Transforming and Selecting Functional Test Cases for Security Policy Testing

Tejeddine Mouelhi, Yves Le Traon
Institut TELECOM ; TELECOM Bretagne ; RSM, 2 rue de la Châtaigneraie CS 17607, 35576 Cesson Sévigné Cedex
Université européenne de Bretagne, France
{tejeddine.mouelhi,yves.letraon}@telecom-bretagne.eu

Benoit Baudry
IRISA/INRIA
35042 Rennes
France
bbaudry@irisa.fr

Abstract

In this paper, we consider typical applications in which the business logic is separated from the access control logic, implemented in an independent component, called the Policy Decision Point (PDP). The execution of functions in the business logic should thus include calls to the PDP, which grants or denies the access to the protected resources/functionalities of the system, depending on the way the PDP has been configured.

The task of testing the correctness of the implementation of the security policy is tedious and costly. In this paper, we propose a new approach to reuse and automatically transform existing functional test cases for specifically testing the security mechanisms. The method includes a three-step technique based on mutation applied to security policies (RBAC, XACML, OrBAC) and AOP for transforming automatically functional test cases into security policy test cases. The method is applied to Java programs and provides tools for performing the steps from the dynamic analyses of impacted test cases to their transformation. Three empirical case studies provide fruitful results and a first proof of concepts for this approach, e.g. by comparing its efficiency to an error-prone manual adaptation task.

1. Introduction

Automatically transforming functional test cases in order to test another concern seems an open issue, while it would drastically reduce the testing effort: no more effort would be spent other than the one already paid when testing system functions. In this paper, we explore this issue in the case of security policy testing (the new concern/property to be tested). The intuition which makes the proposed approach feasible is based on two facts:

- All security mechanisms are necessarily exercised at least once by functional test cases if they cover 100% of the executable statements: the functional test sequences can thus be reused. Only the *oracle function* has to be transformed to test that the tested security mechanism behaves as expected,
- The typical access control logic is separated from the application code (business logic): this access control logic can thus be manipulated or modified.

Indeed, in a typical application which embeds access control mechanisms, the business logic implementation is separated from the access control logic [1]. In such systems, the recommended architecture consists of designing a security component, called the Policy Decision Point (PDP), which can be configured independently from the rest of the application. The PDP is configured with respect to a security policy, modeled using an access control modeling language such as OrBAC or RBAC [2, 3] A security policy is composed of a set of access control rules, which specify the conditions for granting or denying access to protected elements. The execution of functions in the business logic should thus include calls to the PDP, which grants or denies the access to the protected resources/functionalities of the system.

From a security testing point of view, the issue is to determine the correctness of the interactions between the business logic and the PDP. The generation of test cases targeting security thus consists in generating a test sequence ending with a given access control call, and also relies on the creation of the oracle function. Observing that the execution of a given test sequence effectively leads to an access granted does not mean that a security mechanism has correctly checked the access rights for this specific call. To be fully efficient, the oracle should be built by observing the piece of code which corresponds to the call to the PDP.

The problem we are tackling in this paper is related to the definition of an efficient oracle function for

security test cases. In particular, we propose an approach to adapt existing functional test cases that already define relevant test scenarios, by updating their oracle to take security concerns into account.

In [4], we studied the overlap between functional test cases (generated for testing the functions of the business logic) and the test cases needed to test all the security mechanisms. To adapt functional test cases, in order to take security into account, we need to identify test cases that trigger security mechanisms. Then, their oracle should be enhanced to also check that the triggered security mechanisms work properly. This paper provides an automated solution for selecting functional test cases which are qualified to test security mechanisms. A dynamic analysis based on mutation of access control rules helps pinpointing which access control rules are already exercised by each test case. These rules need to be tested specifically by the security tests as outlined our previous work [4]

Detecting the test cases impacted by a security rule (triggering at least one security mechanism) is also useful to perform regression testing on the parts of the policy that have not changed. It allows testing that the evolution has not introduced unexpected changes.

In this paper, we propose a method and tools (1) to detect the test cases which are impacted by a security policy and (2) transform existing functional test cases in security policy test cases.

To reach this objective, several problems are treated incrementally:

1. Select tests that are impacted by the security policy: this selection step aims at reducing the step 2 execution times, we perform a conservative impact analysis
2. for each selected test case, determine exactly which rules impact the test case,
3. for each test case, modify and adapt the oracle to observe the security mechanism.

At the end of this three-step process, the existing functional test cases that trigger security mechanisms are transformed into security policy test cases. The presented approach has been applied to Java programs.

To validate the approach, we apply it to the case study presented in [4] and augmented with two other case studies. We study the feasibility of the approach and show the obtained results. In addition, we study the efficiency of our approach compared to a manual one. Also, we present to what extent the first selection step improves the performances of the approach regarding the execution times of the second step.

Section 2 introduces the context of this work and defines important concepts that are used in this study. Section presents the overall process for selecting functional test cases that trigger security mechanisms and adapt their oracle functions to take security concerns into account. Section 4 and 5 detail the two steps analyses performed to identify the relationships between functional test cases and security rules. Section 6 details how these test cases are adapted to consider security and section 7 discusses the results obtained from the three case studies.

2. Definitions and Context

Before detailing the proposed approach, we need to provide a few fundamental concepts. In this section, we present the context and definitions of the important concepts.

2.1. Context

When building a software system that has specific requirements, the business logic and the security policy are usually modeled, analyzed and developed separately. The security policy is specified as a set of rules, according to one of the various access control languages that are available (RBAC and variants such as XACML, OrBAC…). This abstract specification of the security policy allows reasoning about the rules and performing specific types of analyses on these rules. Eventually, the global system has to take into account both the business and the security concerns into a single implementation. The security policy can be implemented as a separate component called the Policy Decision Point (PDP) that interacts with the business logic through Policy Enforcement Points (PEP). Although the PDP can be automatically generated from the security policy, the location of PEPs in the business logic has to be decided in an ad-hoc way to take into account specific design and implementation decisions.

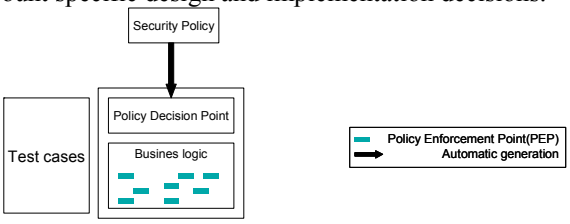

Figure 1 – Context for testing a secured system

Our goal is then to validate that PEPs are correct in the sense that they implement the desired interactions between the business logic and the PDP in order to enforce the security policy. In this work, we validate the integration of the security policy with the business logic through testing. More precisely we want to adapt existing functional test cases so that they can take security into account when they check the correctness of the system. Our concern in this paper is then to identify the test cases which cover PEPs, and thus trigger security mechanisms

In [5], we propose a model-based approach for specifying, deploying and testing an access control policy. The specification of the policy is performed according to a generic meta-model, which allows this policy to be expressed in any access control language (RBAC, OrBAC, DAC, MAC, XACML). The meta-model includes generic mutations operators that define the semantic of mutation operators at the generic model. This framework allows us to produce mutants for any access control language that conforms to the generic model, for instance XACML. The generation of the PDP and the deployment is performed using several model-based tools (AOP, model transformation). The missing component for this fully automated framework is the generation of test cases that exercise the application code to guarantee that the access control policy is fully respected and implemented in the application internal code (no backdoors or hidden mechanisms [6]).

2.2. Definitions

We need to define or recall some definitions:

Business Logic – The business logic is the part of the system which implements the system functionalities.

A security policy defines a set of security rules that specify rights and restrictions of actors on parts and resources of the system. A rule can be a permission or a prohibition. Each rule consists of five parameters (called entities): a *status* flag S indicating permission or prohibition, a *role*, an *activity*, a *view*, and a *context*. Our domain consists of role names RN, activity names PN, view names VN, and context names CN.

Security Policy (SP): A *security policy SP* is thus set of rules defined by $SP \subseteq S \times RN \times PN \times VN \times CN$. The signature of an access rule is thus:

Status(Role, Activity, View, Context)

Examples:

Permission(Borrower,Borrow,Book,WordingDay)

Prohibition(Borrower,Consult,PersonnelAccount, Default)

Security mechanism – It denotes any piece of code or constraint internal to the business logic which restricts (or relaxes) the access to some protected resources/functions of the system. A specific case of security mechanism is called PEP, which explicitly calls the external PDP component (see the following definitions).

PEP - The Policy Enforcement Point is the point in the business logic where the policy decisions are enforced. It is a security mechanism, which has been intentionally inserted in the business logic code. On call of a service that is regulated by the security policy, the PEP sends a request to the PDP to get the suitable response for the requested service by the user and in the current context. Based on the response of the PDP, if the access is granted the service executes, and if the access is denied the PEP forbids the execution of the service.

PDP - The Policy Decision Point is the point where policy decisions are made. It encapsulates the Access Control Policy and implements a mechanism to process requests coming from the PEP and return a response which can be deny or permit.

System/functional testing – the activity which consists of generating and executing test cases which are produced based on the uses cases and the business models (e.g. analysis class diagram and dynamic views) of the system. By opposition with security tests, we call these tests functional.

Security policy testing (SP testing): it denotes the activity of generating and executing test cases that are derived specifically from a SP. The objective of SP testing is to reveal as many security flaws as possible.

Test case: In the paper, we define a test case as a triplet: intent, input test sequence, oracle function.

Intent of a test case: The intent of a test case is the reason why an input test sequence and an oracle function are associated to test a specific aspect of a system. It includes at least the following information: (functional, names of the tested functions) for functional test cases or (SP, names of the tested security rules) for SP ones.

Test cases aim at detecting security flaws in the systems under test. In this paper, the faults (or "flaws") can be classified into two main categories.

Security faults - interaction faults and *hidden security mechanisms. Security interaction faults* occur when the interactions between the business logic implementation and the PDP is erroneous. Interaction faults thus correspond to a fault in the security mechanism itself (and maybe its absence at a given point in the software). It may be caused by manual modifications in the business logic code (e.g. adding manually an unexpected call to the PDP, erroneous parameters) or to error in the use of AOP if this technique is used to insert the PEP code. Such faults may generate unexpected interactions between the business logic and the PDP. *Hidden security mechanisms* correspond to design constraints or pieces of code which may bypass the PEP which is expected to control the access for a specific execution.

SP oracle function: The oracle function for a SP test case is a specific assertion which interrogates the

security mechanism. There are two different oracle functions:

- For permission, the oracle function checks that the service is activated (access granted).
- For a prohibition, the oracle checks that the service is not activated (access denied).

The *intent* of the functional tests is not to observe that a security mechanism is executed correctly. For instance, for an actor of the system who is allowed to access a given service, the functional test intent consists of making this actor execute this service. Indirectly, the permission check mechanism has been executed, but a specific oracle function must be added to transform this functional test into a SP test.

Adaptation of a functional test case – The adaptation of a functional test case consists of (1) modifying its intent, (2) identifying the security rules the test sequence triggers and (3) adding the SP oracle function.

Point 2 is obtained by determining which test cases are impacted by the security policy, and more precisely by a given security rule.

Impacted Test case – A test case is said to be *impacted by the security policy*, if its execution triggers at least one of the PEPs in the business code. A test case is said to be *impacted by a security rule R* of the security policy if it triggers at least one of the PEPs with the parameters corresponding to the security rule R. A test case which is impacted by a security rule is obviously impacted by the security policy.

3. Overview of the approach

Figure 2 presents the methodology. First we identify the test cases which are not impacted by the security policy and remove them from the initial set of test cases. Second, we analyze the impacted test cases in order to precisely associate the security rules impacting each test case. Then we adapt each functional test case by adding a new oracle function, specific to the security mechanism under test. This new oracle function checks that the correct PEP is executed and that the access is granted/denied consistently with the security policy.

In both steps, the analysis is performed dynamically. First, a single execution of all the test cases allows identifying which test case impacts at least an access control mechanism. It thus selects the test cases which are qualified for testing security, in order to reduce the effort needed to perform the second step analysis. This step optimizes the number of test cases to be run in the second step (especially when the test set is large).

In the second step, we perform a mutation analysis which requires executing the test cases for each mutant. We systematically inject errors in every security rule in order to create mutant PDPs. This analysis is a kind of sensitivity analysis [7], and allows locating precisely the relationships linking a test case to security policy rules in the PDP. If the behavior is the same when executing the test case on the initial and modified PDP, then it means it is not impacted. We execute every test case with every mutant. When a test case kills a mutant m, it means that it is impacted by the rule that has been modified to generate m. This mutation analysis thus allows us to associate each impacted test case with a set of security rules that are covered by this test case.

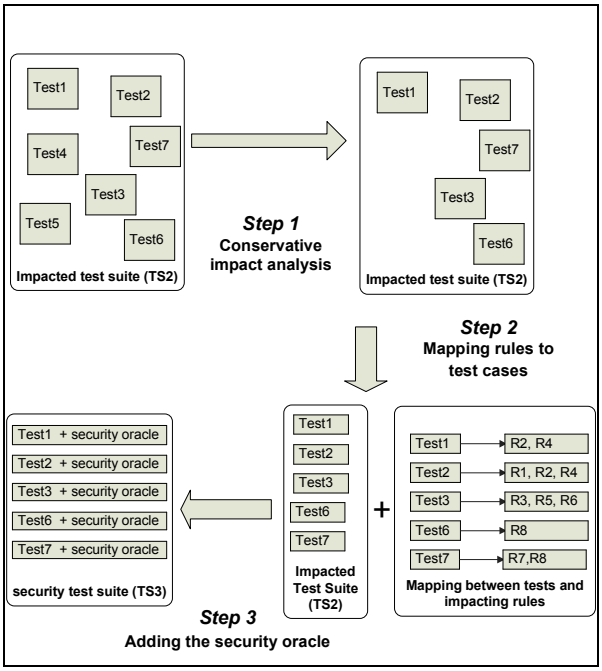

Figure 2 - Overview of the three steps

We assume that the PDP is *controllable*. Controllability means that the rules in the PDP can be easily modified. In this paper, we consider that the PDP is a separate component. This property is needed to modify the rules and check the sensitivity of the test cases execution to changes in the PDP. The PDP security rules can be implemented in several ways, e.g. in a database or in XACML. In this paper we use the environment proposed in [8] to modify the security rules in PDP implemented in several languages (OrBAC, RBAC).

Based on our previous work [5], we are able to claim that this assumption is realistic. By using our generic approach, we are now able to modify the rules of policies expressed in different language (RBAC, OrBAC or XACML).

4. Selecting test cases that trigger security rules

In the first step of our process, we perform a dynamic analysis to select the subset of test cases that are impacted by the SP. The objective of the first dynamic analysis is to filter the test cases which are *not impacted by the security policy.* One test case is considered to be impacted by the SP if it triggers a PEP during its execution. It is an optimization step which is used to make this second dynamic analysis less time and resource consuming. Experiments of Section 7 will show the benefits of using this first step (test cases selection) compared to a direct use of the second step (test cases impact analysis).

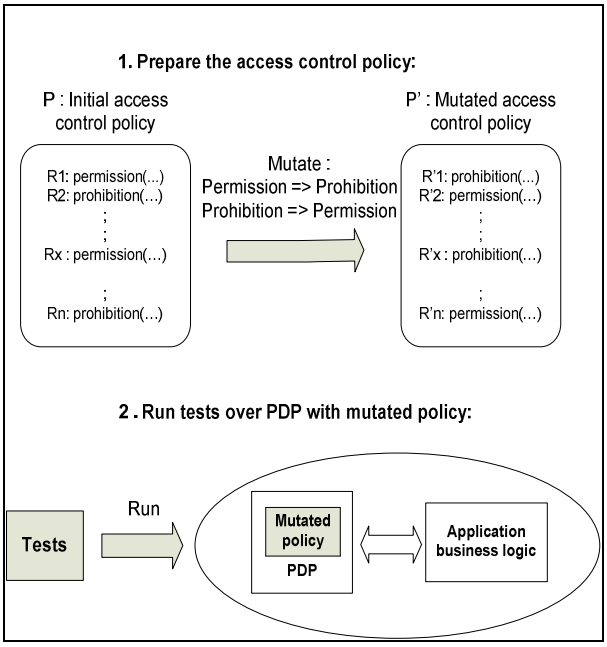

Figure 3 - First step: test cases selection

As shown in Figure 2, the first test selection involves two steps.

We start by producing a special mutant policy which replaces each rule from the initial policy with its opposite rule. We replace permissions with prohibitions and prohibitions with permissions, and we replace the default policy decision by its opposite as well. We end up with a mutated access control policy that contains only faulty rules. In the security mechanism, we use this policy instead of the initial one. Then tests are run on the system containing this new mutated policy in its security mechanism. Tests that exercise functionalities protected by the security mechanism are expected to fail due to the several errors injected in the policy. These tests are selected as impacted by the security policy because they fail if the policy is erroneous. This is a conservative approach, since we are sure that the test cases that kill mutants are impacted. We cannot assess that test cases that pass are not impacted by the security policy mechanism.

5. Relating test cases and security rules

To perform this analysis, we applied mutation adapted to access control policies, we proposed in a previous work [9]. The mutation approach is recalled, then the second step dynamic analysis (test cases impact on security policy rules) is detailed.

5.1. Mutation applied to security policies

Mutation analysis is a technique for evaluating the quality of test cases [10]. We proposed several mutation operators that are applied to the security policy model and are thus independent from implementation-specific details. This approach has the advantage of defining faults that are actually related to the definition of security rules (prohibition instead of permission, wrong role, etc.). We assume that there is a direct link between these SP rules (specification level) and the corresponding PDP (implementation level). This is why the PDP has to be controllable. In [8], a solution to derive directly the PDP from the security policy language has been presented based on a common metamodel to the languages and the mutation operators. Another

The mutation operators inject mutation faults to the SP. The approach was firstly applied to OrBAC (Organization Based access control [2]), and extended to RBAC and XACML like code [8]. Four types of mutation operators were proposed:

- Type changing operators: Turn one permission rule to a prohibition (PPR) or one prohibition to a permission (PRP).
- Parameter changing operators: Take one rule, and replace one of its parameters with a different one.
- Hierarchy changing operators: Replace a rule with one of its descendants.
- Adding rules operator: Add a new rule (ANR), using a combination of parameters and status that is not is the defined rules.

5.2. Detecting test cases impacted by a rule

In this second step, we map test cases to the security rules they trigger. The second step dynamic analysis is used to build these relationships. While it could be done with other techniques (e.g. by comparing the execution traces, regression testing techniques), we choose to use the mutation approach we proposed previously [9] to make this analysis possible. As shown in Figure 4, it has the advantage to be non intrusive in the business logic. No assumptions are needed con-

cerning the PEP 'observability' and the way the security mechanisms are coded in the business logic. In fact, the business logic is treated as a black box which interacts with the PDP. The idea is to perform some "sensitivity analysis" [7] by modifying the PDP security rules and check whether the execution of a test case is impacted by this change. Thus, the mutation analysis is used to create a variant version of the PDP: it could be done using Xie's mutation technique on XACML code [11] or even manually if the number of rules is not too large.

The executions (on the initial reference version and on the mutated ones) of a test case i are compared: a difference means that the change in the PDP has impacted the execution of test case i. In other words, it means that the initial execution triggers the PEP related to the security rule (recall that a PEP is a security mechanism which explicitly calls the PDP). At this stage, we don't need to know whether the initial execution result is correct or not from the security point of view (is the correct rule called? Is the result of the PDP taken into account by the business logic code?). It is just necessary to determine the test cases exercising a PEP related to a given security rule.

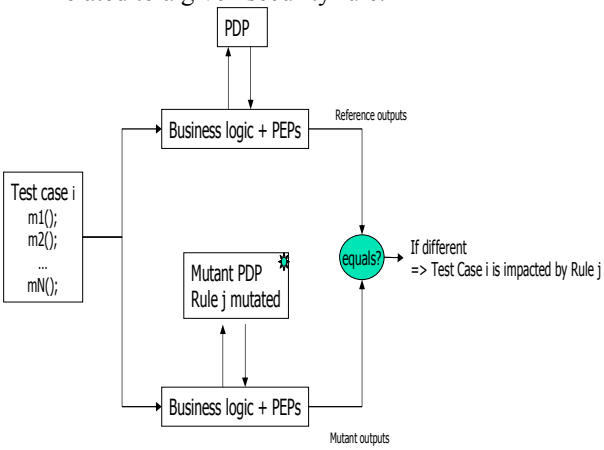

Figure 4 - Second step: dynamic analysis of test cases impact on security mechanisms using mutation

When a test case TC is executed, the PEP sends a request to the PDP that evaluates a security rule Rj which impacts the behavior of the test case: using mutation, we can determine that rule Rj is evaluated by the PDP when the test case is executed.

For detecting the impacted test cases by a security rule, the first operators are sufficient. The two other categories of operators could be used for a more precise analysis. The ANR operator is not used either since existing functional test cases are unlikely testing the robustness of the security policy in terms of default/unspecified behavior (which is the reason why the ANR operator is used). Since this operator is costly, we do not use it, even if it may be used to identify the test case which may be transformed in advanced security test cases (see [4]) for testing the robustness of the security policy.

We generate a set of mutant policies in the PDP by:

- Applying PPR (prohibition to permission) and PRP (permission to prohibition),
- Executing the test cases on each mutant.

If a test case fails with a particular mutant, this reveals that the test case is actually impacted by the rule that has been mutated in this mutant.

6. Oracle modification in the functional test case

We propose three different levels of quality to implement the oracle function. As stated in Table 1, the first level of oracle just checks that no obvious inconsistency exists between the current security policy and the implementation of the system. Level 1 oracle function is "black-box" in the sense no information is needed to build it except the status of the rule(s) impacted by the test case. If the tested rule corresponds to a prohibition, the oracle must check that an exception or a specific message is raised. For a permission, the oracle checks that no such exception is raised.

Table 1. The three quality level of oracle functions

Oracle level	Assumptions	Check that..
1	no (black box)	the access is granted/denied w.r.t. the rules (1)
2	Observable PDP ("Business logic + PEPs" is a black box)	- (1) - the PDP is correctly called (2)
3	Observable PDP and Observable PEP (glass-box)	- (1) - the PDP *and* the expected PEP are correctly interacting (the right PEP calls the PDP with the expected parameters).

Level 2 oracle extends the basic level 1 oracle function by observing the PDP logs. The oracle then relates the oracle 1 results with the execution of the expected rule in the PDP. This allows detecting hidden mechanism. For example if the PDP is bypassed due to a hidden mechanism, this is the only way to detect that a permission has been effectively granted by the PDP.

Level 2 oracle is especially useful to detect hidden security mechanisms.

Level 3 oracle extends level 1 and check that no unexpected interactions occur between the PEP which is expected to send a request to the PDP and the PDP. Level 3 oracle is thus a good way to detect interaction faults.

In this paper, we automated the weaving of level 2 oracle functions in the impacted test cases using AspectJ. To make this process possible, the security policy is taken as an input to determine whether the expected result is 'permission' or 'prohibition' (level 1 oracle). The automation is possible only because we force the PEP, which has been woven using AOP, to raise a specific exception when the access is prohibited. With such a constraint, capturing the access status is feasible as well as comparing it with the rule status of the security policy (specification). Figure 2 presents a test case after the modification. The PDP is observable and logs coming from the business logic are observed. The security oracle retrieves the PEP log and compares it with an expected log according to the rules that are impacting the test (obtained by the second step impact analysis).

```
/**
 * Test for method borrow
 *
 */
public void testBorrow() {

try {                                                          } Test sequence
// init test data
Book book = new Book("book title");
Teacher teacher = new Teacher("teacher1");
// teacher borrow book
bookService.borrowBook(teacher, book);

 // functional oracle                                         } Existing
 // test if borrowed and no longer reserved                   } functional oracle
 assertTrue(teacher.getBorrowed().contains(book));
 // test if data was well stored in DB
 bookReturned = bookDAO.loadBook("book title");
 asserEqualsTrue(bookReturned.getCurrentStateString()
,Book.BORROWED));
       } catch (BSException e) {
          fail(e.getMessage());
       } catch (SecuritPolicyViolationException e) {
          fail(e.getMessage());
       }
// Security oracle                                             } Woven level 2
// get the rules impacting this test                           } security oracle
String rules = getRulesImpactedByTests("testBorrow");
// get both expected and return log
String peplogExpected = getExpectedPEPLog(rules);
String pepLogReturned = readPEPLog();
// security oracle
assertEquals(pepLogReturned,peplogExpected);

  }
```

Figure 5 - Woven security oracle function

7. Experiments and results

The objectives of the experiments are to:

- Study the information provided by the two steps. In particular, we will study the interest of the first step of test selection in the methodology.
- compare various solutions to obtain security policy test cases:
  - Manual creation of SP test cases
  - Manual adaptation of functional test cases
  - Automated detection of impacted test cases and manual modification of the oracle
  - Fully automated approach.

7.1. Presentation of case studies

We applied our approach on three Java case studies applications previously developed by students during group projects and used in a computer science course in the university of Rennes 1:

LMS: The library management system (LMS) offers services to manage books in a public library (see [4] for details).

VMS: The virtual meeting system offers simplified web conference services. The virtual meeting server allows the organization of work meetings on a distributed platform.

ASMS (Auction Sale Management System): The ASMS allows users to buy or sell items online. A seller can start an auction by submitting a description of the item he wants to sell and a minimum price (with a start date and an ending date for the auction). Then usual bidding process can apply and people can bid on this auction. One of the specificities of this system is that a buyer must have enough money in his account before bidding.

The two last studies have been presented in [6] for studying the issue of hidden security mechanisms. Table 2 gives some information about the size of the three applications (the number of classes, methods and lines of code).

Table 2. The size of the three applications

	# classes	# methods	LOC (exec. statements)
LMS	62	335	3204
VMS	134	581	6077
ASMS	122	797	10703

7.2. Examples

Figure 6 presents an example of the approach. An access control rule Ri mutated to R'i by switching the rule status into "permission". The mutated policy makes the test case 'testBorrowBookInMaintenanceDay' fail because the applied rule is 'permission' while the rule expected to be applied is "prohibition". In other words, the case expected an exception or an error message for borrowing a book during a maintenance day. It does not get this behaviour due to the mutated rule that was applied instead of the correct one.

(1) Existing policy

R1
R2
;
Ri = prohibition(borrower,borrow,book,maintenance_day)
;

---

(2) Mutated policy

R1
R2
;
R'i = *permission*(borrower,borrow,book,maintenance_day)
;

---

(3) Impacted test found

Test case: testBorrowBookInMaintenanceDay()

Figure 6 – Example of rule and impacted test case

7.3. Results

In this section, we show the various steps of the process. The functional test cases have been produced either manually or using the use case system testing approach of [12]. The generation process stopped when 100% of code coverage was reached.

a First step impact analysis results

Table 3 shows the number/percentage of test cases which can be filtered thanks to the first step of test filtering. The execution times are less than one second for the three applications (between 0.5s and 1s) and thus can be considered as negligible in our studies. Around 20 to 30 % of the test cases can be identified not impacted by the access control policy.

Table 3. First step analysis results

System	Impacted Tests	Other Tests	All
VMS	41	11 (**21%**)	52
ASMS	85	38 (**31%**)	123
LMS	19	7 (**27%**)	26

The first step allows saving some execution time when using the second step dynamic analysis. We measured the execution time with and without the first step of test selection. Without the first step, the impact analysis is performed on all tests cases, and then only on test cases selected by the first step. As shown in Table 4, the relative saved time is significant (and reflects the percentage of filtered test cases). This first step reduces the execution time of the second one. The execution time of the first step (test selection) is neglected. For instance, on a real enterprise information system, the execution time of a test case may be important, and a relative 20-35 % of saved execution time becomes interesting.

Table 4. Execution time with/without the first analysis

Application	Without 1st step	With 1st step	% Saved time
VMS	30.5 s	24,3 s	**20.4 %**
SMS	114.1 s	74.6 s	**34.6 %**
LMS	19.3	16.4	**15 %**

b Second step: test impact analysis results

Table 5 shows examples of the results that we get by this analysis. For each test case, the analysis pinpoints the exact rules that are impacting it.

Table 5. Examples of detailed results

Tests	# rules	Rules
Test for the LMS: testBorrow2Borrower()	4	1. Permission(Teacher, GiveBack, Book, WorkingDays) 2. Permission (Teacher, Borrow, Book, WorkingDays) 3. Permission (Student, GiveBack, Book, WorkingDays) 4. Permission (Student, Borrow, Book, WorkingDays)
Test for VMS: testUpdateUserAccount()	3	1. Permission (Webmaster, Update, Account, MaintenanceDay) 2. Permission (Admin, Update, Account, MaintenanceDay) 3. Permission (Personnel, Update, Account, MaintenanceDay)
Test for ASMS: testOpenSale_0()	2	1. Permission (Seller, CreateSale, Bid, default) Permission (Seller, Update, Bid, default)

In addition to the previous table, it is possible to find out for each rule the corresponding impacted tests.

Furthermore, we can analyze more the results and observe the number of tests that are impacted by one rule or two or three etc. In Table 6, we show the result for the VMS application. For instance, there are 9 tests that are impacted by only one rule (each test is impacted by only one rule).

Table 6. Tests and impacting rule for VMS application

# Tests	# Impacting Rules
9	1
12	2
2	3
9	4
1	5
5	6
1	7
1	8

7.4. Comparing several degrees of automation

We conducted an experiment to compare between manual and automated approaches and to estimate the

efforts in terms of time spent in obtaining the final test suite. We considered 4 scenarios:
(1) Manually creating, from scratch, all the sec. tests,
(2) Adapting manually existing tests (without step 1),
(3) Adapting manually only selected tests (using the first step of the approach),
(4) fully automated approach (steps 1, 2 and 3).

Two graduate students were in charge of performing these scenarios. Not all the test cases have been created manually since more than 50% of functional test cases were generated using the use case driven approach of [12]. However, the students reported the time spent in the remaining manual tasks. We count in hours of intensive work, which is a lower bound of the real effort that would be needed. Table 7 displays the results showing the interest of the automation. In the three cases studies,, the creation of test cases dedicated to security is very important since it includes the identification of what should be tested and the elaboration of the test sequence. The cost of adapting the test cases selected using the steps 1 and 2 dynamic analyses is still important. Even if allows to significantly reduce the work of analysis which is needed to determine manually which functional test is impacted by a security rule, the remaining task consists in modifying systematically the oracle in the test cases. The fully automated approach is very efficient but can only be applied in a process where the PEP sends specific messages when an access is denied. The install time of the various tools and environments for performing the automated treatments has not been measured but is done once. Even taking into account the bias due to the expertise degree of the students (and all other subjective factors), the benefit of automation seems high.

Table 7. Automated and manual approaches

	LMS	VMS	ASMS
(1)Creating all tests	32 hours	48 hours	64 hours
(2)Adapting manually all existing tests	8 hours	16 hours	24 hours
(3)Adapting selected tests	1 hour	3 hours	4.5 hours
(4)Fully automated	5 min + install time	10 min + install time	13 min + install time

8. Related works

As far as we know, no previous work studies how to automatically adapt and reuse functional tests for SP testing. However, several works proposed techniques and tools for automatically testing the PDP implementation for security policies written in XACML [13, 14] or RBAC [15]. Fisler et al. proposed Magrave a tool for analyzing XACML policies and performing change-impact analysis [16]. The tool can be used for regression testing to identify the differences between two versions of the policies and test the PDP. In [17], Xie et al. proposed a new tool Cirg that automatically generates test for XACML policies using Change-Impact Analysis.

The main difference between their work and ours is that they focus on the PDP alone and testing consists of sending request and getting responses that are compared to the expected ones. We consider the whole system (PDP+PEP) and adapt functional tests to validate the implementation of the security policy. Both works are thus complementary. In fact, it is necessary to begin with making sure that the implementation of the PDP is correct and that the PDP responds as expected according to the specification, before considering the whole system. However, it is obvious that testing the PDP in a separate way does not guarantee that the application behind it actually implements correctly the access control policy. We cannot assume that the application behaves as expected by the access control policy by only checking the front door (the PDP+PEP). For instance, the internal application code (the business code) may contain backdoors that bypass the PDP. Testing the PDP independently does not allow these backdoors to be detected and removed. In addition, as we have previously pointed out [5], applications may contain hidden doors (that we called hidden mechanisms) which implement access control rule in a non-documented ways. These hidden doors can be detected only by testing the whole system w.r.t. the access control policy. For all these reasons, it is absolutely important to test the whole system.

Even if the idea of estimating the impact of test cases is not new (see [18] for design choices), no previous work specifically studies how to automatically adapt and reuse functional tests for SP testing.

9. Conclusion and Future Work

In this paper, we presented a new automated approach for selecting and adapting the functional tests in the context of SP testing. The approach includes a three steps process. The first step uses a special mutant of the policy having all its rules mutated. This step helps selecting the subset of tests impacted by the access control policy. The second step uses single mutations to provide a mapping between tests and the triggered SP rules. According to the mapping, AOP is used to transform them into security test cases by weaving the security policy oracle function. The approach was successfully implemented and applied to three case studies. The experimental results show the effectiveness of the approach when compared to a manual one.

In further work, we will study regression testing in the case of a security policy evolution: the second step (impact analysis) can be used for selecting and adapt-

ing security tests with respect to the security policy modification. In addition, one of the main issues is access control tests automation. This issue can be addressed using combinatorial [19] or computational intelligence algorithms [20, 21] and will be studies in future work.

Acknowledgments: This work is supported by "Région Bretagne" (Britanny Council) through a contribution to a student grant.

10. References

1. Fabio Martinelli, Paolo Mori, Thomas Quillinan, and Christian Schaefer, *A Runtime Monitoring Environment for Mobile Java*, in *(secTest2008) 1st International ICST workshop on Security Testing* 2008.
2. A. Abou El Kalam, et al., *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
3. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
4. Y. Le Traon, T. Mouelhi, and B. Baudry, *Testing security policies : going beyond functional testing*, in *ISSRE'07 : The 18th IEEE International Symposium on Software Reliability Engineering*. 2007.
5. T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon, *A model-based framework for security policy specification, deployment and testing*, in *MODELS 2008*. 2008.
6. Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, *Test-Driven Assessment of Access Control in Legacy Applications*, in *ICST 2008: First IEEE International Conference on Software, Testing, Verification and Validation*. 2008.
7. J. M. Voas, *PIE : A Dynamic Failure-Based Technique.* IEEE Transactions on Software Engineering, 1992. **18**(8): p. 717 - 727.
8. T. Mouelhi, B.Baudry, and F. Fleurey, *A Generic Metamodel For Security Policies Mutation*, in *SecTest 08: 1st International ICST workshop on Security Testing*. 2008.
9. T. Mouelhi, Y. Le Traon, and B. Baudry, *Mutation analysis for security tests qualification*, in *Mutation'07 : third workshop on mutation analysis in conjuction with TAIC-Part*. 2007.
10. R. DeMillo, R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
11. E. Martin and T. Xie. *A Fault Model and Mutation Testing of Access Control Policies*. in *Proceedings of the 16th International Conference on World Wide Web*. 2007.
12. Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel, *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
13. E. Martin and T. Xie., *Automated Test Generation for Access Control Policies via Change-Impact Analysis*, in *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*. 2007.
14. Vincent C. Hu, E. Martin, J. Hwang, and T. Xie. *Conformance Checking of Access Control Policies Specified in XACML*. in *Proceedings of the 1st IEEE International Workshop on Security in Software Engineering*. 2007.
15. A. Masood, A. Ghafoor, and A. Mathur, *Technical report: Scalable and Effective Test Generation for Access Control Systems that Employ RBAC Policies*. 2005.
16. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. *Verification and change-impact analysis of access-control policies*. in *ICSE*. 2005.
17. E. Martin and T. Xie. *Automated Test Generation for Access Control Policies via Change-Impact Analysis*. in *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*. 2007.
18. G. Al-Hayek, Y. Le Traon, and C. Robach, *Impact of system partitioning on test cost.* IEEE Design & Test of Computers, 1997. **14**(1): p. 64-74.
19. A. Pretschner, T. Mouelhi, and Y. Le Traon, *Model-Based Tests for Access Control Policies*, in *ICST 2008*. 2008.
20. B. Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. *Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components*. in *ASE'02, 2002* Edimburgh, Scotland, UK: IEEE Computer Society Press, Los Alamitos, CA, USA.
21. B. Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon, *From Genetic to Bacteriological Algorithms for Mutation-Based Testing.* Software Testing, Verification and Reliability, 2005. **15**(1): p. 73-96.