Mutation analysis for security tests qualification

Tejeddine Mouelhi,
Yves Le Traon
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex, France

Benoit Baudry
IRISA- 35042 Rennes Cedex,
France
bbaudry@irisa.fr

Abstract. *In this paper, we study how mutation analysis can be adapted to qualify test cases aiming at testing a security policy. The objective is to make test cases efficient to reveal erroneous implementations of a security policy. The notion of security policy testing is studied and mutation operators are defined in relation with the security rules. To make the approach applicable in practice we discus and empirically rank the security mutation operators from the most to the least difficult to kill. The empirical study is a library software, which is implemented with a typical 3-tiers architecture.*

1 Introduction

Testing that a system is correct with respect to security is known as a hard task [1]. Among the numerous identified difficulties we can first point that security issues are handled at many different places in a system (network, hardware, server and client). Moreover, since specifying the expected security qualities is complex, it is very difficult to express the expected result when building test cases for security.

In this work, we focus on a particular type of security that consists of assuring access control to sensitive data in a 3-tier application. From a testing point of view, we take benefit from the fact that access control models exist [2-4] and are used to specify security policies. Indeed, these languages allow expressing rules for the permissions or authorizations to access services and data. The set of rules is called a *security policy*. The particular language used in this paper is called OrBAC. Once specified with OrBAC [5], a security policy must be implemented in the business application (in Java in our example). Security test cases are generated to check the correctness of the implementation with respect to those security rules. .

Today there is no systematic way to derive test cases from a security policy and no test adequacy criteria to assess the quality of test cases for security. Thus, once test cases have been produced, it is necessary to estimate their quality in terms of ability to detect security *flaws* in the implementation.

In order to evaluate the quality of test cases for security, this paper adapts mutation analysis and introduces new mutation operators that correspond to fault models for access control security policies. To our knowledge, very few works have modelled faults related to security issues. In [6], Du et al. inject faults in the application's environment. This consists in modifying environment variables, files or processes used by the application under test. In [7] Ghosh injects faults in the application's data flow and internal variables to identify pieces of code which behavior is insecure. However, none of these faults targets the implementation of access control policies.

In section 2 we introduce a running example used for illustration through the paper. This example is based on a library management system. In section 3, we rapidly introduce the issue for security policy testing. Section 4 presents a set of mutation operators for access control test qualification. At last section 5 presents a case study that allows us to select a relevant subset of operators to qualify security test cases.

2 Example

This section introduces an example based on a library management system. Based on the simplified requirements of this system, it is possible to derive a set of access control rules that are modeled using OrBAC. We use these rules to illustrate the main feature of the OrBAC language.

2.1 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 in 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.

2.2 Modeling a security policy with OrBAC

From the previous requirements for the LMS, it is possible to 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): the organization, the role, the activity, the view and the 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 LMS requirements, we identify the entities displayed in Figure 1. The graphical representation is the one from MotOrBAC. First, we identify the services which are constrained by security rule. 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.

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 type of account: borrower and personnel accounts.

In this paper, the context corresponds to a temporal dimension that appears in access control rules for the LMS system. 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`.

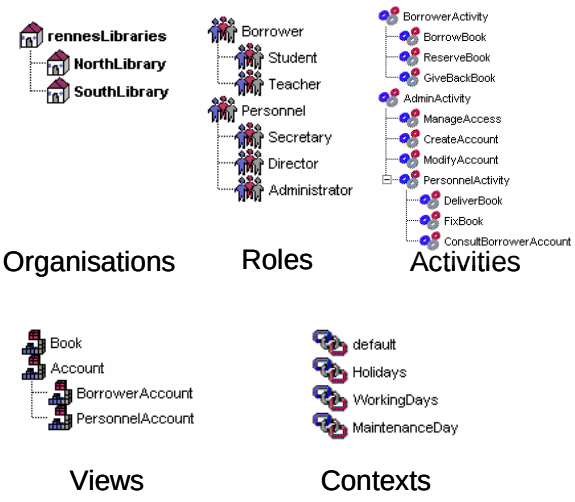

Figure 1 - OrBAC entities for the LMS

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`. The other parameters are entities that have been defined previously.

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)
```

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 [8] that automatically detects the conflicting rules in the definition of a security policy. One way to solve the conflicts consists in assigning priorities to rules. The rule with the highest priority is executed.

3 Testing a security policy

When the security policy has been defined from the requirements, conflicts can be checked with MotOrBAC. Then, these rules must be taken into account in the implementation of the business application. There is no automatic solution to generate code for a security harness on the business application. As described in Figure 2, the implementation is thus manual and consists in selecting the different places in the source code where security rule apply and add the necessary code.

Since the implementation of OrBAC rules in the application is manual, it is necessary to validate that it is correct. Specific test cases have to be designed to detect errors in the implementation of security errors that can entail security flaws. Testers can use the OrBAC specification to build those test cases. Once the test cases are defined, they have to be executed on the application. We have developed a harness to automatically run a set of test cases on the implementation and produce a verdict that reveals the presence or absence of a security flaw. When testing security rules, we look for prohibitions that can be violated and permissions that are not available.

There is no commonly agreed technique to derive test cases from security policy rules, and there exists no test adequacy criteria to assess the quality of the produced test cases. Thus, we propose to adapt mutation analysis to validate the test cases. As it is detailed in the next section, we define a number of mutation operators that model faults that can occur when implementing a security policy.

4 Security mutation operators

In this section, we discuss the issue of adapting mutation analysis to security, we propose security policy mutation operators and discuss their meaning.

4.1 Defining operators for OrBAC

Since there are many different solutions to implement access control rules into the business model, it is very difficult to define security fault models based on syntactic errors in the code. We could inject classical mutation faults (arithmetic, logical etc.) in security mechanisms, but the effect of these faults would be unpredictable w.r.t. a given security policy (and the verdict difficult to define).

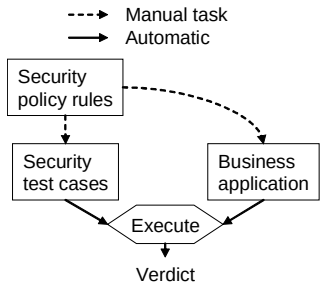

Figure 2 - Testing the security policy

In this section we define mutation operators independently from implementation-specific details. Since the access control rules are expressed with OrBAC, we model the faults at this level of abstraction. The mutation operators are thus expressed on OrBAC syntax. On one hand, 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.). On the other hand, the difficulty consists in transforming mutation operators defined with OrBAC into faults in the implementation.

For this work, we maintain traceability matrices between OrBAC rules and the corresponding code

blocks in the implementation. Using this information, it is easier to map mutant OrBAC rules to faults in the implementation. However, in the general case, the traceability information might not be available, or the mapping between the rules and the code might not be trivial. In that case it is not possible to automatically generate the mutants. If we still want the mutation analysis to be used, it is important to identify the most relevant subset of operators, in order to produce the minimum mutants that lead to the production of the best security test cases. This is the objective of the case study to select a sufficient subset of mutation operators.

In the following we present a set of mutation operators that can be defined for OrBAC rules. We illustrate these operators with examples. Then, in next section we run a case study to select the most relevant subset. The security mutation operators are divided in 4 categories:

- Type changing operators
- Parameter changing operators
- Hierarchy changing operators
- Adding rules operators

4.2 Type changing operators

Type change operators pick a rule and modify its type. There are 3 types of rules: prohibition, permission and obligations. Therefore, there are 6 possible modifications:

1. Prohibition to permission
2. Permission to prohibition
3. Prohibition to obligation
4. Obligation to prohibition.
5. Permission to obligation
6. Obligation to permission

The first four operators simulate faults that occur during the implementation. The obtained policy allows a forbidden activity or prohibits an authorized activity. Let us focus on some examples obtained by the first 2 operators:

Rule used:

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

Rule to use instead:

```
Prohibition (Library, Secretary,

ConsultBorrowerAccount, BorrowerAccount,
default)
```

Rule used:

```
Prohibition(Library,Student,GiveBackBook,
Book, Holidays)
```

Rule to use instead:

```
Permission(rennesLibraries,Student,GiveBackBoo
k, Book, Holidays)
```

Operators 5 and 6 switch between permission and obligation. The problem is that it is difficult to distinguish between obligation and permission. Test cases are not able to validate that an obligation rule is used instead of a permission one. For this reasons, we choose not to use obligation rules and only use prohibition rules. We only implement the first 2 operators (Prohibition to permission named PRP and permission to prohibition named PPR).

4.3 Parameter changing operators

Parameter changing operators pick a rule and change one of its parameters. As there are 5 parameters the following operators are defined:

1. Change organization
2. Change role
3. Change activity
4. Change view
5. Change context

We can not replace a rule's organisation because each organisation defines its own entities and we may not have the role, activity, view and context of the first organisation defined for the new organisation. If we have:

```
Permission(org1,role1,activity1,view1,context1
)
```

We cannot replace it with:

```
Permission(org2,role1,activity1,view1,context1
)
```

We can only replace if org2 defines role1, activity1, view1 and context1 in its scope.

In addition, the notions of activity and view are tightly linked. In fact, the activity must be related to the rule view. The following example illustrates this issue:

Rule used:

```
Permission (Library, Administrator,
ModifyAccount, BorrowerAccount, default)
```

Rule to use instead:

```
Permission (Library, Administrator,
ReserveBook, BorrowerAccount, default)
```

`ModifyAccount` cannot be replaced by `ReserveBook` because `ReserveBook` can not be attached to the `BorrowerAccount` view. The same problem appears when we change the rule's view. In fact, replacing views or activities can be done only for activities that are independent from views (and can be applied to different views). For example, if we have the activity `Modify` that may be applied to 2 views `BorrowerAccount` and `PersonnelAccount` then we can replace `BorrowerAccount` by `PersonnelAccount`. Therefore, the relevant operators are those changing

the role and the context. We show 2 examples of these operators' results:

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

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

Rule used:
```
Permission (Library, Student, BorrowBook,
BorrowerAccount, WorkingDays)
```

Rule to use instead:
```
Permission (Library, Student, BorrowBook,
BorrowerAccount, Holidays)
```

These 2 operators are useful because they will simulate cases where the security policy is too permissive or too restrictive. The first one allows a user (the secretary) to do the same activity allowed for another user. The second one allows a user to perform the action under another context. In this category, we only keep two operators (Change role named RRD and change context named CRD).

4.4 Hierarchy changing operators

Or-BAC allows defining hierarchies for organisations, roles, activities, views and contexts. Then, a mutation operator can be used to change hierarchies by replacing a parameter by its parent or one of its descendants.

1. Change organization hierarchies
2. Change role hierarchies
3. Change activity hierarchies
4. Change view hierarchies
5. Change context hierarchies

Due to the reasons explained in the previous section, the first operator is not used. In addition, context and views hierarchies are not useful. In practice we do not define hierarchies for contexts and views. In fact, we insist on hierarchies for activities and roles.

The only useful operators in this category are operators 2 and 3.
Examples:
Change role hierarchies, rule used:
```
Permission (Library,Borrower,reserveBook,
Book,WorkingDays)
```

Rule to use instead:
```
Permission (Library, Teacher,reserveBook,
Book,WorkingDays)
```

Change activity hierarchies, rule used:
```
Permission (Library, Student,BorrowerActivity,
Book,WorkingDays)
```

Rule to use instead:
```
Permission (Library, Student,BorrowBook,
Book,WorkingDays.
```

The hierarchy changing operators that will be retained are: 'change rule hierarchies' (named RPD) and 'change activity hierarchies' (named APD).

4.5 Add rule operators

Instead of replacing an existing rule, the adding rule operator introduces a new rule. The goal of this operator is to simulate cases where the implementation does something in addition to the requirement. This is a typical security fault and makes security faults different from functional tests. The security breaches are caused by the fact that the application behaves in unexpected way, even if it satisfies all functional requirements.

In order to obtain relevant rules, the add rule operator (named ANR) introduces rules that contain an activity and a view that were already defined by at least one rule in the initial security policy. It is important to note that this operator generates a lot of mutants.
Examples:
Added rule:
```
Permission
(Library,Borrower,consultPersonnelAccount,
PersonnelAccount,MaintenanceDay)
```

Added rule:
```
Permission (Library,Secretary,ManageAccess ,
PersonnelAccount,MaintenanceDay)
```

4.6 Security mutation operators

Table 1 presents the retained operators that were presented in details in previous section.

Table 1 – Mutation operators for OrBAC rules

Mutation Operator	Description
PRP	Prohibition rule replaced with permission
PPR	Permission rule replaced with prohibition
RRD	Rule role is replaced with different role
CRD	Rule context is replaced with different context
RPD	Rule role (a parent) replaced with one of its descendants
APD	Rule activity replaced with one of its descendants
ANR	New rule Added

Number of mutants

The following table shows the number of each type of mutant that can be generated. It is function of the number of entities that are in the model.

Table 2 - Number of generated mutants

Operator name	# generated mutants
Hierarchy changing operator	:# hierarchies levels * #number rules with entities (contained in hierarchies)
Addition of new rule operator	#roles * #contexts * # (activities and views)
Type changing operator	# permission, #prohibition
Parameter changing operator	# entities * # rules

5 Implementation and case study results

5.1 Objectives and experimental protocol

The goal of this study is to analyse which mutation operators are the most useful to check the efficiency a security policy test suite. As for the principles of selective mutation [9], we would like to determine a subset of sufficient mutation operators. The issue is to determine which mutants are easier to kill and thus keep only the hard-to-kill mutants. This study is even more critical to make the approach feasible in a general case, when the generation of mutants can not be fully automated from the security policy rules.

This study has been designed for this objective, as an "ideal" lab case, in order to obtain general results. The generation of mutant security mechanisms is fully automated, allowing the presented experimental protocol to be applied. The *first step* consists of generating minimal test suites per mutation operator, w.r.t. the following definition:

Definition: minimal test suite. A test suite is minimal for a set of mutants iff the test cases it includes have a 100% mutation score and, if when a test case is removed, the mutation score decreases.

We note TS(name of operator) the minimal test suite needed to kill all the mutants generated with this operator. In this study, an important effort has been allocated for generating the test cases and minimizing the test suites. Since it is difficult to have much less than a test case by security rule, we believe the minimal test suites are close from the optimum. For instance, the minimal test suite of 36 test cases selected for killing the basic mutation operators is equal to the number of non generic security policy rules.

The *second step* consists of comparing the mutation operators. This comparison leads to a ranking, which is obtained with two criteria. The first one determines whether a mutation operator can replace another. This aspect is captured by the notion of *subsume relationship*. When two mutation operators are equivalent, any of them can replace the other without loss of efficiency for the generated test cases. To choose which of the two mutation operators can be removed, we consider the number of generated mutants as a second criterion.

Definition: subsume relationship (->). A mutation operator MO1 strictly subsumes MO2 (MO1 -> MO2) if:

a) the minimal test suite TS(MO1) also reaches a 100% mutation score for the MO2 mutants

b) the minimal test suite TS(MO2) does not reach 100% for MO1 mutants.

MO1 and MO2 are equivalent (MO1 <-> MO2) if TS(MO1) reaches 100% on MO2 mutants and TS(MO2) reaches 100% on MO1 mutants.

Figure 3 illustrates the definition. MO1 -> MO2 since the test suite for MO1 is sufficient to kill all mutants generated with MO2. Conversely, MO2 doesn't subsume MO1 since its minimal test suite only kills 80% of the mutants created with MO1. We can thus consider that MO2 can be removed, since it is not needed when qualifying a test suite. MO1 precedes MO2 in the ranking. In case of equivalence, one of the two mutation operators is useless. To determine which one can be removed, we consider that it is better to generate less mutants (due to the execution times, and to the effort needed for generating these mutants when done manually). So, the ranking relation is defined as follows.

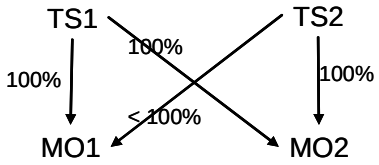

Figure 3 - MO1 operator strictly subsumes MO2

Definition: mutation operators ranking (>).

MO1 > MO2 iff:

MO1 -> MO2

or **(** (MO1<->MO2)

and |{MO1 mutants}| < |{MO2 mutants}| **)**

This ranking only orders partially mutation operators. If a mutation operator is not ranked, it is independent and necessary for a relevant test qualification process.

5.2 Mutation tool

a Generation of mutants

The mutant generator is implemented as part of the MotOrBAC tool that implements the OrBAC security model. The goal of this tool is to allow security administrators to specify and define an OrBAC based security policy model.

We added a module that generates security policy mutants. When the security policy is defined the tool creates the security mutants. Its user interface is shown in Figure 5.

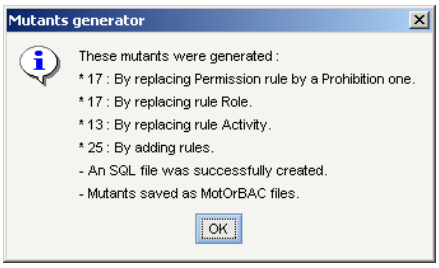

Figure 4 Mutation result dialog box

The user can choose the number of generated mutants for each category. Then the tool creates MotOrBac security policy files and/or an SQL script that creates a database table containing all the mutant rules (each mutant policy having a unique id).

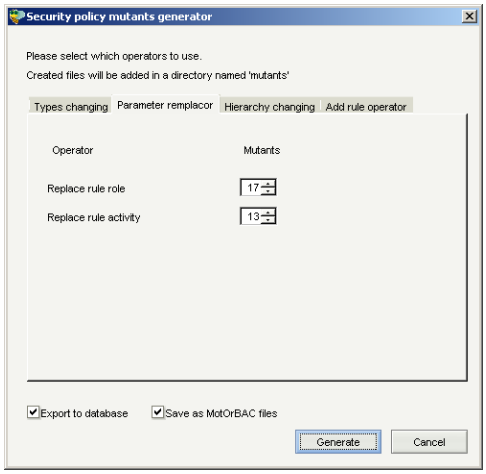

Figure 5 - The mutant generator window

The mutation tool uses MotOrBac libraries to get the current security policy rules list, entities as well as hierarchies. In addition, the mutation tool uses MotOrBac modules to check if conflicts exist and resolves them. In case of conflict, it means that the mutated rule is in conflict with another. The conflict is automatically solved by giving the highest priority to the mutated rule. The result is that this rule is implemented or executed in priority. The numbers of generated mutants are shown in a dialog box (as presented in Figure 4). For our experiments, we generated all the possible mutants, for ranking the mutation operators.

b Oracle function for mutation analysis

In order to decide whether a mutant is killed or not, we use an oracle function that checks the difference between the output of the mutant policy implementation and the correct security policy one. The security mechanism prints the authorization that was granted to the requested action into a file. The oracle function thus consists in comparing the files that are produced by the mutant and the original policy. We present here an example of different outputs:

Output of the application using initial security policy:

```
INFO main root - Permission granted for the
requested action reserveBook BORROWER
```

Output of the application using mutant security policy:

```
WARN main root - Requested action prohibited
reserveBook BORROWER
```

Information about the operator used to generate this mutant:

```
Operator used: Type Changing Operator

Rule to change:
Permission (rennesLibraries,Borrower,
ReserveBook, Book, WorkingDays).

Rule to use instead:
Prohibitionp (rennesLibraries, Borrower,
ReserveBook, Book, WorkingDays).
```

This oracle function, by comparing the behaviour of the initial program with the seeded one, is sufficient to determine that a test case kills a mutant. For a practical use, this comparison is not possible and the tester must define an explicit oracle function or manually establish the verdict. This is another reason why it is important to generate only the necessary and sufficient number of mutants.

5.3 Case study architecture

The case study application has a typical 3-tiers architecture widely used for web application. Figure 6 presents the main characteristics of this architecture.

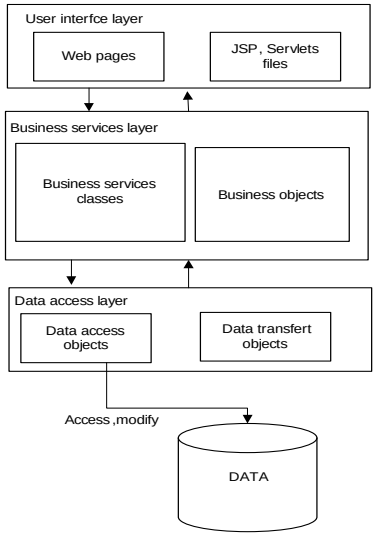

Figure 6 - The 3-tiers architecture for the library system

The security mechanism implementing the Or-BAC based security policy is located in the business layer, in the service methods. Before performing the requested action, the method calls the security mechanism that checks if the security policy allows of forbids that action. When the action is prohibited a security policy 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 leaves 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.

When we implemented the security policy, 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 mutation analysis runs as follows:

- Tests are executed against the initial security policy.
- The output is saved in a file
- The policy is replaced with a mutant policy and the logger output is saved in a different file
- The 2 files are compared with the oracle function defined in the previous section. If behaviours are different then the mutant is killed otherwise it is still alive and the test did not succeed to find the security fault.

5.4 Results and analysis

Operator category	Op. name	# mutants
Rule type changing operator	PPR	22
	PRP	19
Rule parameter changing operator	RRD	60
	CRD	60
Hierarchy changing operator	RPD	5
	APD	5
Rule adding operator	ANR	200
All		371

Table 3 - Number of generated mutants per operator

Table 3 gives the number of generated mutants per operator. The ANR operator generates much more mutants since it adds a non specified security rule. The number of generated mutants thus reflects the fact that all the possible cases have not been specified in the security policy. To our own experience, this is quite usual: the specification focuses on the most critical and important case, and often considers that a default behaviour is acceptable. Testing these cases allow the default policy to be exercised and allow to highlight lack in the specification. On the other hand, they are few mutants generated from hierarchy changing operators because the specification doesn't introduce many hierarchical entities.

Ranking mutation operators

First, the mutation analysis was applied with each minimal test suite for each mutation operator. The results were deceiving since no clear ranking appear. We then consider minimal test suites for couples of mutation operators: PPR-PRP (Rule type changing operators), RRD-CRD (Rule parameter changing operator) and RPD-APD (Hierarchy changing operator). The results for the size of the minimal test suites are displayed in Table 4. The relationships between minimal test suites are the following:

$TS(RRD\text{-}CRD) = TS(PPR\text{-}PRP)$

$TS(RPD\text{-}APD) \subset TS(PPR\text{-}PRP)$

$|\ TS(PPR\text{-}PRP) \cap TS(ANR)\ | = 21$ test cases

So, both PPR-PRP and RRD-CRD operators subsume the RPD-APD operator. PPR-PRP and RRD-CRD are equivalent for the subsume relationship. Taking into account the second criterion, the number of generated mutants per operator, we obtain the following ranking:

PPR-PRP -> RRD-CRD -> RPD-APD

This result shows that the PPR-PRP operator should be used in priority, thus avoiding the creation of most mutants.

Operator category	OP	Size of minimal test suites
Rule type changing operator	PPR	36
	PRP	
Rule parameter changing operator	RRD	36
	CRD	
Hierarchy changing operators	RPD	4
	APD	
Rule adding operator	ANR	154

Table 4 – Number of minimal test suites

The ANR and PPR-PRP operators are not comparable with this ranking. Some test cases are shared by both minimal test suites (21 test). Table 5 shows the overlap between the test suites, in terms of respective mutation scores. The minimal test suite for ANR kills 59.3 % of the non-ANR mutants. On the other hand, the PPR-PRP test suite only covers 17% of the ANR mutants. The ANR mutants are thus necessary but cannot replace the PPR-PRP operator. PPR-PRP and ANR are not comparable, and are recommended operators. On this case study, the ANR operator is the most costly in terms of generated mutants. This number may vary, depending on the completeness of access control rules. The fewer rules are specified, the more mutants this operator generates. We believe that it is likely that the rules do not specify all the combinations explicitly, and that this operator is the most costly. On the other hand, the PPR-PRP operator 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 PPR-PRP and ANR.

	all mutants except ANR	ANR
TS(PPR-PRP)	100%	17%
TS(ANR)	59.3%	100%

Table 5 Overlap of test suites between PPR-PRP and ANR operators

Removing mutants generating conflicts

To reduce the number of mutants to be generated, we remarked that the mutants which caused a conflict (which is solved by giving priority to the mutant rule) are the more easy to kill. The Table 6 presents the number of mutants which generated conflicts per operator. Figure 7 shows an example of such a conflict. It illustrates that two test cases, generated to kill mutants without conflicts, kill this mutant with conflict.

Operator category	With conflicts	Without conflicts
Rule type changing operator	-	41
Rule parameter changing operator	23	97
Hierarchy changing operators	-	10
Rule adding operator	33	167
Total	56	315

Table 6 - # of generated mutants with and without conflicts

So, several test cases, from the minimal test suites, systematically kill these mutants. It means that these mutants are not necessary. Indeed, we have:

TS(RRD-CRD with conflicts) $\subset$ TS(RRD-CRD without conflicts)

TS(RRD-CRD without conflicts) corresponds to the suite of 36 test cases needed both for Rule type and Rule parameter changing operators. This test suite is also minimal for the mutants RRD-CRD which do not cause conflicts. Among this set, only 12 test cases are needed to kill mutants with conflicts which compose the TS(RRD-CRD with conflicts) minimal test suite.

TS(ANR with conflicts) $\not\subset$ TS(ANR without conflicts)

But TS(ANR with conflicts) $\subset$ TS(PPR-PRP) and TS(ANR with conflicts) corresponds to a minimal test suite of 21 test cases (which are the exact intersection of the TS(ANR) and TS(PPR-PRP) test suites.

It means that if the PPR-PRP and ANR operators are used, the minimal test suite of PPR-PRP kills the mutants with conflict generated with ANR. Combining these two operators allows removing the mutants with conflicts generated with the ANR operator. Around 18% of the mutants (those with conflicts) can be removed without any loss of relevance for the generated test cases.

We study why these mutants could be removed and the explanation is interesting, since it corresponds, for security testing, to the 'coupling effect' analyzed by Offutt et al. [10]. In fact, when a mutated rule causes a conflict, it has an impact on at least two rules: this fault is equivalent to two sequential mutations without conflicts.

Operator used: RRD

Initial rule:
Prohibition(Library,Secretary,ManageAccess, PersonnelAccount,Default)

Mutated rule:
Prohibition(Library,Administrator,ManageAccess, PersonnelAccount, Default)

The seeded rule is in conflict with another rule:
Permission(Library,Administrator,ManageAccess, PersonnelAccount,Default)

The two tests that kill this security policy mutant:

Test 1: Test that secretary cannot manage access
Test 2: Test that admin can manage access.

These two tests are already generated to kill mutants without conflicts.

Figure 7 – A mutant causing conflicts generates at least two elementary faults which are killed by two test cases.

Analysis and conclusion

In conclusion, some operators are more relevant than others for improving the quality of security test cases. We present a ranking of the most useful mutation operators:

1. Adding rules operators
1. Rule type changing operators
2. Rule parameter changing operators
3. Hierarchy changing operators

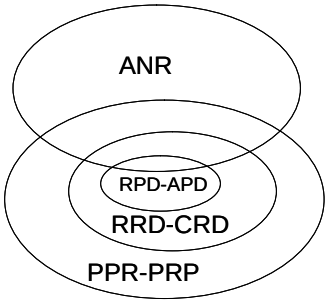

Figure 8. Relation between operators

Figure 8 displays this ranking, and highlight the overlap between ANR and the other operators. All these operators generate mutants which intersect with the adding rule operator. Adding rules operators are the most interesting because they simulate cases that are not tested by functional tests. As shown by the results, they are the most difficult to kill, regarding the number of test cases needed to detect its generated mutants. Only advanced security test cases are able to kill mutants without conflicts created by this operator.

6 Conclusion

This paper proposes mutation operators for security policy testing. To qualify a set of security policy test cases, a classical mutation analysis is applied with these 'security' mutants. In practice, it may be costly to implement all types of security mutants, especially if it must be done manually. We thus performed detailed experiments, as rigorously as possible, on a 3-tiers architecture example (a library system) to select a sufficient subset. The hardest-to-kill security mutant operators are those which must be generated in priority. If this initial study has to be completed with others, the first results reveal that we can limit the types and number of generated mutants. The ranking shows that two operator types are necessary: rule type changing operator (PPR and PRP) and rule adding operator (ANR). Combining these two operators also leads to another minimization of the generated mutants: only mutants which do not cause conflicts in the security rule have to be created. It is due to the fact they introduce at least two coupled elementary faults (coupling effect). While PPR-PRP mutants force the test to cover the specified security rules, the ANR ones are useful to check the robustness of the system in case of default or underspecified policies. Combining both operators provide a good criterion to guide the tester when generating the test cases.

7 References

1. Thompson, H.H., *Why security testing is hard.* IEEE Security & Privacy Magazine, 2003. **1**(4): p. 83 - 86.
2. B. D. Joshi, J., et al., *Security models for web-based applications.* Communications of the ACM, 2001. **44**(2): p. 38 - 44.
3. 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.
4. 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.
5. Abou El Kalam, A., et al. *Organization Based Access Control*. in *International Workshop on Policies for Distributed Systems and Networks* 2003. Lake Come, Italy.
6. Du, W. and A.P. Mathur. *Testing for Software Vulnerability Using Environment Perturbation*. in *International Conference on Dependable Systems and Networks*. 2000.

7. 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.
8. Cuppens, F. *MotOrBAC*. 2007 [cited 2007; Available from: http://motorbac.sourceforge.net/index.php?page=home&lang=en.
9. Offutt, A.J., et al., *An Experimental Determination of Sufficient Mutant Operators*. ACM Transactions on Software Engineering and Methodology, 1996. **5**(2): p. 99 - 118.
10. Offutt, A.J., *Investigations of the software testing coupling effect*. ACM Transactions on Software Engineering and Methodology, 1992. **1**(1): p. 5 - 20.