Testing Delegation Policy Enforcement via Mutation Analysis

Phu H. Nguyen, Mike Papadakis and Iram Rubab
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg,
4, rue Alphonse Weicker, L-2721 Luxembourg

Email: {phuhong.nguyen, michail.papadakis, iram.rubab}@uni.lu

***Abstract*—Delegation is an important dimension of security that plays a crucial role in the administration mechanism of access control policies. Delegation may be viewed as an exception made to an access control policy in which a user gets right to act on behalf of other users. This meta-level characteristic together with the complexity of delegation itself make it crucial to ensure the correct enforcement and management of delegation policy in a system via testing. To this end, we adopt mutation analysis for delegation policies. In order to achieve this, a set of mutant operators specially designed for introducing mutants into the key components (features) of delegation is proposed. Our approach consists of analyzing the representation of the key components of delegation, based on which we derive the suggested set of mutant operators. These operators can then be used to introduce mutants into delegation policies and thus, enable mutation testing. A demonstration of the proposed approach on a model-driven adaptive delegation implementation of a library management system is also provided.**

***Keywords*-Mutation Analysis; Security Testing; Model-Based Testing; Access Control; Delegation**

I. Introduction

In the field of access control, delegation is a very complex but important aspect that plays a key role in the administration mechanism [1]. Delegation is a process of delegating access rights from one user (called delegator) to another user (delegatee). The management and enforcement of a delegation policy is crucial because delegation can be considered as administrative mechanism of access control. In a process of delegation, user gets exceptional capabilities to act on behalf of other users. Any discrepancy in delegation can cause a malicious user to get access to the protected resources of the system. It may cause privacy issues in case the data is used without the consent of authorized user. Therefore the enforcement of delegation in the system should be free from any disparity. Testing is a way to get confidence on the correctness of security policies enforcement, including delegation policies enforcement.

Software Testing aims at finding errors by executing tests against the flaws of the system. It is a way to establish confidence on the correctness of the system behavior and to ensure its quality. Although work is being done using formal verification [2], [3] and static/dynamic program analysis based techniques [4], testing approaches are still in need. As formal verification identifies design flaws but, some of the underlying implementation defects still remain undetected. To this end, some work have been done on testing access control policies [5], [6], [7], [8]. However, none of these works aims at testing delegation policies. This forms the main issue addressed by the present paper.

Mutation analysis [9], [10] is a well established technique supporting various software development activities like testing [11] and debugging [12]. It operates by introducing artificial defects called mutants into the artifacts of the program under investigation [13]. Its requirement is to design test cases capable of revealing these defects. Researchers has shown that mutants despite being artificially introduced behave quite similarly to real faults [14]. Thus, the benefits of their use are evident. Mutation testing has been applied on XACML policies [15] and OrBAC policies [16]. Additionally, generic policy mutants and Obligation specific mutants have also been proposed in [17] and in [18], respectively. In this lines, the present paper proposes the use of mutation analysis for testing the behavior of a system with respect to its delegation policy. This practice provides an effective way to specialize the testing process to the delegation enforcement mechanism.

Delegation policy enforcement has certain testing challenges that makes its testing task interesting and hard. It requires verifying the mapping between different delegation elements such as delegator, delegatee and role or permission. Additionally, since delegation is often context dependent, it is vital to ensure that a specific delegation rule is enforced correctly in its specified context. Moreover, advanced (complex) delegation features (like monotonicity, temporary delegation, multiple delegation, multi-step delegation, etc.) pose further difficulties to the testing process. However, as a key role in the administration mechanism of access control, the more complex delegation features that a system supports, the easier (and more secure) its administration task is. Thus, testing delegation policy enforcement has to take into account various advanced delegation features that this paper is dealing with.

Testing of a delegation policy is the process of ensuring the correct enforcement of its delegation rules in a system. Mutation analysis for delegation policy involves creating mutants of a policy by injecting defects into the delegation rules of the policy. The system implementation

is then checked against the enforcement of the mutated policy versions. In view of the request-response nature of a delegation, policy mutants can be recognized as killed or live. In this paper, we introduce the issues of testing delegation policy enforcement. Ultimately, we suggest the use of mutation testing for their effective test. Briefly, our main contributions consist of 1) a formal specification of access control and delegation policy supporting advanced delegation features; and 2) a set of mutant operators derived from the process of analysing different aspects of delegation.

The rest of this paper is organized as follows. Section II illustrates the aspects of testing a delegation policy via a running example. Some background material is given in Section III. In Section IV the notations of delegation model, context and features are introduced. In Sections V and VI the proposed set of mutant operators and a demonstration example of their application are given respectively. Finally Sections VII discuses some related work and Section VIII summarizes the paper.

II. A Running Example

This section presents a library management system (LMS) providing library services. There are two types of user accounts in the LMS: personnel accounts such as director, secretary, administrator and librarian, are managed by administrator; and borrower accounts like lecturer and student that are managed by secretary. A secretary can add new books in the LMS when they are delivered. The director of the library has the same accesses as the secretary, but additionally, he can also consult the personnel accounts. The librarian can consult the borrower accounts. Lecturers and students can borrow, reserve and return books, etc.

The LMS implementation details can be found in [19]. In this paper we describe some delegation situations as follows to illustrate our approach. We refer to these delegations as delegation 1, 2 and 3.

1) Bill (*director*) delegates his permission of consulting personal account to a secretary (*Bob*).
2) Alice (*secretary*) transfers her permission of creating borrower account to a librarian (Jane). Alice cannot use this permission while delegating it.
3) Bill (*director*) delegates his permission of consulting personal account to a secretary (*Bob*) during his vacation (the delegation is automatically activated at the start of his vacation and revoked at the end of his vacation).

These delegation situations also will be used further in testing their correct enforcement via mutation analysis.

III. Background

A. Access Control

Access Control aims at securing a software system by controlling the access of users (or processes, components, etc.) to the resources (e.g. files, methods, etc.) of the system. Every access is controlled through the enforcement of an access control policy, which is composed of a set of rules for allowing or denying the access to the system resources, in different contexts.

Definition 1 (Access Control): Let U be a set of users, P be a set of permissions, and C be a set of contexts. An access control policy AC is defined as a user-permission-context assignment relation: $AC \subseteq U \times P \times C$. A user u is granted permission p in a given context c if and only if $(u, p, c) \in AC$. Additional details about contexts are given in Section IV-B.

B. Delegation

In the field of access control, delegation is a very complex but important aspect that plays a key role in the administration mechanism [1]. By supporting delegation, a system allows its users to grant some authorizations, even though those users do not have any specific administrative privileges. There are two types of authorization can be delegated: right and obligation. This paper studies right delegation, further work will be dedicated to obligation delegation. In the rest of this paper, "delegation" is understood as "right delegation".

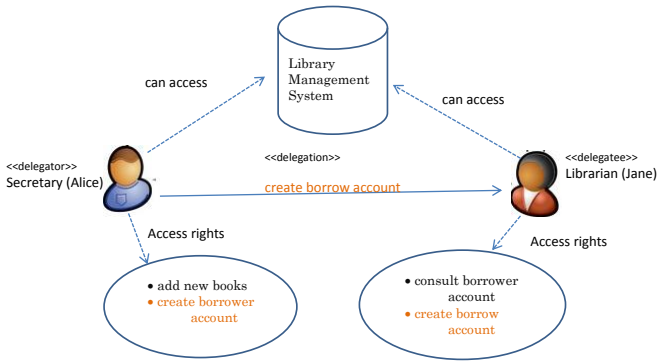

Figure 1. Delegation Process Example

So, delegation is the process of delegating access rights (permissions) from one subject i.e. a delegator to another subject i.e. a delegatee. The basic concept of delegation is presented in Figure 1. In this example, Alice (secretary) and Jane (librarian) have access to the resources of LMS with their personal rights, i.e. Alice as a secretary can add new books, and create borrower account, while Jane as a librarian can consult borrower account. For some reason, Alice wants to delegate her permission of creating borrow account to Jane. As described by the arrow from Alice to Jane in Figure 1, Alice (delegator) is delegating the permission of "create borrower account" to Jane (delegatee). Once Alice has been delegating this permission to Jane, it is added to the access rights of Jane. By this delegation, Jane can create borrower account while being delegated by Alice on this permission.

IV. Specifications of Delegation Features

This section defines a delegation policy by taking into account its various features. We will further use these definitions to express delegation mutant operators in Section V.

A. Delegation Policy

A delegation policy can be considered as an administrative-related security policy that is built on top of an access control policy. It is composed of delegation rules that can be specified at two levels: master-level and user-level. Basically, a delegation policy is two-fold:

1. It specifies who has the right to delegate which permission (for accessing to a given resource/action/subject) to whom, and in which context. Let us call this kind of rule is master-level delegation rule as such a rule is normally defined by policy officers. For example, a policy officer can define a rule to specify that the head of department at a university can delegate the permission of updating personnel accounts to a professor only.

Definition 2 (Master-Level Delegation): Let U be a set of users, P be a set of permissions, and C be a set of contexts. A master-level delegation policy MLD is defined as a user-user-permission-context assignment relation: $MLD \subseteq U \times U \times P \times C$. A delegation of a permission p from a user u_1 to a user u_2 in a given context c **can be performed** if and only if $(u_1, u_2, p, c) \in MLD$.

2. It specifies who is delegating to whom which permission, and in which context. Let us call this kind of rule is user-level delegation rule as these rules are mostly defined by normal users. Note that user-level delegation rules must conform to master-level delegation rules. For example, *Bill* (the head of department) is delegating his permission of updating personnel accounts to *Bob* (a professor) during his absence.

Definition 3 (User-Level Delegation): Let U be a set of users, P be a set of permissions, and C be a set of contexts. A user-level delegation policy ULD is defined as a user-user-permission-context assignment relation: $ULD \subseteq U \times U \times P \times C$. A user u_2 can have a permission p **by delegation** from a user u_1 in a given context c if and only if $(u_1, u_2, p, c) \in ULD$.

B. Context

A context is a condition or a combination of conditions in which an access control/delegation rule is active, i.e. enforced in the running system. Cuppens et al. [20] discuss five different kinds of contexts. These kinds include temporal, spatial, user-declared, pre-requisite and provisional context. Temporal delegation is delegation within a time constraint, for example delegation is active for two days or delegation is active for the time user is on vacation. The spatial context is used for subject location, e.g. a delegated permission is only active when the delegatee is at office. User-declared context is related to the purpose of the subject. Prerequisite context allows delegation when some pre-condition is satisfied and the provisional context depends on the previous actions that subject has performed on the system.

Our security model supports context composition using following operators: conjunction $\&$, disjunction $\oplus$, and negation $\bar{}$.

Note that every delegation rule defined in this paper is always associated with a context c. By default, if not specified explicitly, a context c is at least composed of a condition, called *Default*, i.e. always true.

C. Delegation Features

Delegation is a powerful and very useful way to perform access control policy administration. On one hand, it allows users to temporarily modify the access control policy by delegating access rights. By delegation, a delegatee can perform the delegated job, without requiring the intervention of the security officer. On the other hand, the grantor/delegator and/or some specific authorized users should be supported to revoke the delegation either manually or automatically. On both hands, the administrative task can be simplified and collaborative work can be managed securely, especially with the increase in shared information and distributed systems. However, the simpler the administrative task can be, the more complex features of delegation have to be properly specified and enforced in the software system.

In this section, we define the most well-known complex delegation features and formally specify them w.r.t the definitions of access control and delegation policies. In the following definitions, we use pre, $body$, and $post$ to respectively specify the state of the policy before changing, the state while it is being changed by the function, and the state after changing.

1) Monotonicity of Delegation: Monotonicity of delegation refers to whether or not the delegator can still use the permission while delegating it [21]. If the delegator can still use the permission while delegating it, the delegation is called grant delegation. Of course, the delegatee can use the permission while it is being delegated to him. This is monotonic because available authorizations (in AC) are increased due to successful delegation operations. Again, note that every delegation can only be performed if and only if it satisfies the master-level delegation policy.

Definition 4 (Grant Delegation):
$grantDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD$
body $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

Vice versa, if the delegator can not use the permission while delegating it, the delegation is called transfer dele-

gation. As such, this is non-monotonic because available authorizations (in AC) are not increased due to successful delegation operations.

Definition 5 (Transfer Delegation):
$transferDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD$
body $AC := AC \setminus \{(u_1, p, c)\}$; $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \notin AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

2) Temporary Delegation: This is also a very common feature of delegation used by users. When revocation is handled automatically, the delegation is called temporary. In this case, the delegator specifies the temporal conditions in which this delegation applies: only at a given time, after or before a given time, or during a given time interval. The temporal conditions may correspond to a day of the week, or to a time of the day, etc. If the temporal context is not used, the delegation needs to be revoked manually.

As we have mentioned, a delegator (e.g. *Bill*) can include a temporal condition in the context of the delegation rule, for instance during his vacation. Thus, here the temporal context is *vacation_period* defined as follows:

Definition 6 (Temporary Delegation): Let c be a given context of a delegation (either grant delegation or transfer delegation). A delegation is specified as temporary if its context is associated with some time constraint. The delegation will be only active while the time constraint is satisfied.

$c := c\&vacation\_period(startDate, endDate)$
where $vacation\_period(startDate, endDate) : -$
$startDate \leq endDate \wedge afterDate(startDate) \wedge beforeDate(endDate)$

Here, $afterDate(date)$ returns $true$ iff $date$ is equal or later than the current date. Similarly, $beforeDate(date)$ returns $true$ iff $date$ is equal or earlier than the current date.

3) Multiple Delegation: Multiple delegation refers to the maximum number of times a permission can be delegated at a given time. We define a counting function to count the number of delegations of a permission, delegated by a delegator in a given context. The number returned by this function is always updated according to the change in the delegation policy, i.e. the number of delegation rules related to permission p.

$countDelegation(u, p, c) := |\{(u, v, p, c) \mid \forall v \in U : (u, v, p, c) \in ULD\}|$

Let N_m be this number and it is pre-defined by the security officer.

Definition 7: $grantDelegation(u_1, u_2, p,$
$c\&countDelegation(u_1, p, c) < N_m) : -$ pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD \wedge countDelegation(u_1, p, c) < N_m$
body $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

4) Multi-step Delegation: This characteristic refers to the maximum number of steps (N_s, normally specified by a security officer) that a permission p can be re-delegated, counted from the first delegator of this permission. So if $N_s = 0$ that means the permission p can not be re-delegated anymore. First, let us define a helper function that returns how the number of times a permission is re-delegated in a given context.

$stepCounter(u_0, p, c) := N_s$ with u_0 is the first delegator of p in the delegation chain: u_0 delegates p to ... in a given context c; ... re-delegates p to u_1 in context c; and u_1 re-delegates p to u_2 in context c.

If there exists a pre-defined maximum number of steps N_s for a permission p as described above, the delegation is specified as following.

Definition 8: $grantDelegation(u_1, u_2, p, c$
$\&stepCounter(u_1, p, c) \geq 1) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD \wedge stepCounter(u_1, p, c) \geq 1$
body $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$; $stepCounter(u_2, p, c) := stepCounter(u_1, p, c) - 1$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

5) Delegation Revocation: Delegation support revocation feature in which a delegation can be revoked and permissions are returned back to the original user. The revocation of a grant delegation means to deny access of the delegatee to the delegated permission.

Definition 9: $revokeGrantDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$
body $AC := AC \setminus \{(u_2, p, c)\}$; $ULD := ULD \setminus \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \notin ULD$

The permission to be revoked is deleted from the access rights of delegatee.

To revoke a transfer delegation, it is not only to deny access of the delegatee to the delegated permission but also to re-grant access to the delegator who is temporarily not having this access.

Definition 10: $revokeTransferDelegation(u_1, u_2, p, c) : -$
pre $(u_2, p, c) \in AC \wedge (u_1, p, c) \notin AC \wedge (u_1, u_2, p, c) \in ULD$
body $AC := AC \setminus \{(u_2, p, c)\}$; $AC := AC \cup \{(u_1, p, c)\}$; $ULD := ULD \setminus \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \notin ULD$

V. Mutant Operators

In this section, we define mutant operators for delegations. The operators are defined solely for delegation in an access

control policy and we do not consider other aspects of policy such as obligations. Testing of access rights and obligations is done separately in several works [18], [17]. We categorize the operators into two broad categories: basic delegation operator and advanced delegation operator.

Note that we omitted the notion of role in Section III because we focused on the formalisation of different features of delegation where they can be specified without the definition of role. However, in practice, the notion of role is commonly used for supporting natural abstractions like sets of permissions. Role-Based Access Control (RBAC) introduces a set of role and decomposes the relation AC into user-role assignment $UR \subseteq U \times R$, and role-permission-context assignment $RPC \subseteq R \times P \times C$. Thus, $AC = UR \circ RPC$. For evaluation of operators, we will derive our delegation operators based on RBAC but we consider aspects that are commonly found in most access control and delegation models.

A. Basic Delegation Mutant Operators

Basically delegation are of two types: permission delegation and role delegation.
- Permission delegation: Users can delegate specific permission(s) that are associated with their own roles.
- Role delegation: Role delegation means a user empowered in some role(s) can delegate his role to other user (s). In this way the delegatee can use permissions of his role and permissions of role that is delegated to him. We introduce the following operators based on the basic types of delegation.

1) Permission Delegation Operator: Permission delegations are the most frequent type of delegations used in access control. Permissions are delegated between two subjects, like in LMS, one such delegation is Alice (secretary) delegated her permission to create borrow account to Jane (librarian). The Permission Delegation Operator (PDM) will mutate the permission delegation rule by replacing the permission being delegated by another permission of the delegator.

Definition 11: $PDM(u_1, u_2, p_{1a}, c) : -$
pre $(u_1, u_2, p_{1a}, c) \in ULD \wedge (u_1, r_1) \in UR \wedge (u_2, r_2) \in UR \wedge (r_1, p_{1a}, c) \in RPC \wedge (r_1, p_{1b}, c) \in RPC$
body $ULD := ULD \setminus \{(u_1, u_2, p_{1a}, c)\} \cup \{(u_1, u_2, p_{1b}, c)\}$; $AC := AC \cup \{(u_2, p_{1b}, c)\}$ end
post $(u_2, p_{1b}, c) \in AC \wedge (u_1, u_2, p_{1b}, c) \in ULD$

2) Role Delegation Operator: The Role Delegation Operator (RDM) is used to simulate errors in delegation of roles. The RDM operator will mutate the delegation rule by replacing the delegator by some other user having different role. For example in the LMS system, one delegation is: Bill (Director) delegates his role to Bob (Secretary). The most important aspect in such a delegation is to establish correct link between delegator-role-delegatee. This means that the right delegator delegates the right role to the delegatee.

Definition 12: $RDM(u_1, u_2, r_1, c) : -$
pre $(u_1, r_1) \in UR \wedge (u_2, r_2) \in UR \wedge (u_3, r_3) \in UR \wedge r_1 \neq r_2 \neq r_3 \wedge (u_1, u_2, r_1, c) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, r_1, c)\} \cup \{(u_3, u_2, r_3, c)\}$; $UR := UR \cup \{(u_2, r_3)\}$ end
post $(u_2, r_3) \in UR \wedge (u_3, u_2, r_3, c) \in ULD$

B. Advanced Delegation Operators

Advanced delegation operators are described w.r.t advanced properties of delegation such as transfer delegation, temporary delegation, user-specific delegation, role-specific delegation, etc., we introduce mutants to simulate errors that affects the correctness of these types of delegation.

1) Monotonic Delegation Operators: A delegation can be monotonic or non-monotonic. The former specifies that the delegated access right is available to both the delegator and delegatee after enforcing this delegation. The latter means that the delegated role/permission is transferred to the delegatee, and the delegator temporarily loses his rights while delegating them. In this case, the delegation is called a transfer delegation. For instance in LMS, a secretary (Bob) transfers his role to an administrator (Sam) and Bob no longer can use his role during the delegation period.

By mutating the property $monotonicity$ of a delegation, we can simulate errors such as the enforcement of a transfer delegation is implemented as a grant delegation, or vice versa, a grant delegation is changed to a transfer delegation. The Transfer to Grant Delegation Operator (T2G) and the Grant to Transfer Delegation Operator (G2T) are defined for performing these mutations.

Definition 13: $G2T(u_1, u_2, p, c\&IsMonotonic) : -$
pre $(u_1, p, c) \in AC \wedge (u_1, u_2, p, c\&IsMonotonic) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, p, c\&IsMonotonic)\} \cup \{(u_1, u_2, p, c\&IsNonMonotonic)\}$ end
post $(u_1, p, c) \notin AC \wedge (u_1, u_2, p, c\&IsNonMonotonic) \in ULD$

Definition 14: $T2G(u_1, u_2, p, c\&IsNonMonotonic) : -$
pre $(u_1, p, c) \notin AC \wedge (u_1, u_2, p, c\&IsNonMonotonic) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, p, c\&IsMonotonic)\} \cup \{(u_1, u_2, p, c\&IsMonotonic)\}$ end
post $(u_1, p, c) \in AC \wedge (u_1, u_2, p, c\&IsMonotonic) \in ULD$

2) Context-based Delegation Operators: Delegations are always applied in some context. Test should guarantee correct implementation of context of delegations. Delegation contexts can be temporal, spatial, provisional, pre-requisite or a user declared context. The context is tested by reducing the scope of the context (CR), by extending the scope of the context (CE) and by negating the original context (CN).

In this paper, we only deal with temporal delegation because of its popularity [20]. The other types of context-based delegations are not considered. We introduce the Temporal Delegation Operator (TDM) to mutate the duration of temporal delegation. To be more specific, we mutate the temporal delegation rule by reducing or expanding the

duration of a temporal delegation. For example, we apply TDM for the temporal delegation defined in Definition 6 as follows.

Definition 15: $TDM(startDate, endDate) : -$
pre $vacation\_period(startDate, endDate)$
body $startDate := laterStartDate \wedge laterStartDate > startDate \wedge laterStartDate < endDate$ end
post $vacation\_period(laterStartDate, endDate)$

3) Role-Specific Delegation Operators: A security policy could allow users having a specific role can only delegate to other users having some role that belongs to the possible delegation targets of that role. This can be seen as the case where the security policy ensures that no conflict of interest can occur by delegation. Roughly speaking, one example is in any case a student must never have permissions of both roles $student$ and $lecturer$ because he can edit his own grades. That means a lecturer does not have right to delegate his role to his student. To test this kind of restriction we propose the following operators. The Role Delegation Off-Target 1 Operator ($RDOT_1$) will replace the delegatee by another user whose role is not a possible target of the delegator's role. That means there exists a delegation rule at master level saying that a user of this role is a possible delegatee of a user of another role: $(u_1, u_2, r, c) \in MLD$. Note that to keep it general, we use the definition of master-level delegation rule to refer to all kinds of delegation rule specifying constraints, e.g. role-specific delegation, multiple delegation, multi-step delegation, etc.

Definition 16: $RDOT_1(u_1, u_2, r, c) : -$
pre $(u_1, u_2, r, c) \in MLD \wedge (u_1, u_3, r, c) \notin MLD \wedge (u_1, u_2, r, c) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, r, c\} \cup \{(u_1, u_3, r, c)\}$ end
post $(u_1, u_3, r, c) \notin MLD \wedge (u_1, u_3, r, c) \in ULD$

Vice versa, the Role Delegation Off-Target 2 Operator ($RDOT_2$) will replace the delegator by another user whose role's delegation targets set does not contain the delegatee's role.

Definition 17: $RDOT_2(u_1, u_2, r, c) : -$
pre $(u_1, u_2, r, c) \in MLD \wedge (u_3, u_2, r, c) \notin MLD \wedge (u_1, u_2, r, c) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, r, c\} \cup \{(u_3, u_2, r, c)\}$ end
post $(u_3, u_2, r, c) \notin MLD \wedge (u_3, u_2, r, c) \in ULD$

4) Permission-Specific Delegation Operators: In this category, we discuss permission specific operator. The Non-Delegable Permission Delegation Operators (NDPD) will mutate a permission delegation by changing the delegated permission from delegable to non-delegable.

Definition 18: $NDPD(u_1, u_2, p, c) : -$
pre $(u_1, u_2, p, c) \in MLD \wedge (u_1, u_2, p, c) \in ULD$
body $MLD := MLD \setminus \{(u_1, u_2, p, c\}$ end
post $(u_1, u_2, p, c) \notin MLD \wedge (u_1, u_2, p, c) \in ULD$

5) Multiple Delegation Operator: Tests should reveal problems in ensuring that the number of concurrent delegations of a specific role/permission does not exceed the threshold defined for it at the master level. To simulate the case where adding a new delegation will make that number exceeds a pre-define threshold of a role/permission, we introduce Multiple Delegation Operator (MultiD).

Definition 19: $MultiD(countDelegation(u_1, p, c) < N_m) : -$
pre $(u_1, u_2, p, c) \in MLD \wedge countDelegation(u_1, p, c) = N_m$
body $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ end
post $(u_1, u_2, p, c) \in ULD \wedge countDelegation(u_1, p, c) = N_m + 1$

6) Multi-step Delegation Operator: Tests should also reveal problems in re-delegation of permissions and roles. Some roles and permissions can be defined as re-delegable only after a limited number of times. We mutate the policy by re-delegating a role or permission that is not re-delegable and vice versa. In the mutated policy the delegatee will take the role/permission of delegator. Similarly the role and permissions that should not be redelegated are mutated by re-delegating them. The Re-delegation Operator (ReD) add a new delegation rule into the policy where the delegating permission/role must not be re-delegated any more ($stepCounter = 0$).

Definition 20: $ReD(stepCounter(u_1, p, c)) : -$
pre $(u_1, u_2, p, c) \in MLD \wedge stepCounter(u_1, p, c) = 0$
body $ULD := ULD \cup \{(u_1, u_2, p, c)\}$; $stepCounter(u_2, p, c) := stepCounter(u_1, p, c) - 1$ end
post $(u_1, u_2, p, c) \in ULD \wedge stepCounter(u_2, p, c) = -1$

7) Delegation Removal Operators: Tests should be able to detect that a delegation rule is missing. We introduce the Delegation Removal Operator (DR) that removes one of the delegation rules.

Definition 21: $DR(u_1, u_2, p, c) : -$
pre $(u_1, u_2, p, c) \in ULD$
body $ULD := ULD \setminus \{(u_1, u_2, p, c\}$ end
post $(u_1, u_2, p, c) \notin ULD$

All the delegation mutants defined in this section are derived from basic delegation types as well as various (advanced) well-known delegation features. Thus, we believe they are representative for testing delegation policies. A demonstration example of using some of these mutants is given in the next section.

VI. Demonstration of Testing Delegation Policy Enforcement

In this section, we first give a brief description of a prototype system (LMS) implemented as a proof-of-concept. Then, we demonstrate how the proposed mutation analysis approach has been applied on this system.

A. Model-Driven Adaptive Delegation

The implementation of LMS is part of our previous work [19] on Model Driven Adaptive Delegation. In [19], we

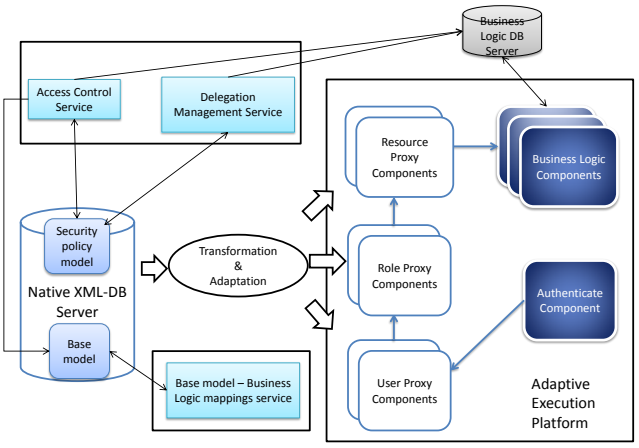

Figure 2. Architectural framework

propose a model-driven framework in which on one hand, we can specify a security model using a Domain Specific Language (DSL). On the other hand, the business logic of the system can be specified using another DSL, i.e. component-based architecture. As shown in Figure 2, the security model will be transformed and composed with the architecture model of the system. The composed model actually reflects the security policy at architecture level. By leveraging the notion of model@runtime [22], the running system can be adapted according to the newly composed model. Thus any change in the policy (access control and/or delegation) will be automatically enforced in the running system at runtime. Figure 3 shows an example of the 3-layer architecture reflecting security policy. In this policy configuration, $Bill$ has role $Director$ that can have access rights to $BorrowerAccountResource$ but currently $Bill$ is temporarily transferring his role to Bob (his secretary). The links between components at different layers are actually the service bindings from "client" ports (of the components on the left) to "server" ports (of the components on the right).

B. Mutation Process

Figure 4 gives an overview of the mutation process for testing delegation. Based on the mutant operators presented in Section V our delegation policy can be easily mutated. Via the model-driven framework described above, those mutated policies can be automatically enforced in the running system.

We performed mutation analysis on the examples described in Section II. We used those three delegation situations to demonstrate the application of the proposed approach. We chose five test cases to test these three types of delegation of our running example. Table I gives an overview of them.

In case of delegation 1, Bill (Director) delegates his permission consultPersonnelAccount to Bob (Secretary). This simple delegation is mutated in two different ways, first

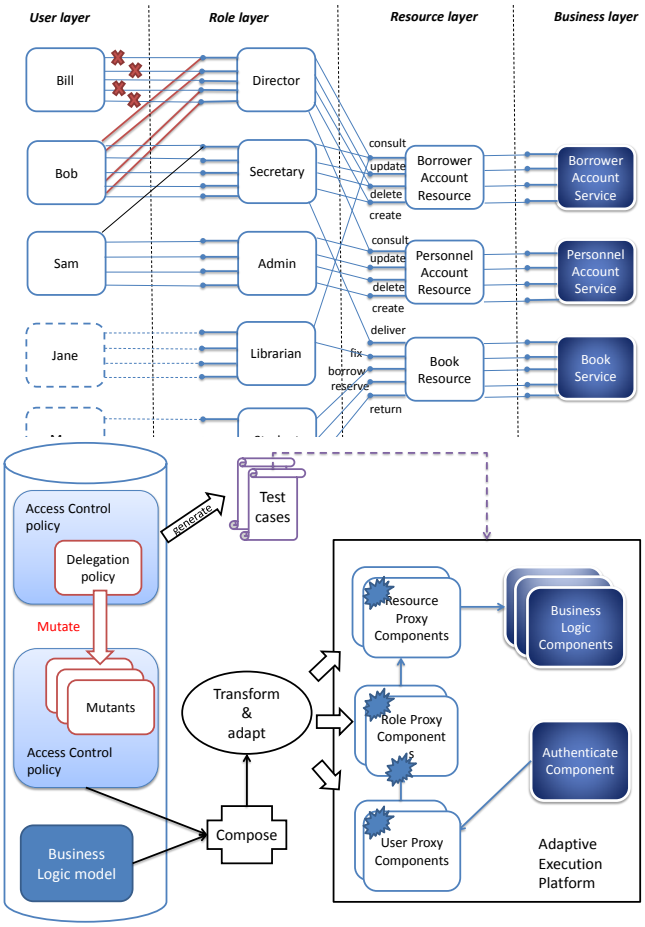

Figure 4. Mutation Process for testing Delegation Policy enforcement

replacing Bill by Sam who is an administrator (by using RDM) and then by replacing consultPersonnelAccount (PDM) with another permission, e.g. consultBorrowerAccount. We created one test case (TC1) to test if Bob can do consultPersonnelAccount after enforcing the delegation rule. The test case was able to kill the second mutant because Bob was not delegated consultPersonnelAccount but consultBorrowerAccount. However, it did not kill the first mutant because Sam also has the right consultPersonnelAccount as Bob. Thus, Sam's rights including consultPersonnelAccount have been delegated to Bob, that made him accessible to consultPersonnelAccount.

In case of delegation 2, we use two test cases to test this delegation. One (TC2) is used to test the correct enforcement of the delegated permission (createBorrowerAccount). It resembles the test case for delegation 1, in which we test right allocation of permission. Similarly, it detects mutant in where a wrong permission is inserted (PDM) but fails to kill mutant when the delegator is replaced (RDM). The other test case (TC3) is used to test transfer (monotonicity) property of the delegation, i.e. Alice cannot use createBorrowerAccount while transferring it to Jane. For this case, we used T2G

to change the type of delegation 2 from transfer to grant. TC3 can kill the mutant of T2G but not the other mutants of PDM and RDM because in those mutants Alice cannot use createBorrowerAccount.

We test delegation 3 in a temporal context. We chose two test cases (TC4 and TC5) to kill the four mutants of this delegation. Two mutants are created by injecting permission and delegator related faults (PDM and RDM). The other two mutants are created by reducing and expanding the duration of a temporal delegation (TDM). We created a test case (TC4) similar to TC1. The other test case (TC5) is designed to detect if enforcement of delegation 3 is correct during its active duration. TC5 kills TDM but leaves undetected the mutants of PDM and RDM. Vice versa, TC4 cannot kill the mutant of TDM.

VII. Related Work

Testing of access control policy is a relatively new domain. Some work is available on testing of access control policy with XACML. Martin and Xie [17] propose a fault model for access control policies based on XACML. This fault model was then used to measure the effectiveness of their test cases. The fault model considers access control features composed of permissions and prohibitions. Hu et al. [15] elaborate the conformance checking of access control policy. Other approaches try to test a policy by using state machines. The main focus of such approaches is the generation of test cases for access control policies [16], [7]. To the best of our knowledge, there has not been any approach dedicated to testing delegation enforcement taking into account an extensive delegation model like ours.

Le Traon et al. [23] establish a mutation analysis approach for performing security testing. Xu et al. [8] propose a model based testing approach for access control policies. They use both the policy and its contracts implemented in a number of industry usable languages. The test adequacy is also measured through mutation analysis. El Rakaiby et al. [18] use mutation analysis to test obligations. Their approach (not model-based) is on the same lines with the one proposed here except that their focus is obligation and our focus is delegation. However, there is a big difference because delegation and obligation are two different aspects. Especially, because of the "meta-level" character of delegation w.r.t access control, applying mutation analysis for testing delegation enforcement has to take into account different delegation features in the whole process. Moreover, our mutation analysis approach bases on our model-driven framework showed in [19] making it easy to leverage model-based testing techniques.

VIII. Conclusion

The process of delegating access rights forms one of the central issues in the administration of an access control policy. The delegation of authority gives users more rights over protected resources so its undisputed implementation and enforcement require a rigorous testing process. In view of this, the present paper suggests the use of mutation analysis to test the delegation policy enforcement. To achieve this, a careful and formal analysis of different delegation features was performed. Based on this analysis a set of mutant operators specific for delegation has been derived.

In future, a thorough empirical study using the proposed mutant operators is planed. This will lead in identifying the most useful and effective mutant operators. Moreover, we aim at investigating ways to automatically generate test cases for killing the proposed mutants. Towards this direction we aim at extending existing approaches [24], [25] to deal with delegations. Finally, the integration of model-based testing with the suggested mutation approach will be also researched.

Acknowledgment

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under the MITER project C10/IS/783852.

References

[1] M. Ben-Ghorbel-Talbi, F. Cuppens, N. Cuppens-Boulahia, and A. Bouhoula, "A delegation model for extended RBAC," *International Journal of Information Security*, vol. 9, pp. 209–236, June 2010.

[2] T. Ahmed and A. R. Tripathi, "Static verification of security requirements in role based CSCW systems," *SACMAT 03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pp. 196–203, 2003.

[3] F. Hansen and V. Oleshchuk, "Conformance checking of rbac policy and its implementation," *Proceedings of Information Security Practice and Experience: First International Conference, ISPEC*, vol. 3439 / 2005, 2005.

[4] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," *In Proceedings of the 14th Usenix Security Symposium*, August,2005.

[5] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing security policies: going beyond functional testing," *in Proceedings of 18th IEEE international symposium on Software Reliability. ser ISSRE*, 2007.

[6] E. Martin and T. Xie, "Automated Test Generation for Access Control Policies via Change-Impact Analysis," *Third International Workshop on Software Engineering for Secure Systems , SESS'07: ICSE Workshops*, 2007.

[7] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur, "Scalable and Effective Test Generation for Role-Based Access Control Systems," *IEEE Transactions on Software Engineering*, vol. 35,5, pp. 654–668, 2009.

[8] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon, "A Model-Based Approach to Automated Testing of Access Control Policies," *SACMAT*, 2012.

Table I
SOME PRELIMINARY MUTATION ANALYSIS RESULTS

Test Case	Killed Mutants	Live Mutants
Test Case-1	PDM (wrong permission mutant)	RDM (delegator fault)
Test Case-2	PDM (wrong permission)	RDM (delegator replaced)
Test Case-3	T2G (wrong type of delegation)	PDM, RDM (wrong delegator, permission)
Test Case-4	PDM (permission replaced)	TDM (CE,CR)
Test Case-5	TDM (CE,CR)	PDM, RDM

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34–41, Apr. 1978.

[10] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 279–290, July 1977.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[12] M. Papadakis and Y. L. Traon, "Using mutants to locate "unknown" faults," in *ICST*, pp. 691–700, 2012.

[13] J. Offutt, "A mutation carol: Past, present and future," *Information & Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[14] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[15] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance Checking of Access Control Policies Specified in XACML," *Computer Software and Applications Conference, COMPSAC 2007, 31st Annual International*, vol. 2, pp. 275 – 280, 2007.

[16] K. Li, L. Mounier, and R. Groz, "Test Generation from Security Policies Specified in Or-BAC," *Computer Software and Applications Conference, 2007. COMPSAC 2007, 31st Annual International*, vol. 2, pp. 255– 260, 2007.

[17] E. Martin and T. Xie, "A Fault Model and Mutation Testing of Access Control Policies," *International World Wide Web Conference Committee*, 2007.

[18] Y. Elrakaiby, T. Mouelhi, and Y. Le Traon, "Testing Obligation Policy Enforcement Using Mutation Analysis," *ICST '12 Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 673–680, 2012.

[19] P. H. Nguyen, G. Nain, J. Klein, T. Mouelhi, and Y. Le Traon, "Model-Driven Adaptive Delegation," in *Proceedings of the Aspect-Oriented Software Development conference MODULARITY: aosd13*, ACM, 2013.

[20] F. Cuppens and N. Cuppens-Boulahia, "Modeling contextual security policies," *International Journal of Information Security*, vol. 7, pp. 285–305, Nov. 2007.

[21] J. Crampton and H. Khambhammettu, "Delegation in role-based access control," *International Journal of Information Security*, vol. 7, pp. 123–136, Sept. 2007.

[22] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, pp. 44–51, Oct. 2009.

[23] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, "Test-Driven Assessment of Access Control in Legacy Applications," *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 238 – 247, 2008.

[24] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information & Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[25] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, pp. 691–723, 2011.