Testing of PolPA Authorization Systems

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

Fabio Martinelli, Paolo Mori
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
via G. Moruzzi 1 - 56124 Pisa, Italy
{firstname.lastname}@iit.cnr.it

Abstract—The implementation of an authorization system is a difficult and error-prone activity that requires a careful verification and testing process. In this paper, we focus on testing the implementation of the PolPA authorization system and in particular its Policy Decision Point (PDP), used to define whether an access should be allowed or not. Thus exploiting the PolPA policy specification, we present a fault model and a test strategy able to highlight the problems, vulnerabilities and faults that could occur during the PDP implementation, and a testing framework for the automatic generation of a test suite that covers the fault model. Preliminary results of the test framework application to a realistic case study are presented.

Keywords—Authorization systems; PolPA language; testing; request generation;

I. INTRODUCTION

Security is a crucial aspect of modern information management systems, because stored data and other resources could be sensitive and valuable, and hence a proper support must be in place to protect them against unauthorized, malicious, improper or erroneous usage. For this purpose, authorization systems allow for the specification of access control policies which rule various protection aspects such as: the level of confidentiality of data, the procedures for managing data and resources, the classification of resources and data into category sets with different access controls. Several authorization system models have been defined in scientific literature, and some implementations of these systems, both free and commercial, are currently available on the Internet, such as the SUN’s XACML engine1.

Due to the vast area of research, in this paper we focus on testing the implementation of a specific authorization system, the PolPA one. The PolPA authorization system has been defined in [1], and its main peculiarity is that it supports Usage Control (UCON) and history-based control, i.e. it checks the sequence of security relevant actions performed by the user in order to prevent or even interrupt the execution of an action when the policy is not satisfied. This authorization system has been successfully adopted in several scenarios, such as the Grid [1], mobile devices [2], and Next Generation Networks [3].

From an architectural point of view, the PolPA authorization system includes several components, as detailed in Section III. All those components are critical from a security point of view and would require careful verification and testing. However, this paper is focused on testing the Policy Decision Point (PDP), which is the component that performs the decision process to decide whether an access should be allowed, according to the large amount of information and resources to be managed and ruled. The PDP implementation becomes a very critical activity for developers since unintentional points of security vulnerability, missing or misrepresentation of access authorization policy might be introduced. The risks grow in the case of complex, distributed and large systems, where multiple policy specifications and implementations are required and have to be managed. To prevent these problems a rigorous and accurate verification and testing process must be adopted. Available solutions for testing the PDPs are mainly focused on the well-known access control languages, such as OrBAC or RBAC or rely on the XACML standard, and cannot be easily transferred into the UCON environment. For this we define a specific testing framework customized for the PolPA language, specifically designed to deal with history-based security policies and the UCON model. Thus, focusing on the PDP that implements the authorization decision process specified into the PolPA security policy, we present:

- a fault model that highlights the problems, vulnerabilities and faults that could occur during the usage control policies implementation;
- a testing framework for the automatic generation of a test suite that covers the fault model. The test case generation relies on a original domain specific testing methodology presented also in this paper.

The complete testing process, from the fault model application to the test case execution, has been completely automated and adopted for the development of an illustrative case study.

The rest of this paper is structured as follows. Section II motivates the proposed approach. Section III briefly introduces the PolPA authorization systems. In Section IV we report details about the proposed testing framework whereas

1http://sunxacml.sourceforge.net/
in Section V we provide results of its application and also discuss about the test cases effectiveness and the efficacy of the adopted fault model. Section VI puts our work in context of related work and Section VII concludes the paper.

II. Motivations and Key Ideas

The PDP is a key component of the PolPA authorization system, because is the one that performs the decision process and determines whether an access can be performed or not. Hence, any error in the PDP implementation could have a serious impact on the decision process, authorizing accesses that should be forbidden by the policy or denying accesses that should instead be authorized. Moreover, we focus on the PDP implementation because the other components of the authorization system are typically dependent from the specific scenario where the authorization system has been embedded. On the contrary, the PDP is independent, because it does not interact directly with entities that are external to the authorization system.

An important aspect in test cases generation is clearly identify the target of the test case, i.e. problems, weaknesses that the test case should detect. The collection of test purposes represents a sort of fault model, which can be exploited for test case derivation. The generated test cases can specifically be designed and customized to detect the related system faults with the advantages of reducing the number of executed test cases and better managing the test effort.

However, 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 testing phase, either the coverage of a test criterion is considered or the fault detection effectiveness of the executed test suite is usually measured 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 effectiveness, but usually not directly defined for test cases derivation purposes. We refer to [4] for a recent and extensive survey of mutation testing approaches and their application.

In this paper the testing activity is focused on the detection of specific kinds of faults in the PDP implementation. Thus we adopt the following test process:

- identify the main problems of the PDP implementation, i.e. the fault model;
- from a given policy (gold policy), derive a set of faulty policies according to the defined classes of problems;
- according to a domain specific test strategy, generate the test cases able to detect the seeded faults, i.e. the selection of usage requests able to evidence a misinterpretation of authorization rules;
- execution of the test cases on a PDP implementation compliant with the gold policy;
- evaluation of the obtained test results against the expected output.

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 facilities that make easier the application of the various steps. Thus the focus of this paper is a framework which automatically generates the test cases, i.e. authorization requests, starting from a defined fault model. However, during the framework implementation we had to solve two main issues: how to automatically derive the set of faulty policies reflecting the problems defined in the fault model; how to automatically derive a set of test cases able to detect the faults. Every statement of the policy is directly related to the physical implementation (i.e. PDP), which dynamically executes and interprets the security relevant operations. Thus, every modification of the PolPA policy represents in a natural way a fault of the PDP component. Thus the idea 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 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 which have to be applied to evaluate an access, thus the policy defines the input domain and its evaluation. From this, the idea 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, by applying the parsing to 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 implementation and to highlight the misbehavior of the PDP implementation. Note that, for the moment, in this scenario we do not consider in the fault model problems related to additional interactions among authorization system components because we focus mainly on the possible faults of the PDP. Detail about the framework are in Section IV.

III. PolPA Authorization System

PolPA is a process algebra based language that allows to write history based security policies according to the Usage Control (UCON) model [1]. The UCON model [5], [6] 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 security policy to grant the initial access to the resource could change while the access is in progress in a way such that the access right does not hold any more. 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 behaviour, 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 order they appear in the policy if they are related to the same seq composition operator. Two or more actions can be executed alternatively or in parallel if they are related to a or or par composition operator, respectively. Moreover, PolPA allows to specify some predicates involving action’s parameters and user’s and resource’s attributes, that need to be satisfied in order to proceed with the action execution. For a detailed description of PolPA language refer to [1].

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. The tryaccess(s,o,r) command is sent by the PEP to the PDP when the user tries to execute a security relevant action. 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. 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 every time that 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, and it could invoke the PEP to terminate this access through the revokeaccess(s,o,r) command. This is a main novelty of the UCON model with respect to prior access control work, where the PDP is usually only passive. To enforce the revokeaccess(s,o,r) command, the PEP should be able to interrupt an action that is in progress.

### A. Example of PolPA Policy

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

The policy defines a sequence of actions where action A must be firstly executed, then either action B or C can be executed, and finally action D can be executed. Line 1-4 of the policy regulate the execution of action A on resource R1, imposing a control on the value of parameter $x_1$, and requiring that action A terminates (line 3) before allowing...
the execution of the other actions. Lines 5-14 allow the alternative execution of action B or action C on resource R2. 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, the only constraint that is required is on the parameter x4 of the action itself. In both cases, the action must be finished before a new one can be started; this is represented by the endaccess commands in lines 7 and 12. Finally, lines 15-19 allow the execution of action D on resource R1, again imposing a constraint on the value of action’s parameter x6. This policy does not require that the actions are executed by the same user.

This work is aimed at testing the PDP implementation, i.e. verifying that the PDP actually enforces the security policy that it gets as input. This means that the testing framework emulates a possible PEP by issuing the tryaccess and endaccess commands to the PDP, and by checking the returned answers. We assume that the input policy is correct, i.e., that does not contain errors or conflicting rules and expresses the security requirements of the resource owner who wrote it.

IV. TESTING FRAMEWORK

The derivation of adequate test cases is one expensive task in software testing, especially for software systems whose inputs have a complex structure such as requests for PolPA authorization systems. These requests consist of a variable number of commands with their parameters interrelated according to a set of rules specified in the policy. Combinatorial approaches deriving all combinations of commands and parameters values of a PolPA policy are not suitable since they do not specifically address the order in which the request commands are submitted to the PDP. Manually preparing such test cases is expensive, difficult and error-prone. We propose in this paper a test case generation technique, involving the design of a fault model along with the corresponding set of mutation operators, and the application of them for generating test cases. In this section we describe the testing framework implemented for the test case derivation starting from a fault model. As explained in Section II, the faults considered are expressed in terms of changes applicable to PolPA policies (see Section IV-A), so that modified policy versions can be automatically derived and used for testing purposes. From each available policy, the testing framework automatically derives the set of test cases according to a methodology able to parse the policy itself and extract the useful information (see Section IV-B). Specifically, the testing framework consists of the following components (see Figure 2):

- **Fault Model Manager (FMM)**. This component manages a predefined collection of possible types of faults that can occur during the evaluation of a PolPA policy due to incorrect control manipulation or violation of specific conditions.
- **Policy Test Set Manager (PTSM)**. This component collects the set of PolPA policies useful for testing purposes. The policies 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 for highlight the potentialities of the PDP and exercising 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 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)**. For each of the available policies (i.e. the policy and its faulty versions) this component automatically derives the test cases in terms of (sequence of) access requests.
- **Test Driver (TD)**. This component coordinates the test cases execution. Collaborating with the TCG it selects one by one the available test cases and, by simulating the PEP behavior, transforms the test case in the opportune tryaccess, and endaccess commands. If necessary, before sending the tryaccess to the PDP, the TD interacts also with the PIP to make available the attributes values necessary for the test case execution.
- **Test Oracle (TO)**. This component is the responsible of the collection of the PDP responses (permitaccess, revokeaccess or denyaccess) 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. It is important to specify that in the current implementation of this component the correct PDP verdict is not auto-

### Table I

**EXAMPLE OF POLPA SECURITY POLICY**

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(tryaccess(user_id,R1,A(x1,x2)).</td>
</tr>
<tr>
<td>2</td>
<td>([sequal(x1,&quot;val1&quot;)],permitaccess(user_id,R1,A(x1,x2)).</td>
</tr>
<tr>
<td>3</td>
<td>endaccess(user_id,R1,A(x1,x2))</td>
</tr>
<tr>
<td>4</td>
<td>)</td>
</tr>
<tr>
<td>5</td>
<td>(tryaccess(user_id,R2,B(x3)).</td>
</tr>
<tr>
<td>6</td>
<td>([sequal(attr(role,user),&quot;admin&quot;)],permitaccess(user_id,R2,B(x3)).</td>
</tr>
<tr>
<td>7</td>
<td>endaccess(user_id,R2,B(x3))</td>
</tr>
<tr>
<td>8</td>
<td>)</td>
</tr>
<tr>
<td>9</td>
<td>or</td>
</tr>
<tr>
<td>10</td>
<td>(tryaccess(user_id,R2,C(x4,x5)).</td>
</tr>
<tr>
<td>11</td>
<td>([iequal((x4,&quot;val2&quot;)),permitaccess(user_id,R2,C(x4,x5)).</td>
</tr>
<tr>
<td>12</td>
<td>endaccess(user_id,R2,C(x4,x5))</td>
</tr>
<tr>
<td>13</td>
<td>)</td>
</tr>
<tr>
<td>14</td>
<td>)</td>
</tr>
<tr>
<td>15</td>
<td>(tryaccess(user_id,R1,D(x6)).</td>
</tr>
<tr>
<td>16</td>
<td>([iequal(x6,&quot;val3&quot;)],permitaccess(user_id,R1,A(x6)).</td>
</tr>
<tr>
<td>17</td>
<td>endaccess(user_id,R1,D(x6))</td>
</tr>
<tr>
<td>18</td>
<td>)</td>
</tr>
<tr>
<td>19</td>
<td>)</td>
</tr>
</tbody>
</table>
matically derived. We are evaluating and defining possible solutions.

In the rest of this section we detail the defined Fault Model and the Test Case Generator component.

A. Fault Model

The fault model is specifically conceived for PolPA language. The approach is inspired by mutation testing techniques but differs from them since mutation operators are used not for test adequacy measurement and analysis 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 can be encountered 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. In the following a detailed description of the classes:

Change Composition Operator (CCO). This class implements a violation of the order of execution of the commands sent by the PEP (tryaccess and endaccess); it is implemented by changing the composition operator. Specifically, let CO be the bag of composition operators (·, or, par)2 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 two.

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 commands (tryaccess and endaccess) included in the policy, \( \forall c_i \in C \), change \( c_i \) with each \( c_j \) such that \( c_j \in C \setminus \{c_i\} \). The number of mutants derived by this class is equal to \( n \times n - 1 \) where \( n \) is the cardinality of C.

Change Guard String Predicate (CGSP). This class implements a wrong management of the values of string parameters; it is implemented changing the predicate involving string parameters. Specifically, let S be the bag of PolPA predicates involving string parameters (sequa, 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 predicates involving string parameters that are in the policy times two.

Change Guard Integer Predicate (CGIP). This class implements a wrong management of the values of integer parameters; it is implemented changing the predicate involving integer parameters. Specifically, let I be the bag of PolPA predicates involving integer parameters (iqual, 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 predicates involving integer parameters that are in the policy times two.

The fault model is used by the Faulty Policies Generator for faulty policies derivation. As an example, applying the mentioned mutation operator classes to the policy of Table I, we derive seventy-eight faulty policies. Specifically, applying the CCO class we derive fourteen faulty policies because there are in the policy six seq and one or composition operators related to the tryaccess and endaccess commands. Applying the CC class we obtain fifty-six faulty policies, because there are four actions (A, B, C, D), each one having a tryaccess and endaccess commands. Finally, applying the CGSP class to the two string predicates (line 2 and 6) that are in the policy we obtain four faulty policies whereas applying the CGIP class to the two integer predicates (line 11 and 16) we derive four faulty policies.

B. Test Cases Generator

The Test Cases Generator implements a testing procedure specifically conceived for a PolPA policy and for each faulty policy derived from it. The test strategy consists in 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 where parameter values are set according to predicates of each command;

(iii) visit depth-first of the tree. For each node k derive the multi set \( MS_k = \{S^1, \ldots , S^n\} \), 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 \( MS_i \) is the multi set derived for the node i that is child of node k; if the node k is the composition operator, and \( MS_{13} \) and \( MS_{12} \) are the multi sets associated to \( il \) and \( i2 \) that are children of k, \( MS_k \) contains all the ordered combinations of the elements of \( MS_{13} \) and \( MS_{12} \); 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 \( S^i \) and the associated parameters values.

V. EMPIRICAL EVALUATION

In this section we detail the application of the proposed framework to the exploratory policy of Table I. 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 have been substituted by our testing framework, the PAP an PIP components are supposed correct and the SUT is the PDP implementation exploited in [1]. The oracle, that is able to recognize if the PDP responses are correct or not, is included in the testing scenario and relies on a predefined verdicts table which collects the detailed set of couples (test case / (set of) expected response(s)).

Following the steps described in the previous section, by the application of the fault model to the policy of Table I, the FPG component derived 78 mutated policies (as motivated at the end of Section IV-A). We report the number of derived mutated policies for each mutant class in the second column of Table II. 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 IV-B. Specifically, TCG generated 2 test cases from the original policy and 45, 112, 8 and 8 test cases from the policies mutated according to the CCO, CC, CGSP, and CGIP mutant class respectively (see Table II third column) for a total of 175 test cases. It is worth noticing that, when the test strategy described in Section IV-B 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 30 over the 175 derived, where 28 are in the set derived from the policies mutated according to CC mutant class and 2 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 II reports the comparison results in the last column. We see that all the responses obtained from the PDP were the same of those expected, unless 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 9 over the 112 test cases executed, the responses obtained were not the expected ones. These 9 test cases do not include redundant ones. In particular, we noticed that the faults were always detected by the test cases derived by the mutants having two \( \text{tryaccess} \) commands involving the first action allowed by the policy (i.e., action A in the policy of Table I).

This behavior is not compliant with what explicitly specified in the policy of Table I, because after the first request for executing action A, represented by the \( \text{tryaccess} \) command in line 1, the PDP only allows the corresponding \( \text{endaccess} \) command in line 3, that represents the end of action A. Thus, the detected anomalies in the test case responses 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 because the tests where the user tries to execute the first action of the policy for the second time returned a result that wasn’t the expected one. Hence, the generic release of the PDP won’t implement this feature.

Although simple, this exploratory example represents a real application context and the fault discovered an important limitation of the considered PDP implementation. Thus, the preliminary obtained testing results confirmed the effectiveness of the proposed testing framework and provided interesting hints for future researches.

VI. RELATED WORK

As surveyed by [7], in literature several works propose fault models, based on FSM or LTS, for test case derivation. Other approaches, as in [8], apply mutation operators to a set of test cases to generate a larger test suite. In the context of access control systems, common proposals, as for instance [9], define a fault model for XACML access control policies even if the mutation operators are used to evaluate coverage criteria and fault-detection capabilities and not for test cases derivation as in our approach. Other approaches, as for instance the X-CREATE framework [10],
focus on the application of the combinatorial approaches to XACML policies values for generating test inputs. However, combinatorial approaches are not suitable for testing the potentialities of the PolPA policies because they do not deal with the semantic of the commands composition. Model-based testing has also been investigated for policy testing, e.g. [11], even if mainly for the generation of abstract test cases to be then refined into concrete requests for the PDP. Alternatively, the Cig [12] approach applies change-impact analysis for test cases generation starting from policy specification.

One of the closest works to our proposal is that of [13]. 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. Thus the protocol model is modified in order to violate a security property, the traces of the mutated model are collected, and the counter-examples for the specified security policy 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 implement the generation of the mutants and the test cases and the execution of the test cases on the PDP.

VII. DISCUSSION AND CONCLUSIONS

In this paper we proposed a testing framework for the automated generation of the test cases from a PolPA policy specification for the testing of the PDP implementation. The framework implements a test strategy based on a specific fault model which focuses on the main problems of policies implementations in the authorization systems. We also presented an exploratory example of the framework application for testing a real PDP implementation.

The preliminary conclusion we can draw from this initial evaluation is that the automatic tests generation guided from the fault model represents a valid contribution for detecting behavioral problems of a PDP implementation due to the specific context of application. Of course, such conclusion must be taken in light of the threats to validity of the performed experiment. The presented study is limited to a simple real policy, then further experimentation would be required. The fault model considered is just a subset of the possible mutation operators, thus further classes can be conceived and implemented in the testing framework. Finally, the expected results are currently not automatically derived requiring additional labor from the user of the testing framework. Except for possible faults in our framework implementation, for the rest all steps have been carried out in automated way, so we do not see other internal threats.

We are currently working in several future directions: developing a methodology for automatically derive the expected results from the PolPA policy specification; increasing the mutation classes considered in the fault model; reducing the number of redundant test cases keeping the same test effectiveness; defining a set of PolPA policies that can be used as conformance test suite for the various application domains; testing other peculiar aspects of the PolPA language (e.g., access revocation).

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