Testing Obligation Policy Enforcement using Mutation Analysis

Yehia Elrakaiby
Security, Reliability and Trust Interdisciplinary Research Center, SnT
University of Luxembourg
Luxembourg
Email: yehia.elrakaiby@uni.lu

Tejeddine Mouelhi
Security, Reliability and Trust Interdisciplinary Research Center, SnT
University of Luxembourg
Luxembourg
Email: tejeddine.mouelhi@uni.lu

Yves Le Traon
Laboratory of Advanced Software SYstems (LASSY) and Security, Reliability and Trust Interdisciplinary Research Center, SnT
University of Luxembourg
Luxembourg
Email: yves.letraon@uni.lu

***Abstract*—The support of obligations with access control policies allows the expression of more sophisticated requirements such as usage control, availability and privacy. In order to enable the use of these policies, it is crucial to ensure their correct enforcement and management in the system. For this reason, this paper introduces a set of mutation operators for obligation policies. The paper first identifies key elements in obligation policy management, then presents mutation operators which injects minimal errors which affect these aspects. Test cases are qualified w.r.t. their ability in detecting problems, simulated by mutation, in the interactions between policy management and the application code. The use of policy mutants as substitutes for real flaws enables a first investigation of testing obligation policies in a system. We validate our work by providing an implementation of the mutation process: the experiments conducted on a Java program provide insights for improving test selection.**

I. Introduction

The use of a policy-based approach to specify system requirements is getting more and more popular [1], [2], [3], [4], [5], [6]. This is because policies enable the uniform expression and the dynamic update of requirements as well as their formal analysis. Security policies specify controls over actions that system users take on system resources. A security policy typically consists of permissions, prohibitions, obligations and dispensations. Permissions (prohibitions) enable the specification of access control requirements. Obligations (dispensations) allow the specification of actions that users should take to fulfill the responsibilities associated with their roles. Obligations also enable the specification of usage control requirements [7], i.e. the necessary actions related with particular accesses. For instance, access to a meeting could be conditional to acceptance of the terms of an access agreement or the use of a pay phone could be conditional to the continuous payment of the usage cost.

When policies are used to control user actions, it is essential to ensure their correct enforcement in the system. Several works have studied the verification of access control policies [8], [9], [10]. Mutation analysis was applied in the context of XACML policies [11] and OrBAC policies [12]. Generic mutation operators for access control policies have been also proposed in [9]. In this paper, we aim at studying obligation policies. In particular, we propose the use of mutation analysis to qualify tests that are used to check the correct behaviour of a system when the specification of this system includes obligations.

To qualify tests that check a system with obligations, we propose a set of mutation operators. These operators are chosen to assess the ability of a test case to detect potential problems in the management of obligations. More specifically, we consider the following policy management aspects: entity assignments to abstract policy concepts such as roles, hierarchies, context management (a context defines the application conditions of a rule), obligation rule enforcement and rule coverage. For each of these aspects, we define a number of mutation operators that inject minimal errors into them. Thus, the inability of a test case to detect mutants would reveal its incapacity to detect policy enforcement errors. The main advantage of our approach is that it considers policy management aspects that are found in most policy languages. Therefore, they can be easily adapted to be used with other policy languages than the one considered in this paper [13].

To validate our work, we present an implementation of an application, which includes obligations. The architecture of the application that we use separates the application code and its core functionalities from the policy management module, which updates security rules in the policy when a change is detected in the application. We generate mutant policies using the mutation operators that we introduce in this paper and then use test cases to study their ability at detecting the generated mutants. We present the obtained results and discuss their relevance to the testing of obligation policies.

This paper is organized as follows. Section 2 introduces a running example that will be used through the paper to illustrate both the obligation rules and the mutants. Section 3 then introduces the obligation model considered in this paper and discusses policy management elements. Section 4 introduces the set of mutation operators that we introduce to simulate policy management errors. Section 5 discusses the implementation of our policy manager and its integration with

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.157

the application code. Section 6 presents the results of the use of our methodology to the application example in Section 2 and discusses some of the lessons learnt. Finally, Section 7 overviews related works and Section 8 concludes the paper.

II. Running Example

As a running example, we consider a virtual meeting management system (VMS) providing web conference services. The virtual meeting server allows meetings to be organized on a distributed platform. When connected to the server, a user can enter or exit a meeting, ask to speak, speak, or plan new meetings. Each meeting has an owner. The owner is the person who plans the meeting and sets its main parameters (such as its name, its agenda, etc.). The owner appoints a moderator for the meeting. The moderator gives the floor to participants who may ask to speak. This application contains 6070 lines of code in 134 classes.

We list below some of the application's requirements.

- R_1: Only the moderator of the application may give the floor to a user to speak.
- R_2 The users have to accept the usage terms of the meeting before joining the meeting.
- R_3: If the speaker does not push a button to keep the floor, access should be revoked.
- R_4: After the speaker finishes, s/he should send a note including the points that s/he raised in his/her talk if the meeting type is standard.

The requirement R_1 specifies an access control rule, namely a permission. The requirements R_2, R_3, and R_4 specify obligations that represent a pre-, ongoing and post-usage requirements respectively. More precisely, they specify the actions which should be taken before, while and after certain accesses respectively. In this paper, we assume that access control is enforced by the application and we only focus on the management and testing of obligation policies.

III. Obligations for Usage Control

Obligations allow the expression of different requirements. We focus on usage control, i.e. obligations that need to be fulfilled before, while and after access. For this reason, we consider the usage model depicted in figure 1. In this model, an access may assume one of the following states {*idle, requested, accessing, pre*}. The state of an access changes when relevant actions to this access are taken as shown in figure 1. The actions {**request, end**} are user-actions taken to request and end access respectively. Other actions {*allow, deny, cond_allow, revoke*} are system-actions used to enforce usage control. In particular, these actions represent access acceptance, denial, conditional access acceptance (i.e. some obligations should be fulfilled before access is allowed) and access revocation respectively. In this paper, we assume that every access is associated with the usage control model in figure 1 for simplicity. For instance, an action $speak$ will in fact correspond to two actions $request(speak)$ and $end(speak)$. Note that an atomic action a with no duration can be simulated by the immediate execution of the action $end(a)$ after $request(a)$. For instance, an action $send$ can be simulated by the consecutive execution of $request(send)$ and $end(send)$.

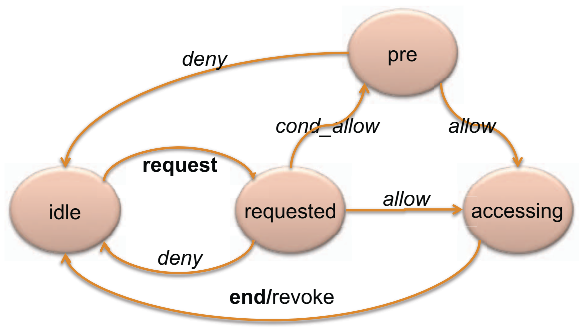

Fig. 1. Usage Control Model

A. Policy Language

To specify and manage obligations, we consider the policy language and model introduced in [13]. This language is a sorted first-order language having finite sorts of subjects (S), objects (O) and actions (A). These sorts represent concrete entities in the system. The language also includes sorts for contexts ($\mathcal{C}$), security rule identifiers ($\mathcal{N}$), roles ($\mathcal{R}$), activities ($\mathcal{T}$) and views ($\mathcal{V}$). These latter sorts denote rule conditions, rule identifiers and groups of subjects, actions and objects respectively.

User-role, action-activity and object-view assignments are specified in the language using the relations *empower(Subject,Role)*, *consider(Action,Activity)* and *use(Object,View)* respectively. For instance, $empower(s,r)$ specifies that the subject s is empowered into the role r. The concrete entities in the language may have attributes. These attributes are described using application-dependent predicates.

The language also supports the definition of role, activity and view hierarchies [14]. A role, activity and view hierarchy is a partial order relation over the sets of roles, activities and views respectively. Hierarchies are specified using $sub\text{-}role(Role_1, Role_2)$, $sub\text{-}activity(Activity_1, Activity_2)$, $sub\text{-}view(View_1, View_2)$.

Note that assignments of concrete entities to more abstract concepts such as roles and hierarchies are key concepts in many policy languages such as [15], [16], [17].

B. Obligation Specification

Our obligation rules have the following form.

$$Obligation(N,SR,AA,OV,Ctx_a,Ctx_v)$$

where *N* is a unique security rule identifier, *SR* is a subject or a role, *AA* is an action or an activity and *OV* is an object or a view, *Ctx*$_a$ and *Ctx*$_v$ are context expressions. *Ctx*$_a$ is called the obligation's *activation context* and *Ctx*$_v$ is called its *violation context*. A context represents a set of conditions that is relevant to the management of obligations: the activation context defines the conditions which while true the fulfillment of obligations is required. On the other hand, the violation context specifies when the obligation is violated.

a) Context Specification: We consider two types of contexts, namely state-based and event-based contexts. State-based contexts define conditions on the state which have to be true for a context to be true (hold). They are expressions of the following form[1].

$$Hold(S, A, O, Ctx) \ \leftarrow \ f_1, ..., f_n$$

The context rule above states that the context *Ctx* holds for the subject, action and object *S,A,O* while the conditions f_1,..., f_n are true.

For instance, we may specify a state-based context *in_meeting* that holds for a subject S when S is a participant in a meeting M and the status of M is *ongoing*.

$$Hold(S, A, O, in\_meeting) \leftarrow$$
$$Meeting(M, status, ongoing), Meeting(participants, S)$$

Event-based contexts enable the definition of contexts in terms of the actions which activate and deactivate them. These contexts are specified using expressions of the following form.

$$Hold_e(S, A, O, start/end(Ctx))$$
$$\textbf{after}\ do(S, A, O) \quad \textbf{if}\ p_1, ..., p_n$$

Every state-based *Ctx* is associated with two event-based contexts: An event-based context *start*(*Ctx*) which represents the moment at which *Ctx* begins to hold, and another event-based context *end*(*Ctx*) to denote the moment at which *Ctx* ceases to hold. For instance, consider the state-based context *in_meeting* just defined. An event-based context *start*(*in_meeting*) holds at the moment when *in_meeting* becomes true. The event-based context *end*(*in_meeting*), on the other hand, holds when *in_meeting* becomes false (when the subject *S* exits the meeting or when the meeting is ended).

b) Context Language: Our context language allows the composition of contexts using the logical operators conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. The language also supports the definition of contexts in the form of an interval $[C_1, C_2]$. An interval context holds from the moment the context C_1 is true until C_2 is true. The context language is defined by the following Backus-Naur Form (BNF) grammar.

$$\mathcal{CE}_S ::= \top \mid \bot \mid C_S \mid \mathcal{CE}_S \,\&\, \mathcal{CE}_S \mid \mathcal{CE}_S \oplus \mathcal{CE}_S \mid \neg\mathcal{CE}_S \mid [\,\mathcal{CE}, \mathcal{CE}\,]$$
$$\mathcal{CE}_E ::= C_E \mid start(\mathcal{CE}_S) \mid end(\mathcal{CE}_S) \mid \mathcal{CE}_S \,\&\, \mathcal{CE}_E \mid \mathcal{CE}_E \,\&\, \mathcal{CE}_S$$
$$\mathcal{CE} ::= \mathcal{CE}_S \mid \mathcal{CE}_E$$

where C_S and C_E are user-defined state context identifiers and event-based context identifiers respectively. The special context $\top$ which reads true ($\bot$ which reads false respectively) represents a context that always (never) holds.

C. Usage Control Contexts:

Usage control contexts are special fluents (facts whose value may change) that we use at the policy manager level to track the usage state of a resource. For instance, when a fluent $accessing(S, A, O)$ holds, this means that the subject S is executing the action A on the object O. We update these fluents at the occurrence of the actions presented in the model shown in figure 1. We specify the update of fluents using propositions called *effect laws*. For instance, we specify that the fluent *requested* holds for a subject S, action A and resource O after a user S requests an access (denoted by the occurrence of an action $request(S, A, O)$) until the request is granted, denied or conditionally allowed (denoted by the occurrence of one the actions $grant(S, A, O), deny(S, A, O), cond\_grant(S, A, O)$. The rules below show how we specify when the policy manager updates usage control contexts of type *requested*.

$$request(S, A, O)$$
$$\textbf{causes}\ requested(S, A, O)$$

$$cond\_allow(S, A, O)$$
$$\textbf{causes}\ \neg requested(S, A, O)$$

$$deny(S, A, O)$$
$$\textbf{causes}\ \neg requested(S, A, O)$$

Other usage-control contexts, namely pre and $accessing$, are specified similarly using the actions in the usage control model depicted in figure 1.

D. Examples

In this section, we specify the obligations discussed in section II. These examples are specified as follows.

$$Obligation(r_2, users, accept\_terms, meeting,$$
$$join\_requested, delay(2.minutes))$$

$$Obligation(r_3, users, push\_button, meeting,$$
$$speaking, delay(1.minutes))$$

$$Obligation(r_4, users, send\_report, meeting,$$
$$end(speaking) \ \wedge \ standard\_meeting, delay(2.minutes))$$

$$Hold(S, A, O, join\_requested) \leftarrow requested(S, join, meeting)$$
$$Hold(S, A, O, speaking) \leftarrow accessing(S, speak, meeting)$$
$$Hold(S, A, O, standard\_meeting) \leftarrow$$
$$participant(S, O), type(O, standard)$$

where the context $delay(X.timeUnit)$ is a special context that is detected $X.timeUnit$ after the activation of the obligation. Note that an obligation typically has a deadline (a violation context) which specifies the condition before which the obligation should be fulfilled.

The examples above show how usage control contexts simplify the specification of usage control obligations. For instance, the context $speaking$ holds for a subject S while the usage control context $accessing(S, speak, meeting)$ is true. Thus, the use of this context in r_3 activates this obligation while S is speaking. In r_4, the obligation is activated after the person finishes speaking since the context $end(speaking)$ is the activation context of r_4. The context $standard\_meeting$ holds for a subject S and an object (meeting) O if S is one of the participants of O.

[1]The condition part of a rule may be omitted if they are *true*.

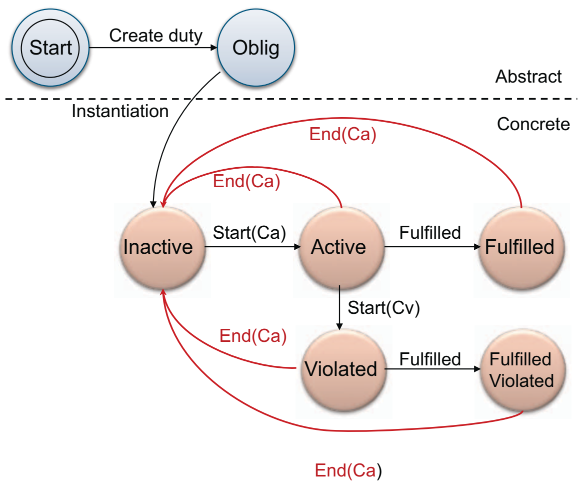

Fig. 2. State Transitions of Individual Obligations

E. *Policy Management*

Figure 2 describes obligation management in terms of change in the state of obligation contexts. Concrete obligations may assume the states inactive (the obligation is not required and may be activated), active (the fulfilment of the obligation is required), fulfilled (the obligation is satisfied), violated (the obligation is violated but is still required) and fulfilled/violated (the obligation is fulfilled but was violated). The end of an obligation is denoted by its return to the inactive state.

An obligation is managed as follows: an obligation is activated when its activation context becomes true. An obligation is fulfilled if its action is taken, it is violated if its violation context becomes true. An obligation is no longer required when it is deactivated, i.e. when its activation context ceases to hold.

Note that we unify the representation of obligation rules in the policy. In particular, we transform all activation contexts of obligations in the policy into interval contexts as follows: when the context is a state-based context C_S, it is represented as a context $[start(C_S), end(C_S)]$. When the context is an event-based context C_E, it is represented as the context $[Ctx, \perp]$. Since the special context $\perp$ is a context that never holds, an obligation which activation context is an event-based context remains required after its activation until it is fulfilled. This means that all obligation rules in the policy have the following form.

$$Obligation(N, SR, AA, OV, [C_a, C_d], C_v)$$

This representation simplifies obligation management and interpretation.

IV. Test Qualification and Mutation Operators

When an application includes an obligation policy, it is necessary to ensure that this policy is correctly managed. In the following, we identify several key aspects that are relevant to policy management and define mutation operators that simulate errors in these aspects. The non-detection of a mutant by a test case would therefore reveal its incapacity to detect problems in policy management. Note that we consider aspects that are commonly found in most policy languages. This enables the reuse of our mutation operators for other policy languages after the making of simple modifications. Table III presents the mutation operators described below. We enumerate our different mutation operators and describe their related policy aspect as follows.

1) Entity Assignments: Tests should reveal problems in subject-role, action-activity and object-view assignments. To verify this, we simulate an error that affects a mapping between one of the application's concrete entities and the policy's abstract concepts. We call this category of mutation operators 'Mapping Operators'. There are three operators in this category, namely the subject-role (SRM), action-activity (AAM), and object-view (OVM) mapping operators. The SRM operator modifies the mapping between a subject and a role by changing the subject who is empowered a role by another subject who is not empowered this role. The AAM and OVM similarly modify mappings between action and activities, and objects and views. We may also consider the simple addition/removal of a subject, action or object to a role, activity or view respectively. Note that this error affects the policy only if the role, activity or view of the assignment is used in at least one security rule. This constraint is also relevant for all mutation operators that inject errors into *intermediate* concepts, i.e. concepts that simplify policy specification, such as hierarchies, as opposed to security rules themselves.
2) Hierarchies: Tests should ensure that security rules apply correctly to their role, activity and view. To check this requirement, our mutation operators inject errors in the definition of hierarchies. In particular, we consider the reduction (HR) and the enlargement (HE) of the scope of application of a security rule. For instance, we may use a sub-role or a super-role of the role of a security rule instead of the specified role. This category of mutation operators is called 'Hierarchy Operators'. Note that the use of this technique requires assuming that there is not an obligation in the policy that is *subsumed* by another rule in the policy; a rule is subsumed by another rule when it corresponds to an equivalent set or a subset of the rules defined by this other rule. In table III, we only show these mutation operators for roles. Operators for activities and views are similarly defined.
3) Context Management: It is necessary to ensure that rule conditions are evaluated correctly. In this paper, we consider contextual policy rules where *contexts* affect the applicability of obligations. To ensure that problems in context evaluation are detected, we define a set of mutation operators that change rule contexts. We call them 'Context operators'. We define three operators in this category, one that reduces the scope of application of the context (CR), one that extends it (CE) and one that

negates it (CN). We extend the applicability conditions of a security rule by using the disjunction operator whereas we reduce the applicability of a rule by using the conjunction operator. For instance, we replace the activation context C_a by the composed context $C_a \vee C_1$, where C_1 is not equivalent to C_a nor it is not equivalent to the context $\perp$, to extend the applicability of the security to when the context C_1 is true. In a similar way, we use the context $C_a \wedge C_1$ to reduce the application of the rule to only when the context C_1 holds. In table III, we only show operators for the activation context. Operators for the deactivation and violation contexts are similarly defined.

4) Obligation Management: Obligations are managed according to the state model in 2. To reveal problems in the enforcement of this model, we introduce 'Context Swap operators'. These operators switch the activation, deactivation and violation contexts of an obligation. For instance, the activation context is swapped with the deactivation or violation contexts of the obligation. The intuition behind this operation is that it simulates potential problems in the management of obligations, i.e. the incorrect update of obligation states.
5) Rule Deletion: Tests should be able to detect that an obligation rule is missing. This ensures rule coverage, i.e. that the test case verifies all rules in the policy. This mutation operator is called 'Rule Deletion operator'. The mutant policy in this case is the original after the removal of one of its security rules.

Regarding the equivalent mutants issue, most of our operators do not generate equivalent mutants because the injected errors lead to a policy that is always different from the initial policy. It is true for all operators except the context management operators. In fact, only these operators can generate equivalents mutants in certain case where the changed context is equivalent semantically to the initial context. However, we always check that the context is always different and that it is not equivalent to the initial context. More generally, it is possible to detect equivalent mutants by comparing the initial policy to the mutated policy. The comparison should focus on contexts since they may lead to the same behavior (when evaluated).

V. Implementation

Figure 3 shows an overview of the system implementation and how the obligation policy interacts with the application. In our architecture, the obligation policy manager mechanism is implemented in a non-intrusive way separately from the application core functions. This follows the separation of concerns principle and allows the simplification of both the application code and the policy management module. It also allows the separate verification and testing of each separately. In this paper, we focus on the testing of the application code and the policy manager combined.

The obligation policy manager is composed of two main components:

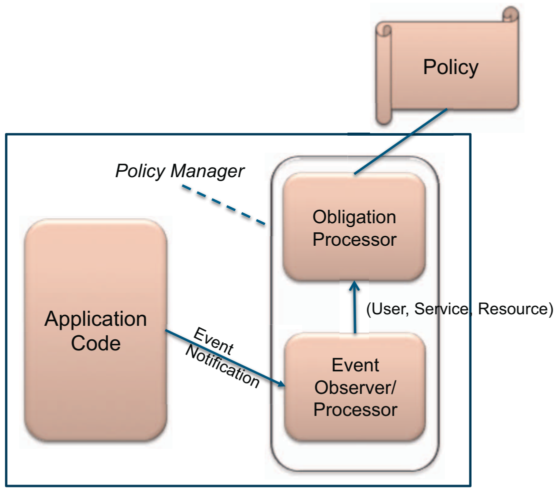

Fig. 3. Overview of the System

- The event observer/processor: Observes policy-relevant events in a non-intrusive way using aspect oriented programming (AspectJ). These events correspond to service calls. The events are then processed and their subject, action and resource are extracted. The obligation processor is then notified to modify the state of the policy accordingly.
- The obligation processor: receives notifications of service executions and updates the state of rules in the policy accordingly. More precisely, given the user, the service and the resource, the obligation processor uses the mapping between the system concepts and the policy to obtain their corresponding role, activity and view. Then, it computes the context of security rules in which these elements are used and their state is updated if necessary, i.e. they are activated, deactivated, violated or fulfilled. Note that the obligation processor is associated with a policy and the policy is managed according to this specified policy. This allows the dynamic update of the set of obligations enforced in the system, i.e. when rules in the policy are modifications, the modification is automatically taken into account by the obligation policy manager.

In our implementation, we reuse the Virtual Meeting System which we have used in previous work. This system is implemented in Java and enforces an access control policy defined using an RBAC policy language. This has simplified our implementation since some of the concepts needed, namely roles, activities and views and their mapping to system concepts were already implemented in the system.

c) Mutation Process: The mutation process proceeds as follows. The mutants are first created by mutating obligation rules in the policy as described in the previous section. The existing obligation policy is then replaced by mutant policies. Test cases that are created or generated to validate the implementation are run against the mutated versions of the policy. A mutant is detected when a test case fails. Note that

Operator	Original Policy ($\mathcal{P}$)	Mutated Policy ($\mathcal{P}_m$)
SRM	$\{..., empower(S,R),\ obligation(N,R,A,V,[C_a,C_d],C_v),$ $subject(S'), \neg empower(S',R), ...\}$	$\mathcal{P} \setminus \{empower(S,R)\} \cup \{empower(S',R)\}$
AAM	$\{..., consider(T,A),\ obligation(N,R,A,V,[C_a,C_d],C_v),$ $action(T'), \neg consider(T',A), ...\}$	$\mathcal{P} \setminus \{consider(T,A)\} \cup \{consider(T',A)\}$
OVM	$\{..., use(O,V),\ obligation(N,R,A,V,[C_a,C_d],C_v),$ $object(O'), \neg use(O',V), ...\}$	$\mathcal{P} \setminus \{use(O,V)\} \cup \{use(O',V)\}$
RHE	$\{..., sub\text{-}role(R,R_1),$ $obligation(N,R,A,O,[C_a,C_d],C_v), ...\}$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R_1,A,O,[C_a,C_d],C_v)\}$
RHR	$\{..., sub\text{-}role(R_1,R),$ $obligation(N,R,A,O,[C_a,C_d],C_v), ...\}$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R_1,A,O,[C_a,C_d],C_v)\}$
CAR	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v),\ context(C_1)...\}$ $\wedge C_a \neq C_1\ \wedge C_1 \neq \bot$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[C_a \wedge C_1,C_d],C_v)\}$
CAE	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v),\ context(C_1), ...\}$ $\wedge C_a \neq C_1 \wedge C_1 \neq \bot$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[C_a \vee C_1,C_d],C_v)\}$
CAN	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v), ...\}$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[\neg C_a,C_d],C_v)\}$
CSAD	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v), ...\} \wedge C_a \neq C_d$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[C_d,C_a],C_v)\}$
CSAV	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v), ...\}\ \wedge C_d \neq C_v$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[C_v,C_d],C_a)\}$
CSDV	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v), ...\} \wedge C_d \neq C_v$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$ $\cup \{obligation(N,R,A,O,[C_a,C_v],C_d)\}$
RD	$\{..., obligation(N,R,A,O,[C_a,C_d],C_v), ...\}$	$\mathcal{P} \setminus \{obligation(N,R,A,O,[C_a,C_d],C_v)\}$

TABLE I
Mutation Operators

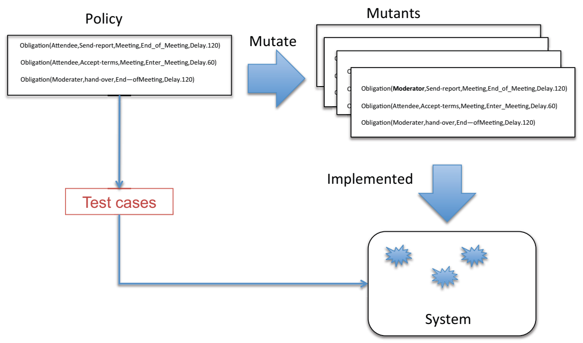
Policy
Obligation(Attendee,Send-report,Meeting,End_of_Meeting,Delay.120)
Obligation(Attendee,Accept-terms,Meeting,Enter_Meeting,Delay.60)
Obligation(Moderater,hand-over,End—ofMeeting,Delay.120)
Mutate
Mutants
Obligation(Moderator,Send-report,Meeting,End_of_Meeting,Delay.120)
Obligation(Attendee,Accept-terms,Meeting,Enter_Meeting,Delay.60)
Obligation(Moderater,hand-over,End—ofMeeting,Delay.120)
Implemented
Test cases
System

Fig. 4. Obligation mutation process

we assume that the state of obligations can be checked (are observable) by test cases that need to verify what obligations are active, violated and fulfilled.

VI. Results

We use the meeting system as a case study for getting preliminary results on mutating obligation policies. We start by explaining the test selection process, then we study the generated mutants and the test results, especially those regarding alive mutants.

A. Test Selection

We had 21 tests for validating all obligations rules. In fact, we ended up with 7 rules that are manually derived from the three obligation rules presented in the previous section. The derivation is done according to the hierarchy for the first two rules (two derived rules for each rule), except for the last one because only the moderator is expected to send the report. We had 21 tests for validating the 7 rules. Each 3 tests target a specific obligation rule and check three scenarios (one for activation, and one for violation and one for fulfilment). Test cases are independent from each others. For instance, the test that checks for rule violation context, starts by initializing the obligation activation context, then validating the expected behaviour in case of violation.

B. Mutation results

The next table shows the number of mutants that have been generated. The first three mutation operators were not implemented due to technical reasons related to the meeting system. For instance, concerning SRM, in the meeting systems users are not stored in the database and therefore, we cannot modify the userŕole and delete his/her mapping with that role. This is the first lesson learnt from this case study: implementing obligation policies - and thus mutation operators - may be complex and system-specific. The implementation of obligations requires weaving monitors at the right locations in the system and collecting dynamically all the events that are relevant to the computing of contexts for each instantiated obligation rule. More precisely, the monitoring is performed by weaving a piece of code that is called when a method is executed (a method related to obligation activation, violation or fulfillment). The method parameters are then extracted to store the value of the user and the other parameters that will then be used to create the event with all its parameters. This event will then be handled to update obligations states. The implementation of obligations is a complex process that is consequently error-prone. Mutation at the obligation level forces the test cases to detect whether this change (mutation in the obligation policy) is propagated into the system. Indirectly, the mutation allows exercising the interactions between the policy manager and the application code.

Mutants	Number
SRM,AAM,OVM	0
RHE	4
RHR	1
CAR	7
CAE	7
CAN	7
CSAD	7
CSAV	7
CSDV	7
RD	7
All	**54**

TABLE II
MUTATION OPERATORS

In the next table, we present the mutation analysis results. It is interesting to analyse the reasons why certain mutants were not killed, namely the single RHR mutant and some CAR and CAE mutants). These alive mutants have in common that they either add new obligation rules or make the scope of an existing rule wider (e.g. applicable to other subjects).

Mutants	Number	Killed	Score
RHE	4	4	100%
RHR	1	0	0%
CAR	7	3	42%
CAE	7	4	57%
CAN	7	7	100%
CSAD	7	7	100%
CSAV	7	7	100%
CSDV	7	7	100%
RD	7	7	100%
All	**54**	**46**	**85%**

TABLE III
MUTATION OPERATORS

Our tests were not able to detect such changes since the test cases just check the initial obligation policy and not rules applicable to other subjects. This is the case for the mutation operator RHR, that replaces a rule with role r_1 by a rule with role r_2 where r_1 is sub-role of r_2. When rules are derived according to these hierarchy rules, we end up with at least two rules (one derived and one newly added rule). Our test selection process, that is straightforward and systematic with regard to the initial policy, we test behaviours related to the roles explicitly associated with an obligation in the initial policy. With such a test selection process, it is unlikely to exercise other behaviours than those specified in the initial policy.

In terms of functionality, testing the implementation of an obligation policy is a two-fold issue that our mutation analysis highlights:

1) test cases must ensure that what is specified for a given role is correctly enforced. Any role that is submitted to an obligation must be monitored, and one must check that the obligation states evolve as expected. All the killed mutants are related to this category of test cases.
2) test cases must also ensure that no other subject is under the control of an obligation rule: this means that the test cases must test unspecified parts of the policy. The alive mutants could only be killed with this second category of test cases.

The selection process that we use for the experiment cannot cover this second case. However, while testing specified cases requires a reasonable effort, it seems costly to generate tests covering all possible wrong assignments of obligations to roles or subjects. This is the second lesson learnt from this experiment: investigating more effective test selection and generation techniques for obligations is a key research issue.

VII. RELATED WORK

Yue Jia's survey [18] lists a relatively small number of papers that are related to security policies and mutation. At the origin of this work are [8], [9], [10] which study testing access control languages. Other studies proposed techniques and tools for testing the PDP implementation of security policies written in XACML [11], [19] and RBAC [20], [21]. Fisler et al. [22] proposes Magrave, a tool for analyzing XACML policies and performing change-impact analysis. In [11], Xie et al. proposed a new tool Cirg that automatically generates test for XACML policies using Change-Impact Analysis. Several researchers generate tests from access control policies using various forms of state machines [12], [23]. In addition, Several studies propose the use of fault-injection or mutation targeting different aspects of security testing. For instance, adaptative vulnerability analysis [24] injects faults to the application data flow and internal variables. The objective is to identify parts of application's code that have insecure behaviour when the state of the application is disturbed. The work that is related to security policy testing was done by Martin et al. [11] who proposed a mutation fault model specific to XACML policies. The same PDP-alone based testing approach is considered by Mathur et al. [20], who also mutate RBAC models by building an FSM model for RBAC and use strategies to produce test suites from this model (for conformance testing). Going along the same lines, Sharma et al. [26] propose the ACPC model (Access Control Policy Checker) which includes mutation operators for comparing the original policy response with the response of mutant policy.

We conclude that all of these papers focus on testing access control policies, whereas a security policy encompasses more than this access control dimension. Going one step further, we investigate how security policies which include obligations should be tested. This justifies this proposal of a mutation-based approach for testing obligation policies, that has, as far as we know, no equivalent today.

VIII. CONCLUSION

Making explicit the rights and duties of system actors is one of the key challenges for nowadays security and privacy. An obligation policy enables the specification of some of theses aspects, but the mechanisms required for managing, monitoring and enforcing it may be complex and depend on the system's nature.

In this paper, we propose an architecture for making testable a system extended with an obligation policy. The policy manager (equivalent to a Policy Decision Point in an access control logic) is responsible for computing the current states of each rule associated to all the actors submitted to an obligation and deciding what actions must be enforced (such as sanctions). Since the interactions between the policy manager and the system may be faulty, we propose a test qualification technique that checks systems enforcing obligation policies. In particular, we identified key elements in the management of obligation policies, then we defined mutation operators that inject errors related to these key elements. A non-detection of these errors would reveal the inability of a test suite to ensure the correct application of the policy. One advantage of our work is that it is relatively generic and can be adapted for testing different policies. We have validated our mutation techniques by implementing obligation policies and their management using a separate policy manager module. We then used our mutation operators to derive mutant versions of the policy and ran test cases and presented the mutation results. First lessons have been learnt from this experiment.

Future work consists of studying more thoroughly the testing of obligation policies. In particular, we intend to develop new techniques to test the policy manager separately from the application and then compare our results with those obtained when the whole security mechanism is tested. In addition, we would like to study the generation of test cases for obligation policies using formal techniques. Finally, the definition of other mutation operators to create more sophisticated mutants for usage control is among the research directions that we intend to pursue.

REFERENCES

[1] A. Pretschner, M. Hilty, F. Schütz, C. Schaefer, and T. Walter, "Usage control enforcement: Present and future," *IEEE Security and Privacy*, vol. 6, pp. 44–53, July 2008.

[2] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter, "Mechanisms for usage control," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ser. ASIACCS '08. New York, NY, USA: ACM, 2008, pp. 240–244.

[3] D. Beimel and M. Peleg, "Editorial: Using owl and swrl to represent and reason with situation-based access control policies," *Data Knowl. Eng.*, vol. 70, pp. 596–615, June 2011.

[4] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 63–.

[5] N. Dulay, E. Lupu, M. Sloman, and N. Damianou, "A policy deployment model for the ponder language," in *Proc. IEEE/IFIP International Symposium on Integrated Network Management (IM2001*, 2001, pp. 14–18.

[6] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert, "A general obligation model and continuity: enhanced policy enforcement engine for usage control," in *Proceedings of the 13th ACM symposium on Access control models and technologies*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 123–132.

[7] J. Park and R. Sandhu, "The $UCON_{ABC}$ Usage Control Model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128 – 174, February 2004.

[8] H. H. Thompson, "Why security testing is hard," *IEEE Security and Privacy*, vol. 1, pp. 83–86, July 2003.

[9] Y. L. Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," in *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ser. ISSRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 93–102.

[10] Y. L. Traon, T. Mouelhi, A. Pretschner, and B. Baudry, "Test-driven assessment of access control in legacy applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 238–247.

[11] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, ser. SESS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 5–.

[12] K. Li, L. Mounier, and R. Groz, "Test generation from security policies specified in or-bac," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, ser. COMPSAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 255–260.

[13] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia, "Formal enforcement and management of obligation policies," *Data Knowl. Eng.*, vol. 71, no. 1, pp. 127–147, 2012.

[14] F. Cuppens, N. Cuppens-Boulahia, and A. Miege, "Inheritance hierarchies in the Or-BAC Model and application in a network environment," *Proc. Foundations of Computer Security (FCS04)*, pp. 41–60, 2004.

[15] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[16] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *LECTURE NOTES IN COMPUTER SCIENCE*. Springer-Verlag, 2001, pp. 18–38.

[17] A. Abou El Kalam, S. Benferhat, A. Miège, R. El Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin, "Organization based access control," *Policies for Distributed Systems and Networks, IEEE International Workshop on*, vol. 0, p. 120, 2003.

[18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions of Software Engineering*, vol. 35, no. 5, pp. 649 – 678, sept-oct 2011.

[19] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance checking of access control policies specified in xacml," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, ser. COMPSAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–280.

[20] A. Masood, "Scalable and effective test generation for access control systems," Ph.D. dissertation, West Lafayette, IN, USA, 2006, aAI3259967.

[21] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur, "Scalable and effective test generation for role-based access control systems," *IEEE Transactions of Software Engineering*, vol. 35, no. 5, pp. 654–668, May 2009.

[22] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 196–205.

[23] W. Mallouli, J.-M. Orset, A. Cavalli, N. Cuppens, and F. Cuppens, "A formal approach for testing security rules," in *Proceedings of the 12th ACM symposium on Access control models and technologies*, ser. SACMAT '07. New York, NY, USA: ACM, 2007, pp. 127–132.

[24] T. M. G. Ghosh, A. K. O'Connor, "An automated approach for identifying potential vulnerabilities in software," in *Proceedings of the IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 104–114.

[25] B. Breech, M. Tegtmeyer, and L. Pollock, "An attack simulator for systematically testing program-based security mechanisms," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 136–145.

[26] S. Sharma, S. K. Jena, and K. Satyababu, "New approach for testing the correctness of access control policies," in *Proceedings of the International Advance Computing Conference (IACC'09)*, Patiala, Punjab, India, 06-07 March 2009.