test cases should detect. Considering the nature of the authorization system, test cases generation can be focused either on the history-based control or on the continuous policy enforcement. In the next subsections we motivate the testing strategies addressing two kinds of faults. Specifically, we distinguish test cases designed to detect faults related to the passive behavior of the PDP (see Subsection 3.1), and those focusing on the problems related to the mutability of attributes during the execution of accesses that require an active PDP (see Subsection 3.2).

3.1 Motivations for history-based testing

The testing of the history-based access control functionality of the PDP relies on a sort of fault model, which can be exploited for test case derivation. The generated test cases are designed and customized to cover the fault model and then to detect the system faults with the advantages of reducing the number of executed test cases and the test effort. In particular, the testing strategy we consider includes three different steps:

- identify the main problems of the PDP implementation concerning the history-based access control functionality, i.e. the fault model;
- derive from a given policy (gold policy), a set of faulty policies according to the defined classes of problems;
- generate the test cases able to detect the seeded faults, i.e. select the access requests able to evidence a misinterpretation of access rules.

Many common approaches for test cases derivation work in the other way round: first a test strategy is defined considering specific testing aspects, usually not related to a specific fault model; then test cases are generated and

executed on a system under test; only at the end of the testing phase, either the coverage of a test criterion is considered or the fault detection effectiveness of the executed test suite is measured, usually by the application of mutation testing. According to this process, a fault model is implicitly conceived just for the definition of mutant operators useful for the evaluation of the test suite effectiveness, but usually not directly defined for test cases derivation purposes. We refer to [10] for a recent and extensive survey of mutation testing approaches and their application.

The peculiarity of the proposed process is that the generated test cases assure the coverage of the fault model and the ability to identify the main problems of the authorization system. However, from a practical point of view, the testing process we propose cannot be easily adopted without automated support. Thus the focus of this paper is a framework that automatically generates the test cases, i.e. authorization requests, starting from a defined fault model.

For the implementation of our testing framework we had to solve two main issues: *i)* how to automatically derive the set of faulty policies reflecting the problems defined in the fault model; *ii)* how to automatically derive a set of test cases able to detect the faults.

Each element of the policy (such as composition operators, predicates, control commands, and so on, see Section 2) is directly related to the PDP module that implements it, which is executed when security actions are evaluated against the relevant policy. Every modification of the PolPA policy represents in a natural way a potential failure of the behavior of the PDP component, as it could result in a verdict (allow, deny access) different from the correct one. Thus our idea is to define the fault model in terms of changes applicable to the PolPA policy, so that the modified policy versions can be used for testing purposes.

The other key idea derives from the analysis of the PolPA policies. In general a PolPA policy can be seen as the behavioral authorization scheme for the various actors accessing the resources of the system. It contains the composing elements, their relations and the algorithms that have to be applied to evaluate an access, thus the policy defines the input domain and its evaluation. Our idea is that of parsing a PolPA policy in agreement to its rules, collecting the information and systematically deriving the combination of elements and values representing the different user's access modes, i.e test cases. Thus, given a PolPA policy, the testing framework first generates the faulty policies according to a PolPA specific fault model; then, parsing the policy and its mutated versions, it generates the tests set able to cover the fault model so to exercise some specific aspects of the policy interpretation and to highlight the misbehavior of the PDP implementation.

3.2 Motivations for continuous policy enforcement testing

In order to test the continuous policy enforcement functionality of the PDP, we need to focus on the mutability of attributes, and specifically on the ability of the PDP to promptly react to any attribute change during the execution of an access. The PIP interacts with the PDP for managing the attribute values. Notice that the PDP can provide different replies to distinct evaluations of the same authorization request, just depending on the values of the attributes, which dynamically change over time. The influence of the external environment, expressed in terms of mutability of attributes, has therefore a key role in the continuous policy enforcement. The PDP misinterpretation of these attributes changes could have critical consequences on the overall policy enforcement and on the system security. Possible erroneous situations could be:

- Before the evaluation of an authorization request, the PDP asks the PIP for the updated values of the attributes required to perform the decision process. However, an error occurs in the attribute value exchange phase, so the PDP does not authorize the access to the resource. For instance, the PDP used a corrupted attribute name in the attribute value request, or it performed an error extracting the new attribute value from the response received from the PIP.
- During the execution of an authorized access, the PIP notifies the PDP of the change of some attributes that, according to the policy, should cause the revocation of this access. However, the PDP does not manage the PIP message correctly, so the access is not interrupted and the user keeps on utilizing the resource although the corresponding right is not valid any more. For instance, the PDP performed an error extracting the new attribute value from the message received from the PIP, or did not keep trace of the ongoing accesses correctly.

Both above examples are extremely critical from a security point of view so they should be strongly avoided. Moreover, the examples show that only the prompt acquisition of the updated values of the attributes can assure correct enforcement of the security policy by the PDP. Since the PDP malfunctioning is mainly related to faults in evaluating predicates that involve corrupted attribute values, we propose a testing strategy based on predicate coverage criteria. The strategy should derive a set of test cases suitable for covering all cases in which the attributes values could be modified during the access life cycle, i.e., from the access request to the access termination. Specifically, a test case should stimulate the PIP for changing the value of one (or even more) attribute so to enforce the PIP and PDP to interact for exchanging the new attribute value, re-evaluating the PolPA policy predicates, and possibly enforcing the revocation of one (or more) access(es). In PolPA, a predicate is a boolean expression with one or more conditions logically connected, where each condition is, in turn, a boolean expression involving the values of mutable attributes related to the user, to the resource or to the environment. For

example, the predicate in line 17 of the policy presented in Table 1 includes two conditions. The first of the two conditions involves the value of the reputation, which is a mutable attribute of the user, while the second condition involves the value of the workload, which is a mutable attribute paired with the resource.

The proposed test strategy combines the standard technique for conditions coverage with a methodology for simulating all the possible ways in which attributes can change during the access life cycle. Therefore a same predicate can have different PDP evaluations depending on when PIP component notifies the PDP of the attributes change. The analysis of PolPA policy execution allows us to define exactly the moments in which a new attribute value can be sent by the PIP to the PDP during the evaluation of an access request. For aim of simplicity, avoiding the complex details about the exact access request execution, these moments are:

- *before the access request evaluation*: The attribute value changes just before the evaluation of the access request against the security policy. In this case, the new value is exploited to evaluate such request. If the requested access is authorized, but the evaluation of a predicate associated to a revocation command (such as the one in line 17 of the policy presented in Table 1) returns true, the PDP must revoke the previously authorized access.
- *during the access execution*: The access request is evaluated against the security policy and the access is granted, and the PDP subscribes to receive attribute updates from the PIP. If, during the execution of the access, the PIP notifies the PDP of the attribute changes and the evaluation of the predicate associated to a revocation command with the new value results to be true, the PDP must revoke the user access.
- *after the access execution*: The access request is evaluated against the security policy and the access is granted. The access is executed and terminates normally. Only after the access termination the PIP notifies the PDP of the attribute changes. In this case, independently from the evaluation of the predicate associated to a revocation command, the PDP goes ahead with the evaluation of the next request.

The proposed testing strategy derives test cases aimed at covering all the value combinations of the conditions associated to a revocation command as well as the simulation of the runtime interaction between PIP and PDP. More details about the strategy are in Section 6

4 Testing Framework

With reference to the architecture in Figure 1, this work is aimed at testing the PDP implementation, specifically at verifying that the PDP actually enforces the security policy that it gets as input. To do this, the testing framework needs to emulate the behaviour of a possible PEP by issuing the *tryaccess* and *endaccess* commands to the PDP and of a possible PIP component by changing

on the fly the values of the attributes. The subject of the testing is the PDP, while we assume that the input policy is correct, i.e., that it does not contain errors or conflicting rules and properly expresses the security requirements of the resource owner who wrote it.

The derivation of adequate test cases is an expensive task in software testing, especially for software systems whose inputs have a complex structure such as the requests entered to the PolPA authorization system. These requests consist of a variable number of commands with their parameters interleaved according to a set of rules specified in the policy. Manually preparing such test cases is expensive, difficult and error-prone. In traditional access control systems, combinatorial approaches for derivation of test cases have been proposed [5, 2], which generate all different combinations of commands and parameters values. Such approaches are not suitable for application to PolPA policies, since they do not specifically address the order in which the request commands are submitted to the PDP.

We propose in this paper a testing framework able to exercise the functionalities of the PDP involved both in the history-based access control and in the continuous policy enforcement. Specifically, for the former case the generation technique involves the design of a fault model along with the corresponding set of mutation operators, and their application for generating the test cases (see further details in Subsection 6.1). In the latter, we take inspiration from the traditional condition coverage testing strategy [15], which requires that all possible outcomes from every boolean decision in a program are exercised at least once. We adapt it to simulate the runtime attributes modification and the sequence of access requests (see further details in Subsection 6.2).

In this section, we describe the testing framework implementing the above strategies, which consists of the following components (see Figure 2):

Fault Model Manager (FMM). This component is involved in the testing of history-based access control. It manages a predefined collection of possible types of faults that can occur during the evaluation of a PolPA policy due to incorrect implementation of commands, guards or compositional operators. See Section 5 for more details.

Policy Test Set Manager (PTSM). This component collects the set of PolPA policies useful for testing purposes. The polices can be given as an input by the user or taken from a predefined collection memorized into an internal dataset. In this last case a set of PolPA policies are specifically conceived to highlight the functionality of the PDP and exercise the different available features. The PTSM is also in charge of the interaction with the PAP for the correct configuration of the PDP with the policy that is used for testing purposes.

Faulty Policies Generator (FPG). This component is involved in the testing of history-based access control. It takes as input a policy and the fault model and derives a set of faulty policies by seeding the faults defined in the fault model into the policy itself. Each of the faulty policies represents a faulty implementation of the PDP.

Test Cases Generator (TCG). This component derives the set of test cases for history based and continuous policy enforcement testing. In the former case, for each of the available policies (i.e. the policy and its faulty versions) TCG automatically derives the test cases in terms of (sequence of) access requests. Again, in case of continuous testing, TCG derives the sequence of access requests; in addition it interleaves them with specific commands called PIPCommand. See Section 6 for more details.

Mutable Attribute Updater (MAU). This component manages the attributes of the user, resource and environment. In particular, it simulates the PIP behavior by interacting with the PDP both for receiving requests of attributes subscription and for updating the values of the attributes involved in a predicate of a $revokeaccess(s, o, r)$ command during the test case execution. See Section 7 for more details.

Test Driver (TD). This component coordinates test execution. By collaborating with the TCG, it selects one by one the available test cases and, by simulating the PEP behavior, transforms each test case in the opportune sequence of $tryaccess(s, o, r)$, and $endaccess(s, o, r)$ commands. Moreover, during the execution of a test case, TD interacts with the MAU, by means of a PIPCommand so to set the values of mutable attributes specified in the test case. See Section 7 for more details.

Test Oracle (TO). This component is responsible for the collection of the PDP responses (*permitaccess(s,o,r)*, *revokeaccess(s,o,r)* or *denyaccess(s,o,r)*) caused by a test execution. TO also compares the obtained results with the correct authorization replies associated to each of the generated (set of) test cases. For this TO interacts with TCG for having the policy and the derived set of test cases. It is important to specify that in the current implementation only the functionality for the evaluation of the continuous policy enforcement test results is automated. We are evaluating and defining possible solutions for the analysis of history-based test results. See Section 8 for more details.

In the next sections we detail the defined Fault Model, the Test Case Generator, the Test Driver and the Test Oracle components.

5 Fault Model

The fault model is specifically conceived for PolPA language. The approach is inspired by mutation testing techniques [10]. Here mutation operators are not used for test adequacy measurement as is more commonly the case, but for test cases generation. Existing mutation operators are adapted in order to describe modification rules that introduce faults into PolPA policies, so that each faulty policy represents a syntactic fault that could occur during the PDP implementation.

The considered mutation operator classes focus on faults concerning the policy behavior, i.e. the execution of the different commands or their order, and faults in the evaluation of satisfiability of the parameters of each command, i.e. the string parameters and integer parameters. Note that the faults introduced

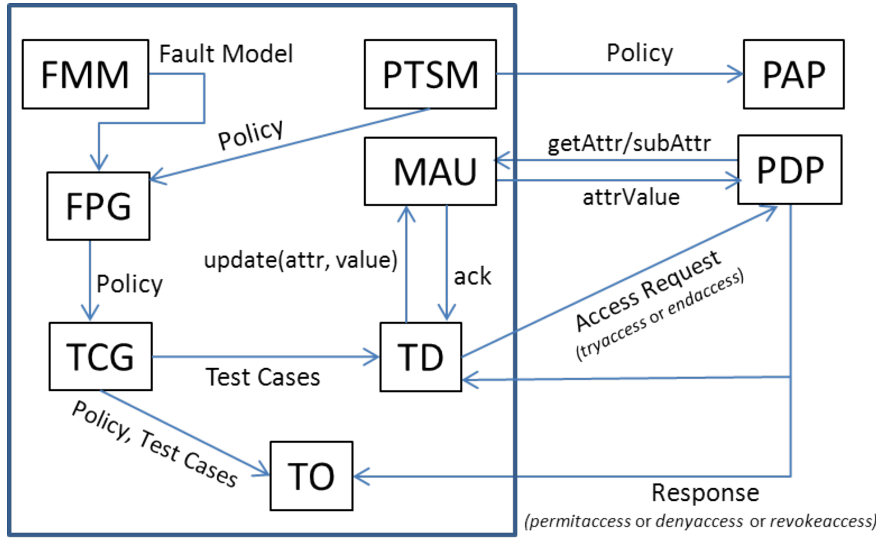

Fig. 2 Testing Framework

exercise only static behavior of the PDP and do not consider the continuous policy enforcement due to the runtime mutability of the attributes. As said, this specific task is addressed into the continuous policy enforcement testing activity as described in Section 6.2.

In the following a detailed description of the classes is given:

Change Composition Operator (CCO). This class implements a violation of the order of execution of the commands sent by the PEP ($tryaccess(s, o, r)$ and $endaccess(s, o, r)$); it is implemented by changing the composition operator. Specifically, let CO be the bag of different occurrences of composition operators ($\cdot$, or, par)[3] included in the policy: $\forall c_i \in$ CO, change c_i with each c_j such that $c_j \in$ CO $\setminus\{c_i\}$. The number of mutants derived by this class is equal to the number of composition operators that are in CO times 2. Note that in case of the history-based testing, this class is not applied to the *or* operator involving a $revokeaccess(s, o, r)$ and a $endaccess(s, o, r)$ command, because the derived mutants would be semantically equivalent.

Change Command (CC). This class implements faults in the execution of a command sent by the PEP; it is implemented by changing the command. Specifically, let C be the bag of different occurrences of commands ($tryaccess(s, o, r)$ and $endaccess(s, o, r)$) included in the policy: $\forall c_i(s_i, o_i, r_i) \in$ C, change $c_i(s_i, o_i, r_i)$ with each $c_j(s_j, o_j, r_j)$ such that $c_j(s_j, o_j, r_j) \in$ C $\setminus \{c_i(s_i,\ o_i,\ r_i)\}$ and $s_i = s_j$, $o_i = o_j$, $r_i = r_j$. The number of mutants derived by this class is equal to n where n is the cardinality of C.

Change Guard String Operator (CGSO). This class implements a wrong management of the values of string parameters; it is implemented by

[3] Where the dot represents the *seq* composition operator.

changing the arithmetic operators involving string parameters. Specifically, let S be the bag of different occurrences of PolPA arithmetic operators involving string parameters (sequal, startwith, scontains) included in the policy: $\forall s_i \in$ S, change s_i with each s_j such that $s_j \in$ S $\setminus\{s_i\}$. The number of mutants derived by this class is equal to the number of arithmetic operators involving string parameters that are in the policy times 2.

Change Guard Integer Operator (CGIO). This class implements a wrong management of the values of integer parameters; it is implemented by changing the arithmetic operators involving integer parameters. Specifically, let I be the bag of different occurrences of PolPA arithmetic operators involving integer parameters (iequal, morethan, lessthan) included in the policy: $\forall i_i \in$ I, change i_i with each i_j such that $i_j \in$ I $\setminus\{i_i\}$. The number of mutants derived by this class is equal to the number of arithmetic operators involving integer parameters that are in the policy times 2.

Change Logical Operator (CLO). This class implements a wrong management of logical operators contained in the PolPA predicate; it is implemented by changing a logical operator with another PolPA logical operator. Specifically, let L be the bag of different occurrences of PolPA logical operators (and, or) included in the policy: $\forall i_i \in$ L, change i_i with each i_j such that $i_j \in$ L $\setminus\{i_i\}$. The number of mutants derived by this class is equal to the number of logical operators that are in the policy.

The fault model is used by the Faulty Policies Generator for deriving faulty policies. As an example, by applying the mentioned mutation operator classes to the policy of Table 1, we derive thirty-seven faulty policies. Specifically, applying the CCO class we derive fourteen faulty policies because the policy includes six *seq* and one *or* composition operators related to the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ commands. Applying the CC class we obtain eight faulty policies, because there are four actions (A, B, C, D), each one having a $tryaccess(s, o, r)$ and a $endaccess(s, o, r)$ command. Finally, applying the CGSO class to the two arithmetic operators (line 2 and 6) that are in the policy we obtain four faulty policies, applying the CGIO class to the five integer predicates (line 9, 14, 17 and 23) we derive ten more faulty policies whereas applying the CLO class to the logical operator of line 17 we obtain one faulty policy.

6 Test Cases Generator

In this section we present the Test Case Generator component that manages the history-based and the continuous policy enforcement test case generation. The former aims at testing the interaction between the PEP and the PDP by generating test cases specifically conceived to target possible faults in the PDP implementation. The latter instead is focused on the runtime execution of the policy and therefore on the interaction between the PDP and the PIP. In this latter case, the test cases focus on modifications of the attributes so to stimulate the continuous policy enforcement. These modifications do not

concern attribute updates triggered as a consequence of the evaluation of the PolPA policy performed by the PDP.

Before detailing the history-based and the continuous policy enforcement test case generation, we introduce the definition of the PIPCommand that is sent from the test driver to the MAU component for updating the values of the attributes of the user, resource and environment. Using the PIPCommand we simulate in the testing framework the runtime modification of attributes values.

Definition 1 (**PIPCommand(C)**). Given a policy P and a predicate p (having a set of attributes A) of a $tryaccess(s, o, r)$ or $revokeaccess(s, o, r)$ command, called C, we define a PIPCommad(C), as a command *update* whose parameter is a set of pairs (a, v), where, a$\in A$ and v is a proper typed value for a.

6.1 History-based test cases generation

The Test Cases Generator implements a testing procedure specifically conceived for a PolPA policy and for each faulty policy derived from it. This strategy (previously presented in [4]) consists of four main steps:

(i) parsing of the policy. The parsing output is a binary tree preserving the hierarchy of the composition operators and the related commands of the policy;

(ii) parameter values assignment. The action parameter values and the PIPCommands are generated such that the result of the evaluation of predicate associated to the $tryaccess(s, o, r)$ command is true while that of the associated $revokeaccess(s, o, r)$ command (if any) is false;

(iii) visit depth-first of the tree. For each node k derive the multi set $MS_k = \{S^1, \cdots, S^n\}$, where S^i is a ordered set of commands, as in the following:
   - if the node k is the command named a_1, $MS_k = \{\{a_1\}\}$;
   - if the node k is the *or* composition operator, $MS_k = MS_1 \cup MS_2$ where $MS_{i \in \{1,2\}}$ is the multi set derived for the node i that is child of node k;
   - if the node k is the $\cdot$ composition operator, and MS_{i1} and MS_{i2} are the multi sets associated to *i1* and *i2* that are children of k, MS_k contains all the ordered combinations of the elements of MS_{i1} and MS_{i2};
   - if the node k is the *par* composition operator, and MS_{i1} and MS_{i2} are the multi sets associated to *i1* and *i2* that are children of k, MS_k contains all combinations of the elements of MS_{i1} and MS_{i2} in each possible order;

(iv) test case derivation. For each element S^i of MS_r where r is the root node, derive a test case containing the ordered sequence of commands of S^i, the associated parameters values and the defined PIPCommads.

For instance a test case generated from the policy of Table 1 applying the testing procedure described above is:

t_h = {*update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R1, A ("val1", "val2")), endaccess(U, R1, A("val1", "val2")), tryaccess(U, R2, C ("val3", "val4")), endaccess(U, R2, C("val3", "val4")), tryaccess(U, R1, D ("val5")), endaccess(U, R1, D("val5"))*}

This test case contains a PIPCommand (*update((U.reputation, 76), (R2.workload, 69))* that sets the evaluation of the predicate associated to the *revokeaccess*(s, o, r) command at lines 17 and 18 of Table 1 to false, then stimulates the execution of: the *tryaccess*(s, o, r) and *endaccess*(s, o, r) at lines 1 and 3 respectively; the *tryaccess*(s, o, r) and *endaccess*(s, o, r) at lines 13 and 15 respectively and finally the *tryaccess*(s, o, r) and *endaccess*(s, o, r) at lines 23 and 25.

We recall that, for the sake of simplicity, we use the placeholders U, $R1$ and $R2$ to represent the real names of the user and of the resources, that in our example could be, respectively: "C=IT, L=PISA, O=CNR-IIT, CN=paolo mori", "http://node3.iit.cnr.it/axis2/services/MyTestService"and "http://node3.iit.cnr.it/axis2/services/AnotherService".

6.2 Test cases generation for continuous policy enforcement

In this section we define the strategy adopted for the generation of test cases specifically conceived for addressing the dynamic runtime behavior of the PDP due to the mutability of attributes. Similarly to those generated for history based testing (see Section 6.1), these additional test cases (requests) consist of a sequence of commands (*tryaccess*(s, o, r) and *endaccess*(s, o, r)) for the PDP. In addition, they include a set of commands for Mutable Attribute Updater (MAU) component in order to change the attributes values. This modification of attributes is performed during the execution of a test case and could invalidate the access rights causing the access revocation according to the PolPA policy. Note that the continuous policy enforcement testing involves only the PolPA policy and not the faulty policies derived from it.

The testing strategy defined in this section is based on the Multiple Condition Coverage (MCC) approach [15]. A multiple condition is made up from one or more conditions, which are combined by logical operators (and, or, not). To get 100% of the coverage, for each multiple condition, all combinations of true and false for its involved conditions have to be evaluated[4]. PolPA language allows for specifying some multiple conditions, called predicates and defined as regular expressions involving the parameters of the actions and the attributes of users, resources and environment. Thus, to get 100% of the coverage of a single PolPA predicate, all combinations of true and false for parameters of the actions and for the attributes of the users, resources and environment have to be considered. By iterating the process for every predicate, the 100% of the coverage of a PolPA policy is reached when the 100% of the coverage of all its predicates is achieved.

[4] Note that the term condition in this section does not refer to the environment conditions specified in the UCON model.

In the rest of this section we first provide some definitions and then the proposed test cases generation procedure.

6.2.1 Definitions

The continuous policy enforcement testing focuses on predicates associated to a $revokeaccess(s, o, r)$ command, therefore we introduce the following definitions, involving a PolPA predicate, useful for the testing strategy specification:

Definition 2 (**BeforeCondition(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, a BeforeCondition(RA), shortened into BC(RA), is a tuple $< PC$, $tryaccess(s, o, r)$, $endaccess(s, o, r) >$ where PC is a PIPCommand(RA) and $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ are the commands in P that RA refers to.

Definition 3 (**DuringCondition(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, a DuringCondition(RA), shortened into DC(RA), is a tuple $< PC1$, $tryaccess(s, o, r)$, PC2, $endaccess(s, o, r) >$ where PC1 an PC2 are PIPCommand(RA)s and $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ are the commands in P that RA refers to.

Definition 4 (**AfterCondition(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, an AfterCondition(RA), shortened into AC(RA), is a tuple $< PC1$, $tryaccess(s, o, r)$, $endaccess(s, o, r)$, $PC2 >$ where PC1 an PC2 are PIPCommand(RA)s and $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ are the commands in P that RA refers to.

Considering the policy of Table 1, and the predicate at line 17 that immediately precedes the *revokeaccess(user_id, R2, C(*x_4*,* x_5*))* at line 18, called RA2, the corresponding PIPCommand(RA2) is equal to $update((U.reputation, v1), (R2.workload, v2))$ where $v1$ and $v2$ are two possible integer values for reputation and workload attributes respectively.

Moreover, for the same $revokeaccess(user_id, R2, C(x_4, x_5))$ we have:

- BeforeCondition(RA2) = <*update((U.reputation,* v_1*), (R2.workload,* v_2*)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3","val4"))*>;
- DuringCondition(RA2) = <*update((U.reputation,* v_1^1*), (R2.workload,* v_2^1*)), tryaccess(U, R2, C("val3", "val4")), update((U.reputation,* v_1^2*), (R2.workload,* v_2^2*)), endaccess(U, R2, C("val3","val4"))*>;
- AfterCondition(RA2) = <*update((U.reputation,* v_1^1*), (R2.workload,* v_2^1*)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3","val4")), update((U.reputation,* v_1^2*), (R2.workload,* v_2^2*))*>;

where $U.reputation$ and $R2.workload$ represent the reputation and workload attributes of the user U and resource $R2$ respectively whereas v_1^1, v_2^1, v_1^2 and v_2^2 are possible integer values for these attributes.

Definition 5 (**MultipleBefore(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, a MultipleBefore(RA), shortened into MB(RA), is a set of BC(RA) such that it guarantees 100% Multiple Condition Coverage (MCC) of p.

Definition 6 (**MultipleDuring(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, a MultipleDuring(RA), shortened into MD(RA), is a set of $DC(RA)_i = < PC1_i$, $tryaccess(s, o, r)$, $PC2_i$, $endaccess(s, o, r) >$ such that $PC1_i$ guarantees that p is false and $\bigcup_i PC2_i$ guarantees 100% MCC of p.

Definition 7 (**MultipleAfter(RA)**) Given a policy P and a predicate p that immediately precedes a $revokeaccess(s, o, r)$, called RA, in P, a MultipleAfter(RA), shortened into MA(RA), is a set of $AC(RA)_i = < PC1_i$, $tryaccess(s, o, r)$, $endaccess(s, o, r)$, $PC2_i >$ such that $PC1_i$ guarantees that p is false and $\bigcup_i PC2_i$ guarantees 100% MCC of p.

Considering the policy of Table 1, and the predicate at line 17 that immediately precedes a *revokeaccess(user id, R2, C(x_4, x_5))* defined at line 18, called RA2, we have:

MultipleBefore(RA2)(MB(RA2))= {*<update((U.reputation, 74), (R2.workload, 71)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3", "val4"))>, <update((U.reputation, 74), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3","val4"))>, <update((U.reputation, 76), (R2.workload, 71)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3","val4"))>, <update((U.reputation, 76), (R2. workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C ("val3", "val4"))>*};

MultipleDuring(RA2)(MD(RA2))= {*<update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), update((U.reputation, 74), (R2.workload, 71)), endaccess(U, R2, C("val3","val4"))>, <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), update((U.reputation, 74), (R2.workload, 69)), endaccess(U, R2, C("val3", "val4"))>, <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), update((U.reputation, 75), (R2.workload, 71)), endaccess(U, R2, C("val3", "val4"))>, <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), update((U.reputation, 75), (R2.workload, 69), endaccess(U, R2, C("val3", "val4"))>*};

MultipleAfter(RA2)(MA(RA2))= {< {*update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C ("val3", "val4")), update((U.reputation, 74), (R2.workload, 71))>, <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3", "val4")), update((U.reputation, 74), (R2.workload, 69))>, <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3", "val4")), update(U.reputation, 75), (R2.workload, 71)>, <update((U.reputation, 76),*

(R2.workload, 69)), tryaccess(U, R2, C("val3", "val4")), endaccess(U, R2, C("val3", "val4")), update((U.reputation, 75), (R2.workload, 69))>}.

Definition 8 (**Multiple Coverage Domain**) Given a policy P and given S_{RA} the set of the $revokeaccess(s, o, r)$ commands of P, the Multiple Coverage Domain (MCD) is the multiset of MB(RA), MD(RA), and MA(RA) such that $RA \in S_{RA}$.

The policy of Table 1 has two $revokeaccess(s, o, r)$ commands called RA1 and RA2 (line 9 and 18 respectively). The MCD for that policy is equal to $\{MB(RA1), MD(RA1), MA(RA1), MB(RA2), MD(RA2), MA(RA2)\}$.

Following the standard approaches for multiple condition coverage [15] we define the following adequacy criterion:

Definition 9 A test suite T for a policy P is considered adequate with respect to the Multiple Coverage Domain (MCD), iff for every element in MCD there exists at least one test case t ∈ T that covers it.

The Test Cases Generator implements a testing procedure specifically conceived for deriving a test suite T that guarantees the above adequacy criterion.

6.2.2 Test cases generation procedure

Similarly to the history-based approach, the test cases generation procedure consists in the following steps:

(i) parsing of the policy;
(ii) assignment of parameter values:
  - for each $revokeaccess(s, o, r)$ command called RA preceded by predicate p, the sets MultipleBefore(RA), MultipleAfter(RA), MultipleDuring(RA) as defined in Section 6.2.1 are derived. For each of these sets a table, called PIPTruthTable, is derived that contains: the truth values of its PIPCommands and the result of the evaluation of the overall predicate p;
  - the action parameter values and the PIPCommands are generated such that the result of the evaluation of predicate associated to the *tryaccess(s,o,r)* command is true (see step *(ii)* of the testing procedure described in Section 6.1);

(iii) visit depth-first of the tree. For each node k, the multi set MS_k as described in Section 6.1 is derived;
(iv) pruning of the tree. For each $revokeaccess(s, o, r)$ command, called RA, the shortest subtree rooted in the · composition operator, involving RA and containing the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ commands, is replaced by a leaf containing the set $MS_{RA} = \{MB(RA), MD(RA), MA(RA)\}$;
(v) test case derivation. For each element S^i of MS_r where r is the root node, a test case is derived that contains the ordered sequence of commands of S^i and the associated parameters values.

For instance, applying the testing procedure described above to the policy of Table 1 and considering in particular the *revokeaccess(user id, R2, C(*x_4*,*x_5*))*

defined at line 18, called RA2, we have that $MS_{RA2} = \{MB(RA2), MD(RA2), MA(RA2)\}$, where MB(RA2), MD(RA2) and MA(RA2) are listed at the end of Section 6.2.1, and a possible generated test case is the following: t_c= {*tryaccess(U, R1, A(“val1”, “val2”)), endaccess(U, R1, A(“val1”, “val2”)), <update((U.reputation, 76), (R2.workload, 69)), tryaccess(U, R2, C(“val3”, “val4”)), update((U.reputation, 74), (R2.workload, 71)), endaccess(U, R2, C (“val3”, “val4”))>, tryaccess(U, R1, D(“val5”)), endaccess(U, R1, D(“val5”))*}

Note that this test case is derived considering the first element of MD(RA) and stimulates the execution of the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ at lines 1 and 3 respectively; then it defines the PIPCommands (*update((U.reputation, 76), (R2.workload, 69))),(update((U.reputation, 74), (R2.workload, 71)))*) and stimulates the execution of the the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ at lines 13 and 15 and finally stimulates the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ at lines 23 and 25.

We outline that the proposed test cases generation procedure does not deal with attributes updates triggered as a consequence of the evaluation of the PolPA policy on the PDP. The only attributes modifications we consider here are those related to the external environment and managed by the PIP component.

7 Test Driver

The Test Driver (TD) executes the test cases derived both according to the history-based strategy and for the continuous policy enforcement.

In the former case, the TD simulates the PEP behavior while interacting with the PDP for executing the sequence of $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ commands specified in the test case. In addition, before the execution of the test case, The TD interacts also with the MAU to update the attributes values. In particular, the MAU is in charge of managing the mutable attributes such that all predicates of the $revokeaccess(s, o, r)$ commands in the executed policy are false. This guarantees that any $revokeaccess(s, o, r)$ command is not thrown during the test case execution.

For instance, during the execution of t_h test case, the TD before sending *tryaccess*(U, $R2$, C(“val3”, “val4”)) to the PDP, interacts with the MAU in order to update the values of reputation and workload attributes such that the predicate at line 17, i.e., the one that triggers the $revokeaccess(s, o, r)$ associated to the action C, results false.

In case of continuous policy enforcement, the TD simulates, as before, the behavior of the PEP and sends the PIPCommands to the MAU. Specifically, the TD executes the ordered sequences of commands in each test case. If the command is a $tryaccess(s, o, r)$ or $endaccess(s, o, r)$, the TD interacts with the PDP. If a PIPCommand is specified in the test case, the TD interacts with the MAU in order to update the value of each attribute with the proper typed value and consequently to warn the PDP about mutable attributes changes.

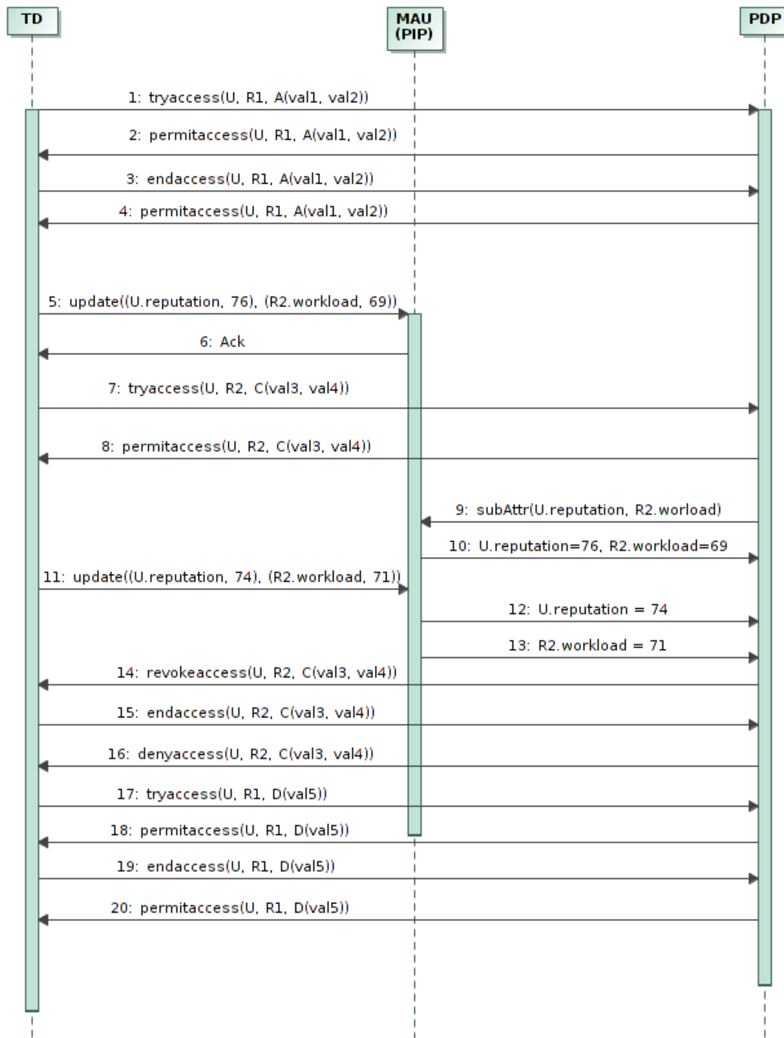

Fig. 3 Test case execution

For instance during the execution of t_c test case presented at the end of Section 6.2, the sequence of performed actions, as showed in Figure 3, includes:

- the TD sends *tryaccess(U, R1, A("val1", "val2"))* and *endaccess(U, R1, A("val1", "val2"))* to the PDP;
- the TD sends the PIPCommand = *update((U.reputation, 76), (R2.workload, 69))* to the MAU so to guarantee that the evaluation of the predicate of the *revokeaccess(s,o,r)* associated to the action C will result false when the *tryaccess*(U, $R2$, C("val3", "val4")) will be allowed;

- the PDP sends to the MAU the subscription for *U.reputation* and *R2.workload* attributes;
- the TD sends the PIPCommand = *update((U.reputation, 74), (R2.workload, 71))* to the MAU to update the attributes values. The MAU changes the *U.reputation* and *R2.workload* attribute values and notify the PDP about this runtime change that triggers the *revokeaccess(U, R2, C(val3,val4))* command given back from the PDP to the TD;
- the TD executes $endaccess$(U, $R2$, C("val3","val4")) command that will trigger a deny access because the access for the action C has been revoked;
- finally the TD sends $tryaccess(U, R1, D($"val5")) and *endaccess(U, R1, D("val5"))* to the PDP.

8 Test Oracle

The test oracle interacts with the Test Case Generator to retrieve the policy and the derived set of test cases. It collects the PDP responses (*permitaccess(s,o,r)*, *revokeaccess(s,o,r)* or *denyaccess(s,o,r)*) caused by a test execution and compares the obtained results with the correct authorization replies associated to each of the generated test cases. The oracle relies on a predefined verdicts table that collects the detailed set of couples (test case / (set of) expected response(s)). In the current implementation, for the history-based testing, this verdict table is manually derived. We are evaluating and defining possible solutions for speeding up the derivation and analysis of the history-based test results.

On the contrary, in case of continuous policy enforcement testing, the derivation and evaluation of the expected test results is fully automated. Specifically, the oracle analyzes the test case structure and performs the following actions:

1. it extracts the sequence of commands before an element of a MultipleBefore(RA) (MultipleDuring(RA) or MultipleAfter(RA)) included in the test case, where RA is the addressed $revokeaccess(s, o, r)$ command, and ignores the replies of the PDP associated to these commands: these are not the target of the continuous policy enforcement testing[5]. Note that in the rest of this section with the sentence *ignores the replies of the PDP* we mean that the oracle ignores the $permitaccess(s, o, r)$ and $denyaccess(s, o, r)$ that can be received from PDP. In case of error or delay of the replay, the PDP replay is considered erroneous, the test case execution is terminated and the overall verdict is fail. For aim of completeness, we recall here that the element of a MultipleBefore(RA) (MultipleDuring(RA) or MultipleAfter(RA)) included in the test case is a BeforeCondition(RA) (DuringCondition(RA) or AfterCondition(RA));

[5] Indeed in this evaluation we suppose that the correctness of these PDP replies has been validated during a previously executed phase of history-based testing.

2. it lists the PIPcommand(RA)s of the BeforeCondition(RA) (DuringCondition(RA) or AfterCondition(RA)) included in the test case, extracts from the policy the single conditions defined for the predicate preceding RA, and uses the values of the PIPcommand(RA)s to evaluate them. Thus for each PIPcommand(RA) the list of the boolean values is derived;
3. it maps the list of the boolean values of the PIPcommand(RA)s with the proper row of the PIPTruthTable of the MultipleBefore(RA) (MultipleDuring(RA) or MultipleAfter(RA)) at which BeforeCondition(RA) (DuringCondition(RA) or AfterCondition(RA)) belongs to and gets the associated overall boolean evaluation of the predicate preceding RA;
4. if the evaluation of the predicate preceding RA is false, it ignores the PDP replies for the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ commands involved in the BeforeCondition(RA) (DuringCondition(RA) or AfterCondition(RA));
5. if the evaluation of the predicate preceding RA is true:
   - in case of BeforeCondition(RA), it ignores the reply associated to the $tryaccess(s, o, r)$ command specified in the BeforeCondition(RA) and the expected reply is a $revokeaccess(s, o, r)$ command;
   - in case of DuringCondition(RA), it ignores the reply associated to the $tryaccess(s, o, r)$ command specified in the DuringCondition(RA) and the expected reply is a $revokeaccess(s, o, r)$ command;
   - in case of AfterCondition(RA), it ignores the replies associated to the $tryaccess(s, o, r)$ and $endaccess(s, o, r)$ commands specified in the AfterCondition(RA). No $revokeaccess(s, o, r)$ command is allowed;
6. it extracts the sequence of commands after the BeforeCondition(RA) (DuringCondition(RA) or AfterCondition(RA)) included in the test case and ignores replies of the PDP associated to these commands that are not the target of the continuous policy enforcement testing.

9 Empirical Evaluation

In this section we detail the application of the proposed framework to the exploratory policy of Table 1. In particular, we simulate a generic situation in which the system under test (SUT) is the PDP implementing a policy specification that is considered correct. Considering the architecture presented in Figure 1, in this testing scenario the real PEPs and the PIP have been substituted by our testing framework, the PAP is supposed to be correct and the SUT is the PDP implementation exploited in [14].

We performed the assessment in two steps, the former to validate the static behavior of the PDP focusing on the history-based aspects of the PolPA language, the latter to assess the continuous policy enforcement. We describe both steps in the next sections.

9.1 First step

Following the approach described in Section 6.1, by applying the fault model to the policy of Table 1, the FPG component derived 37 mutated policies (as motivated at the end of Section 5). We report the number of derived mutated policies for each mutant class in the second column of Table 2. Successively, from each of the available policies, the TCG component derived the corresponding set of test cases by the application of the procedure described in Section 6.1. Specifically, TCG generated 2 test cases from the original policy and 45, 16, 8, 20 and 2 test cases from the policies mutated according to the CCO, CC, CGSO, CGIO and CLO mutant class respectively (see Table 2 third column) for a total of 93 test cases. It is worth noticing that, when the test strategy described in Section 6.1 is applied to the mutated versions of a PolPA policy, there is the possibility to derive the same test case several times. In this exploratory example the amount of redundant test cases is 7 over the 93 derived, where 4 are in the set derived from the policies mutated according to CC mutant class and 3 in the rest. However, since this redundancy does not compromise the effectiveness of the methodology, in the current version of the testing framework we did not include techniques to eliminate this problem; this is part of our future work. Finally, each of the test cases has been executed on the PDP and obtained responses have been collected and compared with the expected ones.

Table 2 reports the comparison results in the last column. We saw that all the responses obtained from the PDP were the same of those expected, except those related to the test cases derived from the policies mutated according to the CC mutant class (thus a 0 in the column labeled # Faults). This means that the PDP implementation compliant with the gold policy, did not contain any of the related faults. A different situation has been experienced for test cases derived from the policies mutated according to the CC mutant class. In this case, for 1 over the 16 executed test cases, the response obtained was not the expected one. In particular, we noticed that the fault was always detected by the test case derived by the mutant having two $tryaccess(s, o, r)$ commands involving the first action allowed by the policy (i.e., action A in the policy of Table 1). This behavior is not compliant with what explicitly specified in the policy of Table 1, because after the first request for executing action A, represented by the $tryaccess(s, o, r)$ command in line 1, the PDP only allows the corresponding $endaccess(s, o, r)$ command in line 3, that represents the end of action A. Thus, the detected anomaly in the test case response pointed out a problem in the PDP implementation. Talking with the developers, they said that the implementation they provided for the test allows users to behave as stated by the policy an arbitrary number of times, even in parallel, because this was a specific requirement of the scenario the tested implementation was developed for. This explains why the tests in which the user tries to execute the first action of the policy for a second time returned a result that was not the expected one. For a generic release of the PDP this feature should

Table 2 Experimental DATA for history-based testing

		# Mutants	# TS1	# Faults
Policy		-	2	0
Mutation Class	CCO	14	45	0
	CC	8	16	1
	CGSO	4	8	0
	CGIO	10	20	0
	CLO	1	2	0
Total		37	93	1

Table 3 Experimental DATA for continuous policy enforcement testing

	# MB(RA)	# MD(RA)	# MA(RA)	# TS2	# Faults
RA1	2	2	2	6	4
RA2	4	4	4	12	0
Total				18	4

be modified, and if a same $tryaccess(s, o, r)$ command in a row is issued, it should be denied[6].

The discovered fault showed an important limitation of the considered PDP implementation and confirmed the effectiveness of the proposed testing framework.

9.2 Second step

For the continuous enforcement testing, we applied the approach described in Section 6.2 to each $revokeaccess(s, o, r)$ command of the policy of Table 1. Specifically, as showed in Table 3, for the *revokeaccess* of line 9 $((lessthan(attr(reputation, user_id), 85)).revokeaccess(user_id, R2, B(x_3)))$, here called RA1, we derived the sets MB(RA1), MD(RA1) and MA(RA1), each with cardinality 2, guaranteeing the Multiple Condition Coverage (MCC) of the predicate that immediately precedes RA1. Following the procedure described in Section 6.2.2 we derived a test case for each element of MB(RA1), MD(RA1) and MA(RA1), for a total of 6 test cases.

We performed the same steps for the *revokeaccess* of line 17 and 18 *((lessthan(attr(reputation, user_id),75))and(morethan(attr(workload,R2),70)).revokeaccess* $(user_id, R2, C(x_4, x_5)))$, here called RA2, and we derived first the sets MB(RA2), MD(RA2) and MA(RA2), each with cardinality 4, and then we derived a total of 12 test cases.

[6] Note that the same error was also detected in a previous experiment described in [4].

Each of the 18 test cases derived for the overall policy, has been executed on the PDP and the obtained responses have been automatically evaluated by the test oracle component as described in Section 8. For 4 of the 18 test cases, two of them derived from the elements of MD(RA1) and two from those of MA(RA1), the obtained responses were not the expected ones. Specifically, for these test cases the obtained result was the same error message. This error was due to the fact that the PDP subscribed to a corrupted attribute name. In more detail, the PDP in the storing of the *U.reputation* name in its local memory, used a corrupted name of the attribute due to an internal error. Thus when PDP subscribed to the MAU component, it did not recognize *U.reputation* name and consequently sent an error to the PDP. The PDP in this case was not able to perform the decision process and returned an error to the TD and TO components. Reporting this error to the developers, they found out that in the integration of the functionalities for the continuous policy enforcement, they introduced an error in the attributes memory allocation (please notice that the PDP has been implemented using the C language). For the remaining 14 test cases the obtained results were the expected ones.

9.3 Time Evaluation

In this section we provide some details about the time required for completing the history-based and continuous policy enforcement testing phases of the above proposed experience. Specifically, we run the PDP implementation exploited in [14] and the proposed testing framework on the same PC having an Intel(R) Pentium(R) 4 CPU (2.00 GHZ), 1,5 GB of Memory and Linux Ubuntu Release 10.04, Kernel 2.6.32-43-generic, as operating system. We run 20 times the history-based and continuous policy enforcement testing on the PolPA policy of Table 1, and we computed the average completion times. In the following, we summarize the results about the main phases of the proposed testing approach:

for history-based testing:

- around 3 minutes for completing the execution of Fault Model Manager and Test Cases Generator components;
- around 10 seconds for completing the execution of Test Driver component;
- around 4 hours for completing the execution of Test Oracle component (note that this time includes the manually derived verdicts table).

for continuous policy enforcement testing:

- around 12 seconds for completing the execution of Test Cases Generator component;
- around 6 seconds for completing the execution of Test Driver and Test Oracle components.

The obtained results evidenced that the history-based testing took more time than the continuous policy enforcement testing. This was due mainly to the manual derivation of the verdicts table. However, also the generation and execution of test cases for the history-based testing were longer than the corresponding ones for the continuous policy enforcement testing. This was due to the generation of the fault model and the derivation of a higher number of test cases, as showed in Tables 2 and 3 (the history-based testing included 93 tests cases while continuous policy enforcement testing 18 test cases).

9.4 Discussion

The conclusions we can draw from this case study must be taken in light of the threats to validity of the performed experiment. The presented study relates to a real policy and one PDP implementation, then further experimentation involving more policies and PDP implementations would be required. The proposed fault model represents a subset of possible mutation operators, further mutation classes introducing for instance modifications inside a single PolPA command should be included and evaluated in the testing framework.

The proposed testing strategy for continuous usage control is based on modifications of the attributes performed by external entities to the PDP and does not deal with updates of attribute values triggered as a consequence of the runtime evaluation of the PolPA policy performed by the PDP. Additional test cases aimed at forcing specific PDP behaviors and policy evaluation states of the PDP should be derived.

Finally, the expected results for history-based testing are currently derived in non automated way, requiring additional labor from the user of the testing framework. Except for possible faults in our testing framework implementation, for the rest all steps have been carried out in automated way, so we do not see other internal threats.

Without neglecting the above limitations, this case study confirmed the applicability of the proposed testing strategy to real PolPA policies and the effectiveness of the implemented testing framework for evaluating the runtime PDP behavior. Although simple, this exploratory example provided the possibility of deriving an effective automated oracle. Moreover, it gave the opportunity of applying the multiple condition coverage approach even in case of predicates preceding *revokeaccess(s,o,r)* commands involving more than one attributes, so to extensively explore all the different PDP behaviors in the continuous policy enforcement.

10 Related Work

Our proposal in inspired by approaches of software testing, specifically by test cases derivation and mutation analysis. As surveyed in [18], in literature several works propose fault models, based on FSM or LTS, for test case derivation.

In the context of access control systems, that is our domain of research, the work in [11] defines a fault model and a set of mutation operators for XACML access control policies. The defined mutation operators manipulate the predicates and logical constructs of target and condition elements, emulating syntactic and semantic faults introduced in the policy. The main differences with our approach are: *i)* the defined mutation operators are used to evaluate the fault-detection capability of a test suite and its relation with the structural coverage achieved by the same test suite and not for test cases derivation; *ii)* the fault model and the defined mutation operators are specifically conceived for XACML language and they do not address PolPA language peculiarities. In [16] the authors try to extend the mutation operators of [11], focusing on specific security aspects and relying on the use of a metamodel that allows to simulate the faults in the security models independently from the used role-based formalism (R-BAC or OrBAC). Finally, the work in [3] includes and enhances the mutation operators of [11] and [16] addressing specific faults of the XACML 2.0 language and providing a tool for the derivation of XACML mutation operators and their application to XACML policies. Differently from our proposal, also in this work mutation analysis is performed for test adequacy and not for test cases generation.

Few works address mutation analysis for test case derivation as in our proposal. Specifically, the authors of [21] propose data mutation to generate a larger test suite from a few seed test cases. Mutation operators are applied not to the program under test or the specification of the software but to the input data with the aim of varying a test case in order to produce another one. One of the closest works to our proposal is that of [6]. In this paper, security-specific mutation operators are used to introduce leaks into communication protocols models so to evidence a correlation between model-level mutants and implementation faults. In particular, the protocol model is modified in order to violate a security property in a specific way that is related to a common mistake at the implementation level, the traces of the mutated model are collected, and the counter-examples (test cases) for the specified security property are derived accordingly. Our work differs in the definition of the fault model, which is based on PolPA language, and on the methodology for test cases derivation. We also provide a testing framework to automatically derive the mutants and the test cases and to execute the test cases on the PDP.

Other approaches focus on test cases derivation and use already defined mutation operators for assessing the effectiveness of the proposed testing strategy [5,12,19,13]. Specifically, the X-CREATE framework [5] and the Targen tool [12] generate test inputs using combinatorial approaches of the XACML policies values and the truth values of independent clauses of policy values, respectively. The work in [19] deals with model model-based testing and provides a methodology for the generation of abstract test cases to be then refined into concrete requests for the PDP. Also in this work, combinatorial approaches of the elements of the model (role names, permission names, context names) are used to derive test cases. However, combinatorial approaches are not suitable for testing the potentialities of PolPA policies because they do not deal with

the semantic of the commands composition and do not take into account the history of the previous accesses and more in general the Usage Control mechanisms. Alternatively, the Cirg [13] approach applies change-impact analysis for test cases generation starting from policy specification. Specifically, it provides a framework able to synthesize two versions of the policy under test: the original version and a modified policy version in which, for instance, a decision rule is negated. The same framework applies change impact analysis in order to output counterexamples that evidence semantic difference between the two policies. Each counterexample represents a request that when executed with the two policies gives two different responses. These requests with the associated responses represent the test cases generated by the proposed framework.

Considering the Usage Control model, there are not proposals similar to ours for testing this specific type of authorization systems, to the best of our knowledge. Available solutions, such as [1], propose proactive mechanisms for preventing possible policy violations and present a combination of runtime monitoring and self-adaptation to simplify the autonomic management of authorization infrastructures. In recent years, as surveyed in [17], testing of authorization systems has been focused on evidencing the appropriateness of the UCON enforcement mechanisms, focusing on performance analysis or establishing proper enforcement mechanisms by means of formal models. By proposing a framework for testing PDP implementation, our approach nicely complements the existing works.

A comparison of our proposal with related works and their critical analysis are presented in Table 4. For each reference, we provide a short description in the last column of the table and we classify it according to seven dimensions: if the specification of a fault model is provided (column labeled *FM*); the usage of mutation analysis either for evaluating the test suite effectiveness (column labeled *Test Adequacy*) or for deriving test cases (column labeled *Input Derivation*); the provided strategy for tests case derivation (column labeled *Test Generation Strategy*), the considered system under test (column labeled *SUT*), if the reference provides a framework implementing the proposed approach (column labeled *Framework*) and finally if the reference provides a case study for validating the results (column labeled *Case Study*).

Table 4 evidences that many of the analyzed references use mutation analysis for assessing the effectiveness of a test suite, among them few proposals define new mutation operators while most works use existing mutation approaches for evaluating the proposed testing strategy. Our proposal, as some of the analyzed works, focuses on mutation of the model for deriving test cases. Specifically, we define a fault model and a fault seeding strategy of the PolPA policy for deriving test cases. Many of the analyzed works deal with test cases generation strategies for access control systems specified by XACML policies or RBAC models: most of them are based on combinatorial approaches of the input or model data, other ones on change impact analysis. Differently from these works, the testing strategy proposed in this paper is based on the coverage of the derived fault model for the history based testing and on Multiple

Condition Coverage methodology for the continuous policy enforcement testing. However, to the best of our knowledge, our proposal is the only existing solution for testing the UCON system specified by PolPA policy. Finally, most of the analyzed references provide a framework and a case study whereas some of them define a fault model.

Table 4: Comparison of our proposal with main related work.

		Mutation Analysis						
Ref	**FM**	**Test Adequacy**	**Input Derivation**	**Tests Generation Strategy**	**SUT**	**Framework**	**Case study**	**Description**
[18]	✓	-	-	fault model coverage	FSM	-	-	Annotated bibliography of test cases generation approaches from fault model
[21]	-	-	by data mutation	fault seeding of test inputs	CAMLE modeling language	-	✓	A generic approach for generating test cases by data mutation
[11]	✓	mutants definition	-	-	XACML policies	✓	✓	A set of mutation operators for XACML policies
[3]	✓	mutants definition	-	-	XACML policies	✓	✓	A tool for mutation analysis of XACML policies
[5]	-	mutants usage	-	combinatorial on the input set	XACML policies	✓	✓	A tool for XACML requests generation
[12]	-	mutants usage	-	combinatorial on truth values of independent clauses	XACML policies	✓	✓	A tool for XACML requests generation
[19]	-	mutants usage	-	combinatorial on the elements of the model	extended RBAC model	✓	✓	A model-based approach to derive test cases for access control requirements
[13]	-	mutants usage	-	change impact analysis	XACML policies	✓	✓	A tool for generation of XACML requests and responses
[6]	✓	-	by security model mutation	attack traces derivation	security protocol	-	-	A test generation strategy from security properties
this paper	✓	-	by fault seeding of PolPA policy	fault model coverage and MCC	PolPA PDP	✓	✓	A testing framework for UCON systems

11 Conclusions

In this paper we proposed a framework aimed at the automated generation and execution of test cases, for testing of the PolPA-based PDP implementation. We proposed two test cases generation strategies focused on the main features of the PDP: the history-based control and the usage control. In the former, the test cases cover a defined fault model and target the main problems and vulnerabilities of the passive PDP implementation. The latter is specifically conceived for addressing the dynamic runtime behavior of the PDP due to the mutability of attributes. It is based on the standard technique for multiple condition coverage and integrates a methodology for simulating all the possible ways in which attributes can change during the runtime execution of PolPA commands. For testing the continuous policy enforcement functionality, we also provided a fully automated facility for the derivation and evaluation of the expected test results.

We have evaluated both strategies on a real policy and PDP implementation and showed the effectiveness of the proposed approach in revealing faults. From the obtained results, we concluded from one side that the automated tests generation guided from the fault model represents a valid contribution for detecting static behavioral problems of a PDP implementation. From the other side, a methodology based on the coverage of attributes values conditions extensively explores all the different runtime PDP behaviors during the usage control.

A preliminary evaluation confirmed the applicability of the proposed testing strategy to real PolPA policies and the effectiveness of the implemented framework for testing the runtime PDP behavior.

We are currently working in several directions: in developing a methodology for automatically deriving the expected results in history-based testing; in improving the mutation classes in the fault model; in reducing the number of redundant test cases while keeping the same test effectiveness; in defining a set of PolPA policies that can be used as conformance test suite.

Future work will also include further experimentation and the application and adaptation of the proposed testing framework to other usage control systems based on different policy specification languages such as U-XACML [8]. We have seen that our approach is efficient for the presented PolPA policy, but when policies have a high structural complexity it might yield a high number of test requests When the test budget is limited we need to consider ways to reduce the generated test suite or to prioritize the derived test requests. A way for reducing generated test cases could be to consider weaker coverage criteria of the PolPA commands predicates than Multiple Condition Coverage, such as Modified Condition/Decision Coverage (MC/DC) or Decision Coverage.

Acknowledgment

This work has been partially funded by the Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS) FP7 Project contract n. 256980.

References

1. Bailey, C.: Application of self-adaptive techniques to federated authorization models. In: Proc. of 34th International Conference on Software Engineering (ICSE), pp. 1495–1498 (2012)
2. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E.: Automatic XACML requests generation for policy testing. In: Proc. of Fourth IEEE International Workshop on Security Testing (associated with ICST 2012), pp. 842–849 (2012)
3. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti., E.: XACMUT: XACML 2.0 Mutants Generator. In: Proc. of 8th International Workshop on Mutation Analysis (associated with ICST 2013) (2013)
4. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., Mori, P.: Testing of PolPA authorization systems. In: Proc. of 7th International Workshop on Automation of Software Test (associated with ICSE 2012), pp. 8–14 (2012)
5. Bertolino, A., Lonetti, F., Marchetti, E.: Systematic XACML Request Generation for Testing Purposes. In: Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 3–11 (2010)
6. Büchler, M., Oudinet, J., Pretschner, A.: Security mutants for property-based testing. In: Proc. of 5th International Conference on Tests and proofs (TAP), pp. 69–77 (2011)
7. Castrucci, A., Martinelli, F., Mori, P., Roperti, F.: Enhancing Java ME Security Support with Resource Usage Monitoring. In: Proc. of Information and Communications Security, *Lecture Notes in Computer Science*, vol. 5308, pp. 256–266 (2008)
8. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A proposal on enhancing XACML with continuous usage control features. In: Proc. of CoreGRID ERCIM Working Group Workshop on Grids, P2P and Services Computing, pp. 133–146. Springer US (2010)
9. Colombo, M., Martinelli, F., Mori, P., Martini, B., Gharbaoui, M., Castoldi, P.: Extending resource access in multi-provider networks using trust management. Journal of Comp. Net. and Comm. (IJCNC) **3**(3), 133–147 (2011)
10. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering **37**(5), 649 –678 (2011)
11. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proc. of 16th International Conference on World Wide Web (WWW), pp. 667–676
12. Martin, E., Xie, T.: Automated Test Generation for Access Control Policies. In: Supplemental Proc. of 17th International Symposium on Software Reliability Engineering (ISSRE) (2006)
13. Martin, E., Xie, T.: Automated test generation for access control policies via change-impact analysis. In: Proc. of Third International Workshop on Software Engineering for Secure Systems (SESS), pp. 5–12 (2007)
14. Martinelli, F., Mori, P.: On usage control for grid systems. Future Generation Computer Systems **26**(7), 1032–1042 (2010)
15. Mathur, A.P.: Foundations of Software Testing, 1st edn. Pearson Education (2008)
16. Mouelhi, T., Fleurey, F., Baudry, B.: A generic metamodel for security policies mutation. In: Proc. of Software Testing Verification and Validation Workshop (ICSTW), pp. 278–286 (2008)
17. Nyre, A.A.: Usage control enforcement-a survey. Availability, Reliability and Security for Business, Enterprise and Health Information Systems pp. 38–49 (2011)
18. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Proc. of the 4th Summer School on Modeling and Verification of Parallel Processes, pp. 196–205 (2001)

19. Pretschner, A., Mouelhi, T., Le Traon, Y.: Model-based tests for access control policies. In: Proc. of ICST, pp. 338–347 (2008)
20. Sandhu, R., Park., J.: The $\mathrm{UCON}_{\mathrm{ABC}}$ usage control model. ACM Transactions on Information and System Security **7**(1), 128–174 (2004)
21. Shan, L., Zhu, H.: Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. Comp. Jour. **52**, 571–588 (2007)
22. Zhang, X., Parisi-presicce, F., Sandhu, R.: Formal model and policy specification of usage control. ACM Transactions on Information and System Security **8**(4), 351–387 (2005)

Appendix

Acronym	**Expanded Version**
AC(RA)	AfterCondition($revokeaccess(s, o, r)$)
BC(RA)	BeforeCondition($revokeaccess(s, o, r)$)
CC	Change Command
CCO	Change Composition Operator
CGIO	Change Guard Integer Operator
CGSO	Change Guard String Operator
CLO	Change Logical Operator
DC(RA)	DuringCondition($revokeaccess(s, o, r)$)
FMM	Fault Model Manager
FPG	Faulty Policies Generator
MA(RA)	MultipleAfter($revokeaccess(s, o, r)$)
MAU	Mutable Attribute Updater
MB(RA)	MultipleBefore($revokeaccess(s, o, r)$)
MCC	Multiple Condition Coverage
MCD	Multiple Coverage Domain
MD(RA)	MultipleDuring($revokeaccess(s, o, r)$)
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PTSM	Policy Test Set Manager
SUT	System Under Test
TCG	Test Cases Generator
TD	Test Driver
TO	Test Oracle
UCON	Usage Control