Testing security policies: going beyond functional testing

Yves Le Traon, Tejeddine Mouelhi
GET-ENST Bretagne
35576 Cesson Sévigné Cedex, France
{yves.letraon,tejeddine.mouelhi}@enst-bretagne.fr

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

Abstract.

While important efforts are dedicated to system functional testing, very few works study how to test specifically security mechanisms, implementing a security policy. This paper introduces security policy testing as a specific target for testing. We propose two strategies for producing security policy test cases, depending if they are built in complement of existing functional test cases or independently from them. Indeed, any security policy is strongly connected to system functionality: testing functions includes exercising many security mechanisms. However, testing functionality does not intend at putting to the test security aspects. We thus propose test selection criteria to produce tests from a security policy. To quantify the effectiveness of a set of test cases to detect security policy flaws, we adapt mutation analysis and define security policy mutation operators. A library case study, a 3-tiers architecture, is used to obtain experimental trends. Results confirm that security must become a specific target of testing to reach a satisfying level of confidence in security mechanisms.

1. Introduction

For an organization, the security policy (SP) addresses constraints on access to data or functions by external users (or programs) and by people belonging to the organization. Such constraints are expressed using one of several access control models (RBAC[1-3], OrBAC [4, 5]) For a system to be developed, all these models describe the permissions or prohibitions for people to any of the resources of the system (it may apply to configure a firewall as well as to define who can access a given service or data in a database). The most advanced models ([4, 8, 9]) express rules that specify permissions or prohibitions that apply only to specific circumstances, called contexts. For instance, in the health care domain, physicians have special permissions in specific contexts, such as the context of urgency. Also, some models provide means to specify the different security policies applicable to the various parts of an organization (sub-organizations). At the end of this specification process, the SP specifies what the permissions and prohibitions should be in the system, in function of contexts, roles and views.

The software development process then consists in building a system according both to the functional requirements and the SP. In most cases, the deployment of a SP is not automated and the correctness of its implementation has to be verified or tested. In that context, testing can be applied as a technique to get some evidence that the SP implementation is correct with respect to the requirements.

In this paper, we introduce SP testing as a necessary step that takes place at the end of the development process. We build test cases from the SP specification and run those tests on the system to reveal security flaws or inconsistencies between the policy and the implementation. In practice, the difficulty is that any SP is strongly connected to system functionality: testing functions includes exercising many security mechanisms. The issue is to determine:

- whether testing functions provide enough confidence in the security mechanisms,
- how to improve this confidence by selecting test cases specific to security.

We study how to select SP test cases to reveal security flaws. We propose two strategies for producing SP test cases: in addition to existing functional test cases or independently from them.

The first step when applying testing to a specific context, such as security testing, is to analyze which security flaws can occur in this specific context, and to define test criteria, to qualify a set of test cases.

We propose and discuss what SP testing is and propose test criteria to generate test cases from an access control model. To obtain an experimental basis and compare the criteria and the strategies, a fault/flaw model for security policies is defined. It leads to an adaptation of mutation analysis [6] and thus provides a security testing qualification method. This flaw model is expressed in the form of mutation operators to be applied on the SP mechanisms. Using mutation analysis on a case study (a library system implemented with a 3-tiers architecture), we compare and discuss the criteria and strategies and highlight the specific issues related to SP testing, The most important contribution of the paper consists of demonstrating that testing a SP is a task distinct (but tightly related) from functional testing. The corollary contributions are a first definition of this testing task, the security flaws model, and first SP test criteria and strategies that still need to be improved, as shown by the case study results.

Section 2 introduces the context of the paper and defines important notions for security testing. Section 3 presents an example and the chosen SP model called OrBac. Section 4 proposes a method to qualify SP test cases. In section 5, an empirical study reveals the first results on the proposed criteria and strategies, and shows various issues which arise when testing security.

2. Process and definitions

In this section we present the overall process to derive test cases from requirements and propose precise definitions for the notions used in this paper.

Requirements for a software system include the functional description of the system as well as many extra-functional concerns (performances, real-time, availability,…). Among these concerns, the security aspects are often mixed with the functional ones. The requirement analysts have to extract these aspects and express, on one hand the use cases and the business model and on the other hand explicit the security policy (SP) in the form of an access control model. A policy defines a set of security rules that specify rights and restrictions of actors on parts and resources of the system. The SP may introduce specific concepts, and reuse most of the concepts and functions identified in the business model. The new concepts introduced for security implementation are taken into account in the refinement process, either during design (SecureUML [7]) or at deployment/coding steps.

The Figure 1 highlights the fact that both the code and the test cases are produced from the requirements by independent ways. The important point is that the SP test cases are obtained using the access control model, while the functional test cases are derived only from the use cases and business model. SP test cases are not only dependent on the SP but also refer to the use cases and the business model. In this paper, the functional test cases (or system tests in the sense of Briand's work [8]) are generated using the approach presented in [9], based on the use cases improved with pre- and post-conditions, called contracts. The functional test cases cover all the code implementing the functions of the system. We will study how functional test cases can be reused for testing security mechanisms.

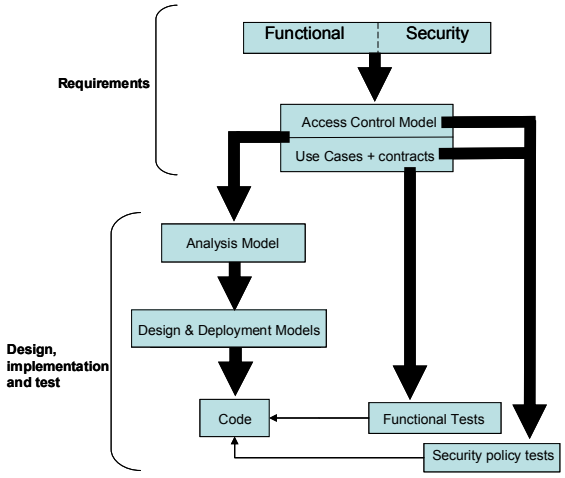

Figure 1. Security policy tests generation

2.1. Definitions

The following definitions help clarifying the differences between functional and SP testing.

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 [8, 9]. 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 which are derived specifically from a SP. The objective of SP testing is to reveal as many security flaws as possible.

Security flaw: A security flaw is the equivalent of a fault for functional testing. It corresponds to an inconsistency between the SP (the specification) and a security mechanism (the implementation) which is revealed at runtime.

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.

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.
- For a prohibition, the oracle checks that the service is not activated.

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.

3. Running the process on an example

This section introduces an example based on a library management system. In this work we use OrBAC [4, 5] as a specification language to define the access control rules (a set of rules specifies a SP). Based on the simplified requirements of this system, it is possible to derive a set of access control rules using the OrBAC model. We use these rules to illustrate the main features of the OrBAC language.

3.1. Library management system

The purpose of the library management system (LMS) is to offer services to manage books in a public library. The books can be borrowed and returned by the users of the library on working days. When the library is closed, users can not borrow books. When a book is already borrowed, a user can make a reservation for this book. When the book is available, the user can borrow it. The LMS distinguishes three types of users: public users who can borrow 5 books for 3 weeks, students who can borrow 10 books for 3 weeks and teachers who can borrow 10 books for 2 months.

The library management system is managed by an administrator who can create, modify and remove accounts for new users. Books in the library are managed by a secretary who can order books, add them in the LMS when they are delivered. The secretary can also fix the damaged books n certain days dedicated to maintenance. When a book is damaged, it must be fixed. While it is not fixed, this book can not be borrowed but it can be reserved by a user. The director of the library has the same accesses than the secretary and he can also consult the accounts of the employees.

The administrator and the secretary can consult all accounts of users. All users can consult the list of books in the library.

3.2. Modeling a security policy with OrBAC

In parallel of the design model, it is possible to model the SP from the requirements. We distinguish 5 different roles: public users, students, teachers, administrator and secretary. It is also possible to identify several rules that authorize or forbid access to data or services of the LMS.

OrBAC allows defining a set of security rules. A rule can be a permission, prohibition or obligation. Each rule has 5 parameters (called entities): an organization, a role, an activity, a view and a context. To increase modularity for the definition of security rules, OrBAC enables the definition of hierarchies of entities. In that case, rules defined on high level entities are inherited by the sub-entities. From the primary rules, secondary rules are derived, as illustrated in the following.

From the LMS requirements, we identify the entities displayed in Figure 2. The graphical representation is the one from MotOrBAC, the tool implementing the OrBAC model. First, we identify the services which are constrained by security rules. They are called activities in OrBAC. All users can perform three activities: borrow, reserve, return a book. Since all these activities are associated to the same rules, we define a high-level activity called `BorrowerActivity`. This means that all rules defined on the super activity will apply on sub activities. There are also a number of administrative tasks that can be executed. Since the administrator is allowed to execute all these tasks we define a super activity `AdminActivity`. The activities that inherit from `PersonnelActivity` are the activities that are permitted for the director and the secretary.

It is interesting to notice that the service for consulting the list of books is not modeled as an activity in the access control model. Since everyone can access this service, there is no restriction for this access, thus there is no rule related to this service.

Concerning the data (called views in OrBAC) that are restricted with access control, we identify the notion of book and of account. There are two types of accounts: borrower and personnel accounts.

The notion of context corresponds to a temporal (or spatial) dimension that appears in access control rules. In the requirements, we clearly identify `Holidays`, `WorkingDays`, `MaintenanceDay` and a `default` context which is used to define rules that are related to no specific time.

The last type of entities that needs to be defined concerns the roles. In the requirements we distinguish two main categories of roles: the users and the

administrative personnel. We define the `Borrower` role which captures the role of user. Since `Teacher` and `Student` are two specific types of users, we model them as two sub-roles for `Borrower`. Concerning the personnel, we distinguish three categories: `Administrator`, `Secretary`, and `Director`.

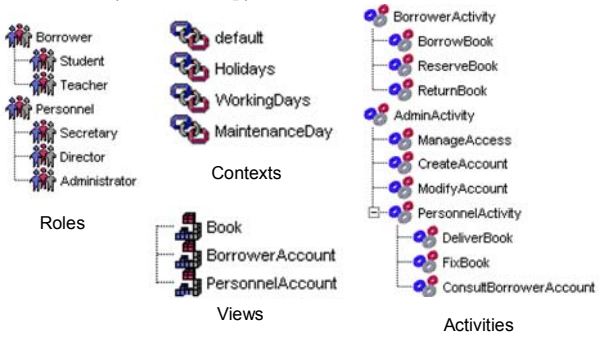

Figure 2. OrBAC entities for the LMS

Once all the entities are defined, they can be used to specify access control rules. Again, these rules are derived from the requirements. For example, requirements specify that users are allowed to borrow books only when the library is opened. This rule is defined as follows:

```
Permission(Library, Borrower,
BorrowerActivity, Book, WorkingDays)
```

The first parameter for this permission rule is the organization. Since there is only one organization in our example we give a very generic name `Library`.

Other example of rules include the prohibition to borrow a book during holidays, the permission for an administrator to manage the accounts for the personnel and the borrowers or the permission for the secretary to consult borrower accounts. These examples can be expressed as follows:

```
Prohibition (Library, Borrower,
BorrowerActivity, Book, Holidays)

Permission(Library, Administrator,
ManageAccess, PersonnelAccount, default)

Permission(Library, Administrator,
CreateAccount, BorrowerAccount, default)

Permission(Library, Secretary,
ConsultBorrowerAccount, BorrowerAccount,
default)
```

From the *explicit* (or *primary rules*), *secondary rules* are derived based on the parameters hierarchy. For example the rule:

```
Permission(rennesLibraries,Borrower,BorrowerAc
tivity,Book,WorkingDays)
```

Based on hierarchies shown in Figure 2, 6 additional rules are automatically derived. The BorrowerActivity is replaced by each possible sub-activity (borrow, rower is replaced by Student or Teacher.

The primary rule which is derived into these six secondary rules becomes useless from a testing point of view, if all the secondary rules are tested. We call such rules *generic rules* since they have no related security mechanism in the system under test.

One issue when specifying control access rules is that *conflicting rules* may appear. These conflicts can occur when 2 opposite rules have exactly the same parameters.

For example the following two rules are conflicting:

```
Permission(Library, Borrower,
BorrowerActivity, Book, WorkingDays)

Prohibition(Library, Borrower,
BorrowerActivity, Book, WorkingDays)
```

It is important to point out that conflicts may also occur when the parameters of the rule are not exactly the same. In the case a parameter in one rule inherits from a parameter in an opposite rule, the two rules are conflicting.

For example, in the following two rules, Teacher inherits from Borrower, thus the rules are conflicting:

```
Permission(Library, Borrower,
BorrowerActivity, Book, WorkingDays)

Prohibition(Library, Teacher,
BorrowerActivity, Book, WorkingDays)
```

An important benefit of using OrBAC is that the rules can be processed by a tool called MotOrBAC that automatically detects the conflicting rules in the definition of a SP, using an underlying Prolog motor. One way to solve the conflicts consists in assigning priorities to rules. The rule with the highest priority is executed.

For the LMS, 20 access control primary rules have been expressed. The total number of secondary rules is 22 and 7 rules are generic. So, the total number of rules is 42, 35 corresponding to a security mechanism.

4. SP test cases selection

For SP testing, the question is to ensure that security mechanisms are covered not only by input test sequences but are also exercised in every way that may lead to a failure of the policy.

In this section, we propose several test criteria to select test cases from an OrBaC specification. We also consider two strategies to produce efficient SP test cases w.r.t. the criteria.

4.1. Test criteria

Two test criteria are studied in this paper to select SP test cases from an OrBAC SP model.

CR1 - The criterion 1 (CR1) is satisfied iff a test case is generated for each primary access control rule of the security model.

In the case of the LMS, we have 20 such primary rules. In the case of a generic rule, a test case testing one instance of this rule is considered as sufficient.

CR2 - The criterion 2 (CR2) is satisfied iff a test case is generated for each primary and secondary rule of the security model, except the generic rules.

In that case, 35 test cases are generated corresponding to the 42 total access control rules minus the 7 generic ones. The CR2 criterion is stronger than CR1, since it forces the coverage of all secondary rules.

Advanced SP test cases - A*dvanced SP test cases* that exercise the default/non specified aspects of the SP.

These test cases are selected to kill mutants generated with a specific mutation operator (ANR) that will be presented in the next section.

Functional test cases: It corresponds to system tests, in the sense they are generated based on the use cases of the system under test.

In the case study, we used the approach based on use cases + contracts (pre- and post-conditions) to generate the functional test cases. The automated generation is obtained using the UCTS (Use Case Transition System) presented in [9]. The generated test cases cover the nominal code (code implementing the specified use of use cases) and a part of the robustness code of the system under test (unexpected use of a use case and specified situations when the use case execution fails).

Security test cases obtained with CR1 or CR2 should test aspects which are not the explicit objective of functional tests, e.g. that all prohibition rules that are not tested by functional tests.

4.2. Test strategies

In this paper, we study whether the functional test cases can be used for SP testing. Reusing functional test cases implies adapting them for explicitly testing the SP. The intent of the functional test becomes *security* and details the SP rules which are tested by the input test sequence. The test oracle does not check the correctness of the service results, but interrogates the security mechanism and checks if the expected permission/prohibition is executed.

So, we consider two types of strategies depending whether we reuse the existing test cases or not.

Incremental strategy: It denotes the strategy for producing security test cases which reuse existing test cases.

An example of incremental test strategy consists of reusing functional test cases, then completing them with one of the CR1 or CR2 criteria, and finally completing the resulting test suite with advanced test cases.

Independent strategy: This strategy consists of selecting functional, SP test cases and Advanced SP test cases independently.

We will compare and discuss in the case study several incremental strategies and the independent one. The goal is to highlight the many issues that arise when selecting test cases for the SP aspect.

4.3. Test Qualification

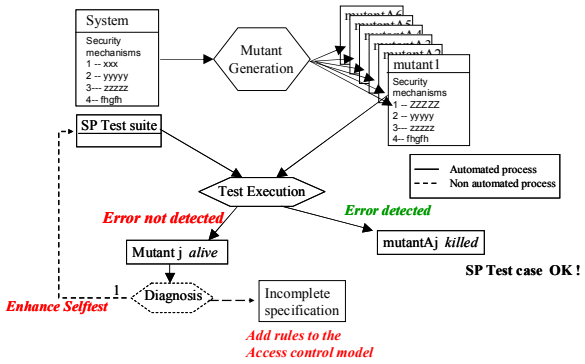

Figure 3. Qualifying security tests

Figure 3 presents the process for improving a SP test cases suite so that it reaches a satisfying level, in terms of capacity in detecting security flaws. When flaws have been corrected by initial test cases, the test qualification loop consists of improving the test cases until all security flaws are detected. A security mutant differs from the initial code by the introduction of a single flaw in the SP implementation. The mutation score corresponds to the proportion of faulty versions of the system which are detected (or "killed") by the test cases. A set of test cases is satisfying to test a SP if it kills all the security mutants. The improvement process ends when the 100% mutation score is reached. The improved SP test cases can then be reapplied on the non-mutated code to check whether actual flaws are detected.

In [10], we have defined mutation operators for security tests qualification. For the analysis presented in this paper, we reuse those operators.

4.4. Mutation operators

Mutation analysis is a technique for evaluating the quality of test cases. We propose to apply it in the context of security test cases generation. In order to apply mutation analysis to security tests we need to find suitable mutation operators. We express high level operators that will be applied to security policies and generate faulty versions of a SP using OrBAC. A

mutant is a copy of the original policy that contains one simple flaw.

a Security mutant operators

We identified 8 mutant operators classified in 4 categories.

Rule type changing operators

The following operators can be used in order to create mutants for security policies by changing predicate type:

PRP : Prohibition rule replaced with a permission one.

PPR : Permission rule replaced with a prohibition one.

To illustrate rule type changing operators, we present 2 examples to show how they modify the SP:

PPR:

Rule used: `Permission (rennesLibraries, Administrator, ModifyAccount, BorrowerAccount, default).`

Rule to use instead: `Prohibition (rennesLibraries, Administrator, ModifyAccount, BorrowerAccount, default).`

PRP:

Rule used: `Prohibition (rennesLibraries, Secretary, ModifyAccount, BorrowerAccount, default).`

Rule to use instead: `Permission (rennesLibraries, Secretary, ModifyAccount, BorrowerAccount, default).`

Rule parameter changing operators

The following operators replace parameters of predicates:

CRD: Rule context is replaced with another context.

RRD: Rule role is replaced with another role

Some examples:

CRD: Rule to use: `Permission(rennesLibraries, Secretary, FixBook, Book, MaintenanceDay).`

Rule to use instead: `Permission (rennesLibraries, Secretary, FixBook, Book, HoldaysDays).`

RRD: Rule to use: `Permission (rennesLibraries, Administrator, ManageAccess, PersonnelAccount, default).`

Rule to use instead: `Permission (rennesLibraries, Teacher, ManageAccess, PersonnelAccount, default)`

Rule adding or removing mutation operators

A mutant with an additional security rule can simulate a security hole. In fact, the SP will be more permissive or more repudiating. The mutation operator introduced:

ANR : New rule added.

Some examples of ANR:

Rule added: `Permission (rennesLibraries, secretary, BorrowBook, Book, WorkingDay).`

Rule added: `Permission (rennesLibraries, Secretary, DeliverBook, Book, Holidays).`

Hierarchy mutant operators

OrBAC implements abstract entities inheritance. This means that we can specify an entity, e.g., "borrower" (here as a role) then its sub-entities, e.g., "student". Rules defined for the "borrower" will be automatically applied to "student". If we have this predicate:

permission(Library,Borrower,borrow, book, WorkingDays)

Then we have:

permission(Library,Student,borrow, book, WorkingDays)

We can create mutants by replacing a parent entity by one of its descendants or by replacing a child by one its ancestor.

RPD: Rule role replaced with sub-role.

APD: Rule activity replaced with sub-activity.

Some examples:

RPD : Rule used: `Permission (rennesLibraries, Borrower, ReserveBook, Book, WorkingDays).`

Rule to use instead: `Permission(rennesLibraries, Student, ReserveBook, Book, WorkingDays).`

APD: Rule used: `Permission (rennesLibraries, Student, BorrowerActivity, Book, WorkingDays).`

Rule to use instead: `Permission (rennesLibraries, Student, ReserveBook, Book, WorkingDays).`

5. Case study and results

The case study application has a typical 3-tiers architecture widely used for web applications. It is representative of the web applications. Figure 4 presents the main characteristics of this architecture. The application contains 3204 lines of code, 62 classes and 335 methods.

The security mechanisms implementing the OrBAC based SP are located in the business layer, in the service methods. Before performing the requested action, the method calls the security mechanism that checks if the SP allows of forbids that action. When the action is prohibited a SP violation exception is raised and thrown to the caller, otherwise (when the action is allowed) nothing is done and the execution of the action is pursued by the service method. There are two possible behaviours:

- The requested action is allowed: The security mechanism let the application continue normally.
- The requested action is prohibited: The security mechanism raises a security exception that interrupts the execution of the method. This exception is then handled by the caller.

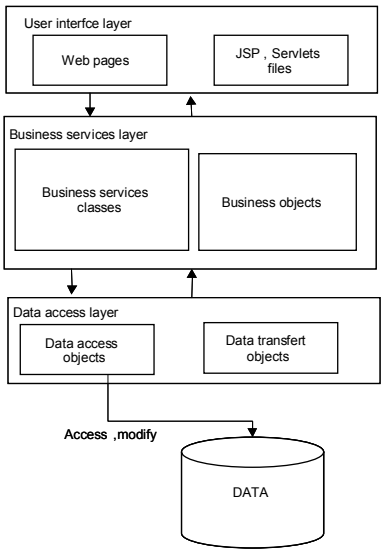

Figure 4. A 3-tiers architecture for the LMS

When we implemented the SP, we maintained traceability links between the OrBAC rules and the corresponding blocks of code. This means that we are able to modify the security rules used by the application and introduce a mutant rule. The system's implementation defines a data structure that contains all the permissions and prohibitions defined in the OrBAC policy. The creation of mutants can be automated because the rules are stored in a unique variable. Thus the introduction of flaws consists in modifying this variable.

We generated 42 functional tests and 189 security tests. To illustrate the difference between functional and security tests, the following example shows two different test cases for borrowing a book:

Functional – a functional test case will make a borrower borrow and return a book. *Intent*: functional, test borrow and return for a unique borrower. *Oracle*: check that the book is available after it has been returned

Security Policy – a SP test case will check that a borrower can borrow a book to the library during the working days. *Intent*: security, permission for a borrower to borrow a book the working days. *Test sequence:* create the context of a working day, make a borrower borrow a book. *Oracle:* interrogate the security mechanism and check that the permission has been computed and given to the borrower.

The mutation analysis runs as follows:

- Tests are executed against the implementation of the initial SP.
- The oracle function associated to the test case checks that the service is activated (permission) or not (prohibition).

5.1. Generated mutants

Table 1 gives the number of generated mutants per operator. The ANR operator generates much more mutants since it adds a non specified security rule. This number may vary, depending on the completeness of access control rules. The fewer rules are specified, the more mutants this operator generates. The number of ANR mutants thus reflects the fact that all the possible cases have not been specified in the SP. To our own experience, this is quite usual: the specification focuses on the most critical and important cases, and often considers that default behaviour is acceptable. Testing these cases allow the default policy to be exercised and allow highlighting lack in the specification. On the other hand, there are few mutants generated from hierarchy changing operators because the specification does not introduce many hierarchical entities. The basic mutation operators (i.e. all operators except ANR) will generate more mutants when more rules are added. In a general case, there is thus a balance in the number of generated mutants between basic operators and ANR.

Table 1. # mutants per mutation type and operator

Operator category		Operator name	Number of mutants
Basic Mutation operators	Rule type changing	PPR	22
		PRP	19
	Rule parameter changing	RRD	60
		CRD	60
	Hierarchy changing	RPD	5
		APD	5
Rule adding operator		ANR	200
Total			371

5.2. Issue 1: Relationships/differences between functional and security tests

The first experiments we performed aim at studying whether the faults detected by functional test cases differ from (are included in or intersect with) the ones detected by test cases dedicated to the SP. While the functional tests, which cover 100% of the code, necessarily execute security mechanisms, they should only focus on the success/failure of method/service sequence executions.

Table 2. Mutation analysis results by test cases category

	functional		CR1		CR2	
#Test Cases	42		20		35	
Rule type changing operator	35/41	85%	38/41	92%	41/41	100%
Rule parameter changing operator	90/120	75%	101/120	84%	120/120	100%
Hierarchy changing operators	10/10	100%	10/10	100%	10/10	100%
Overall score with basic security operators	135/171	**78%**	149/171	**87%**	171/171	**100%**
Rule adding operator (ANR)	22/200	**11%**	28/200	**14%**	33/200	**17%**
Overall score with all operators	157/371	**42%**	177/371	**47%**	204/371	**55%**

When reusing functional test cases with the objective of testing security, the intent changes and thus it is necessary to modify the associated test oracle. This task may be costly in the general case, depending on the difficulty to relate the functional test sequence to the security mechanisms it should exercise.

There are 42 test cases. Some test cases (7 test cases) do not trigger the security mechanisms. It corresponds to a code that is not related to the SP. This confirms the intuition that functional test objectives differ from the objective of explicitly testing security mechanisms.

Table 2 second column shows that the functional test cases, adapted to security, can kill most (78%), but not all, basic operators' mutants. Concerning, the ANR operator, the functional test cases are not efficient (11%): it is due to the fact that the ANR operator generates security flaws which are outside the scope of the specification. In conclusion, the overlap of functional aspect and SP is high, but the functional test cases do not kill all security mutants. It appears as a meaningful task to generate test cases with the explicit objective of testing the SP.

5.3. Issue 2: Comparing test criteria

Table 2 third and fourth columns present the mutation results for CR1 and CR2 test criteria. The 20 test cases selected with CR1 are more efficient than the functional test cases, since with fewer test cases the final mutation score is higher. However, this criterion is not efficient to reach a 100% mutation score on basic mutants (generated with all operators except ANR).

In conclusion, the CR2 criterion is necessary to provide a full coverage of basic mutants, and appears as good trade-off between functional and CR1, in terms of efficiency (mutation score) and cost (#test cases) for detecting security flaws. A corollary conclusion is that a large number of basic mutants are quite easy to kill. Since the hard-to-kill mutants are the most interesting ones (because they require the most efficient test cases), another study would consist of ranking the mutation operators so that only hard-to-kill mutants are generated. This study is beyond the objective of this paper but is necessary to offer a realistic mutation-based approach to test security policies. This paper focuses on the test selection problem for SP and the question of sufficient mutant operator does not impact the conclusions we draw (in the worse case, we consider more mutants than necessary since some mutants are coupled).

We remark that the CR2 criterion allows covering up to 17% of the ANR mutants. While basic security mutants force the tests to cover the specified security rules, the ANR ones force to check the robustness of the system in case of default or underspecified policies. Combining both operators provides a good criterion to guide the tester when generating the test cases.

The *advanced security test cases* are test cases which are explicitly generated in order to kill all the ANR mutants. In fact, the advanced test cases are generated by calling each activity with all possible roles and contexts. This generation process leads to test cases that detect a rule that is added in an ANR mutant and that is not present in the original OrBAC policy.

For example, if the following rule is not defined:

```
permission(borrower,update,personnelAccount,workingDay))
```

The advanced security test cases will try to activate the method `updatePersonnelAccount` with a borrower role. If the method is executed normally then an ANR mutant is detected.

In this approach, we are using mutant score as the test criterion. We do not use mutation for analysis purpose (to compare test criteria or testing technique) but as a test selection technique. When the generation of mutants cannot be fully automated, this test selection technique is not applicable. Still, this study provides interesting results for test selection effort and test quality.

5.4. Issue 3: Advanced vs. basic security tests

The issue here is to compare the test cases selected to cover the security rules with CR2 (and which kill all mutants except ANR) and the advanced security tests which are generated in order to kill all the ANR

mutants. Table 3 presents the overlap between these two approaches. It is interesting to note that the advanced security test cases kill up to 60% of the basic security mutants. On the other hand, the test cases selected with CR2 only kill 17%. The effort to kill the ANR mutants is much more important (154) than for killing the basic security mutants (35).

In conclusion, the test selection based on the ANR mutants cannot replace the CR2 criterion. CR2 and advanced security test cases are not comparable, and are both recommended, the first to efficiently test the specification, the second to cover non-specified cases (robustness).

Table 3. Overlap of CR2 and adv. test cases

	#test cases	Basic security mutants	ANR
CR2	35	100%	17%
Adv. sec. tests	154	59.3%	100%

5.5. Issue 4: Incremental vs. independent test strategies

The issue now is to study whether we can leverage an incremental approach to save test generation effort. Table 4 recalls the number of test cases generated with the following strategies:

- the independent approach (we do not reuse functional test cases)
- reusing the functional test cases, completing them to reach the CR2 criterion and to kill all ANR mutants (*Incremental from functional strategy*).
- generating test cases to reach the CR2 criterion, completing them to kill all ANR mutants (*Incr. from CR2 strategy*).

The first incremental strategy seeks to take benefit from the existing functional test cases (which have to be adapted for security), the second one starts from the CR2 test criterion.

Even if quantitative results are displayed, the comparison is difficult because the effort to adapt functional test cases to security cannot be easily estimated in a general case. It depends on the system to be tested. It may be neglected: that's the case for our study since the security mechanisms are centralized and can be quite easily observed. In a general case, adapting these test cases may be as costly as generating security test cases. In Table 4, we put two values that correspond to the case when the cost of adaptation can be neglected in parenthesis. This issue is related to the problem of security mechanisms testability (controllability and observability).

It appears that the independent generation of basic security and advanced test cases is the most costly. With any incremental strategy, we need to generate 133 test cases to kill all advanced security mutants (saving 21 test cases generation). The final ranking between the two incremental strategies depends on the adaptation cost of functional test cases and may vary from a system under test to another. Only the test experts may estimate this adaptation cost. If it can be neglected, reusing functional test cases and completing them is the most interesting strategy. Another decision has to be taken which is to put an important effort for testing the SP robustness (killing ANR mutants).

Table 4. Independent vs. Incremental strategies

#Test cases / Strategy	Funct.	Basic Secur. CR2	Adv. Secur.	Total
Independent	-	35	154	189
Incremental from functional	(42)	21	133	154 (196)
Incr. from CR2	-	35	133	168

6. Related works

As it has been already mentioned, the most used models to express security at high level help to specify the application access rules and then implement them into the code. Other models exist, which are not dedicated to access control, such as [11] which help expressing security requirements and model the potential attacks to be prevented in the design stage. In this paper, we only consider the access control models, but the security aspects to be tested are not restricted to them and many other security aspects need to be covered by systematic testing.

In our approach we use mutation analysis in order to improve the security tests. Security tests become stronger because they are capable of detecting security faults. Recently mutation was applied to XACML based access control policies testing. Xie et al. [12] proposed a fault model for XACML policies. The mutation operators they use are different from those we propose because they are adapted to XACML style. Xie et al encountered the problem of detecting equivalent mutant due to the complexity XACML policies. We did not have this problem because, by construction, our operators do not produce equivalent mutants. In addition, Mathur et al. [13] applied mutation to RBAC models. They used formal techniques to conceive a fault model and adapt mutation to RBAC models.

In [14], Mathur et al. applied fault injection to the application environment. The application environment

is perturbed by modifying environment variables, files or process used by the application under test. Then the application has to resist to this perturbation and must not have an insecure behaviour that may lead to security flaw. In addition, fault injection was applied in another way. Adaptative vulnerability analysis [15] injects faults to the application data flow and internal variables. The objective is to identify parts of application's code that have insecure behaviours when the state of the application is perturbed.

7. Conclusion

This paper identifies a number of issues related to security and illustrates through several experiments what are the specificities of this testing that testing. In particular we illustrate how functional and security testing can be tackled as complementary activities. The paper proposes a methodology for test cases selection, with various test adequacy criteria based on the security policy (SP) or on mutation. It also distinguishes test selection for testing the "nominal" SP rules from the advanced test selection that aims at testing the robustness of the SP. The first test cases can be derived from CR2 test criterion. In this paper, we use ANR mutants as the target and criterion for testing SP robustness. The case study highlights the issues a test expert has to deal with when facing the objective of testing a SP for a real system. In particular, two qualitative aspects arise: the possibility, or not, to adapt functional test cases to test a SP, and the interest of advanced security tests, regarding the important additional effort it may require. More fundamentally, the aspect of security mechanism testability in relation to system architecture is critical. The way security mechanisms are spread over the system or centralized, the easiness or difficulty to relate a security rule to a piece of code are major issues to run the testing task.

8. References

1. D. F. Ferraiolo, et al., *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
2. S. I. Gavrila and J.F. Barkley. *Formal Specification for Role Based Access Control User/Role and Role/Role Relationship Management*. in *Third ACM Workshop on Role-Based Access Control*. 1996.
3. R. Sandhu, E.J.C., H. L. Feinstein, and C.E. Youman, *Role-based access control models.* IEEE Computer, 1996. **29**(2): p. 38-47.
4. A. Abou El Kalam, et al., *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
5. F. Cuppens, N. Cuppens-Boulahia, and M.B. Ghorbel. *High-level conflict management strategies in advanced access control models.* in *Workshop on Information and Computer Security (ICS'06)*. 2006.
6. DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
7. Basin, D., J. Doser, and T. Lodderstedt. *Model driven security: From UML models to access control infrastructures*. in *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 2006
8. Briand, L. and Y. Labiche, *A UML-based approach to System Testing.* Software and Systems Modeling, 2002. **1**(1): p. 10 - 42.
9. Nebut, C., et al., *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
10. Tejeddine Mouelhi, Yves Le Traon, and B. Baudry, *Mutation analysis for security tests qualification*, in *Mutation'07 workshop*. 2007.
11. Van Lamsweerden, A. *Elaborating Security Requirements by Construction of Intentional Anti-Models.* in *Proceedings of the 26th International Conference on Software Engineering*. 2004.
12. Martin, E. and T. Xie. *A Fault Model and Mutation Testing of Access Control Policies.* in *International Conference on World Wide Web*. 2007.
13. Ammar Masood, Arif Ghafoor, and A. Mathur, *Scalable and Effective Test Generation for Access Control Systems that Employ RBAC Policies*. 2006.
14. Du, W. and A.P. Mathur. *Testing for Software Vulnerability Using Environment Perturbation*. in *International Conference on Dependable Systems and Networks*. 2000.
15. Ghosh, A., T. O'Connor, and G. McGraw, *An automated approach for identifying potential vulnerabilities in software*, in *IEEE Symposium on Security and Privacy*. 1998.

Acknowledgments: This work is a part of the *Politess* project (ANR-05-RNRT-01301), granted by the French National Research Agency (ANR).