Testing and Modeling Security Mechanisms in Web Applications

Tejeddine Mouelhi

N° d'ordre : 2010telb0151

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Rennes 1

Ecole Doctorale – MATISSE

Testing and Modeling Security Mechanisms in Web Applications

Thèse de Doctorat

Mention : Informatique

Présentée par **Tejeddine Mouelhi**

Département : RSM

Directeur de thèse : Yves Le Traon

Acknowledgments

To begin, in order not to forget anyone, let me thank everyone on this earth. Thanks to the reviewers, the jury, my advisors, my colleagues, my family, my friends, my fellow citizens, the French people, the African people, the European people, citizens from all over the world. Thanks to everyone for their help. Thanks to god for helping me.

I would like to thank the reviewers for their insightful comments and interesting remarking. I sincerely thank professor Parissis and professor Bertolino for reading my dissertation and for their helpful and constructive remarks.
I would like to thank the Jury. Thanks to everyone for the interesting questions and the thoughtful discussions.
I am very grateful and thankful to my advisors Yves and Benoit. Thanks to Benoit for his help during my internship and my thesis. I will always remember the insightful discussions that I had with him, from which I learned a lot from.
Infinite thanks go to Yves, for his help during my thesis, he was always there for me, to guide me and to answer my questions. I thank him for his help, his support and for the hours and hours of discussions we had together and from which I learned a lot. I hope that I will continue working with him and doing good and innovative research.
I would like to thank my friends and colleagues in Triskell and in the Serval research teams. I thank those who helped me by reading and correcting this dissertation and I thank those with whom I worked and did research, specially, Ingrid, Thomas, Sylvain, Franck, Brice and Olivier.
My final thoughts go to my family. Thanks to my brothers, Nizar, Issam and Mohamed. They supported me during my studies and where always there for me.
Unlimited gratitude to my parents, this page is not enough to show them how much I am thankful and grateful. Many thanks to my father Youssef, who believed in me, when he told me one day, years ago, that I could become a doctor within years, a discussion that I will remember for the rest of my life. Thanks to my mom; Mabrouka, she never stopped encouraging me and supporting me, endless thanks to you mom.
My last thought goes to my beloved fiancée and soon to be my wife, Teicir. She believed in me too and supported me especially during the last months of my thesis writing. I thank you very much.

Contents

Acknowledgments **3**
1 Introduction **9**
2 Background and state of the art **13**
2.1 An overview on computer security 13
2.2 Application level security 14
2.2.1 The security threats 14
2.2.2 Security mechanisms 16
2.3 Modeling security 22
2.4 Testing and Security Testing 26
2.4.1 Software testing 26
2.4.2 Security testing 29
2.5 Conclusion 31
3 Security testing vs. Functional testing: Going beyond functional testing **33**
3.1 Introduction 33
3.2 Process and definitions 34
3.2.1 Definitions 35
3.2.2 Library management system 36
3.2.3 Modeling an access control policy 37
3.3 Access control test cases selection 39
3.3.1 Access control test cases 39
3.3.2 Test criteria 39
3.3.3 Test strategies 40
3.3.4 Test Qualification 40
3.4 Mutation analysis applied to Access control testing 41
3.4.1 Defining operators for access control policies 41
3.4.2 Type changing operators 42
3.4.3 Parameter changing operators 42
3.4.4 Hierarchy-following parameter changing operators 43
3.4.5 Add rule operators 44
3.4.6 The effectiveness of mutation analysis 44
3.5 Empirical studies and results 45
3.5.1 Generated mutants 45
3.5.2 Issue 1: functional tests for validating the security policy 46
3.5.3 Issue 2: Comparing security test criteria 47
3.5.4 Issue 3: Robustness vs. basic security tests 47
3.5.5 Issue 4: Incremental vs. independent test strategies 48
3.5.6 Issue 4: ranking mutation operators 49
3.6 Conclusion 54
4 Testing strategies for access control policies in applications **55**
4.1 The bottom-up approach: testing security with functional tests 55
4.1.1 Overview of the approach 55
4.1.2 Selecting test cases that trigger security rules 57
4.1.3 Relating test cases and security rules 57
4.1.4 Oracle modification in the functional test case 59
4.1.5 Proof of concept and empirical study 60
4.1.6 Comparing several degrees of automation 61
4.1.7 Summary and conclusion 62

4.2 Top-Down approach: Model-Based testing of access control policies 63
4.2.1 The considered access control model 63
4.2.2 The combinatorial testing 63
4.2.3 Adapting Pair-Wise test methodology 63
4.2.4 Concrete Test Cases 66
4.2.5 Implementation 67
4.2.6 Experiments 67
4.2.7 Summary and conclusion 71
4.3 Detecting hidden security mechanisms 73
4.3.1 Background 74
4.3.2 Flexibility analysis of the system to assess control policy changes 76
4.3.3 Test Selection for assessing system flexibility 78
4.3.4 Experiments and discussion 81
4.3.5 Summary 83
4.4 Conclusion 84
5 Automatic analysis of web applications for shielding and bypass testing 85
5.1 Introduction 85
5.2 Context and motivations 86
5.2.1 Definitions 86
5.2.2 The input validation architecture 87
5.2.3 Client-side validation techniques: 87
5.3 Overview of the approach 88
5.4 Client-side analysis for constraints identification 89
5.4.1 The automated crawler 90
5.4.2 Manual navigation and the use of functional tests 90
5.4.3 Collecting HTML constraints 90
5.4.4 Interpreting JavaScript 91
5.5 Server-side shield: a shield tool for protecting against bypass attacks 92
5.5.1 The contracts manager 92
5.5.2 The bypass-shield 93
5.5.3 Impact of constraints enforcement on security 94
5.6 Automated bypass testing 95
5.6.1 The generation of malicious test data 95
5.6.2 Construction and execution of bypass tests 96
5.6.3 Filtering and classifying the results 96
5.7 Experiments and results 97
5.7.1 The five case studies and the bypass some examples of testing results 97
5.7.2 Performance results 98
5.8 Summary 98
6 An MDE process for specifying, deploying and testing access control policies 100
6.1 Introduction 100
6.2 MDE for Security 101
6.2.1 Overview of the modeling architecture 102
6.2.2 MDE security design process 102
6.2.3 Validation of access control mechanism 103
6.3 The access control metamodel 104
6.3.1 Generic security metamodel 104
6.3.2 Instantiating the metamodel 105
6.4 Description of the security mechanisms 108
6.4.1 The architecture of the security mechanism 108

6.4.2 PDP XACML code generation from the metamodel 110
6.5 Validation of the access control mechanisms 111
6.5.1 The verification checks 111
6.5.2 Mutation analysis applied to access control testing 112
6.5.3 Generated mutants 114
6.6 Case studies and results 116
6.6.1 Validation results 116
6.7 Discussion about language-independent V&V vs. language specific ones 118
6.7.1 Limitations of the verification 119
6.7.2 Language-independent vs. language specific qualification 119
6.8 Conclusion 120
7 Security-Driven Model-Based Dynamic Adaptation 121
7.1 Introduction 121
7.2 Overview 122
7.3 Composing Security and Architecture Metamodels 123
7.3.1 A Generic Metamodel for Describing Access Control Policies 123
7.3.2 A Core Metamodel for (Runtime) Architectures 124
7.3.3 Mapping Security Concepts to Architectural Concepts 125
7.4 Security-Driven Dynamic Adaptation 126
7.4.1 Synchronizing the Architecture Model with the Running System 126
7.4.2 Activation/deactivation of security rules 128
7.4.3 Evolution of the Access Control Policy 129
7.4.4 Discussion 130
7.5 Conclusions and Future Work 131
8 Conclusions and perspectives 132
8.1 Summary 132
8.2 New Research directions 134
8.2.1 Mutation analysis for Usage Control 134
8.2.2 Extending the bypass-shield tool 134
8.2.3 Model-Driven approaches 134
References 135
List of figures 143
List of tables 145

Introduction

Security is growing in importance and currently becoming a key issue for organizations, companies and governments. The market is moving into the Internet and the amount of assets involved is huge. For companies selling their products and services over the Internet, security is a key problem to tackle. For on-line banks, guaranteeing security is crucial and a pre-requisite before even running the service. For governments, security is more a strategic concern that involves protecting the main infrastructures against cyber-attacks for terrorists groups or the rogue states.

Although it is obvious that security is so important for many actors, the status reports on security flaws and successful attacks show clearly that the effort is far from being satisfactory. Websense reports [1] that during the first half of 2009, the number of malicious Website have grown by 671% compared to 2008. 61% of the top 100 websites host malicious content or links to malicious websites. Moreover, according to PandaLabs [2] 25 millions of malware were created during 2009. The number of attacks is growing too. Symantec claims that it had stopped an average of 100 attacks per second in 2009 [3]. Furthermore, the consequence of successful attacks is important. Cybercrime had cost approximately $1 trillion in intellectual property losses and others cost, for repairing the damages, as estimated by McAfee in January 2009 [4].

The emergence of social network website like Facebook and Twitter bring new issues related to privacy and security. These websites are attracting millions of users who share private data that is made publicly available due to security flaws in these platforms. Access control is important to get right for these web applications. Flaws in the access control mechanisms would lead to exposing users private data.

There is indeed a growing need for efficient and powerful security mechanisms to protect against attacks and guarantee privacy such as access control. These mechanisms should be thoroughly validated and assessed. Any problem in implementing or deploying these mechanisms would lead to critical vulnerabilities that could be exploited by attackers to harm the system.

On the other hand, considering industry practices, it is interesting to notice that the industry takes into account only two main ways to perform security testing: automated tools, and manual penetration testing. Security testing denotes the activity of validating security components and checking the security related behavior of the application to locate eventual flaws. The two main approaches considered by the industry are not sufficient and have their limitations.

On one hand, automated tools are not efficient for detecting subtle flaws (as shown by an interesting study about the limitations of automated tools used to detect buffer overflow vulnerabilities that were reported by security experts [5]).

On the other hand, penetration testing effectiveness depends mainly on the expertise of the security tester, and when done manually it cannot be covering all parts of the code (consider large web applications), and will usually reveal a small piece of the existing vulnerabilities. This may be good enough for vendors; to claim that they were able to find many flaws, but does not obviously provide any guarantee or criteria to trust the deployed software.

These approaches are doing *a posteriori* security validation and do not exploit the software engineering techniques to design and test a software in an *a priori*, constructive way, for deploying a secure application. In other words, before doing penetration testing, the software designer and tester should be able to specify its security policy and test that the expected security mechanisms are correctly implemented. A good trade-off between the two

approaches would be to use software modeling and applying existing testing techniques on security. As several other researchers, we promote “software engineering for security” as the right way to deal with security.
The main intent of this thesis work is to provide new approaches for security testing and for building efficient and secure systems.
This thesis work tackles the issues of security testing and modeling security mechanisms. The first part focuses on **both testing the internal security mechanisms**, namely the access control mechanisms, and the external interface of web applications by performing **bypass testing** and leveraging it to provide a protection mechanism. The second part provides model-driven approaches for developing and integrating access control mechanisms.
The architecture considered in this thesis is depicted in Figure 1. The idea is to both protect the interface and the internal part of the web application.
The internal parts of the application are protected using access control mechanisms that aim at regulating the user access to the system resources and functions according to their rights, privilege and to the application or transaction context. The access control mechanisms can be exploited by attackers is case of poor implementations and the existence of errors.

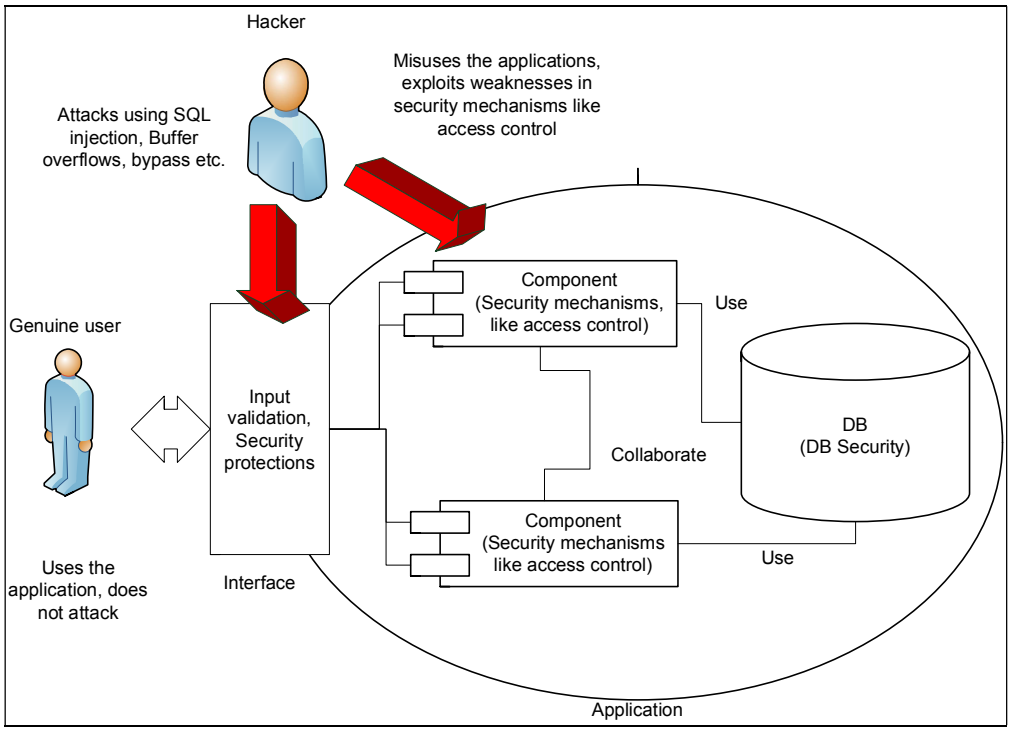


Figure 1 - Overview of the considered architecture

For web applications, the interface protection involves client side input validation. The input validation mechanisms could be bypassed by rogue users by performing bypass attacks. These attacks involve bypassing the client-side input validation and sending erroneous data or attacks to the server-side. When server-side validation is not correctly implemented the attack succeeds.
It is therefore important to thoroughly test and validate both the internal and external part of the application. The first part of this thesis work focuses on this issue.
The remainder of this dissertation is organized as follows. Chapter 2 helps setting the scene and putting into context this research work and shows the existing related work. The following chapters present the thesis work that is divided into to parts. The first part concerns security testing of newly developed or legacy systems, while the second part focuses on

providing new methodologies and processes for building secure systems and automatically integrating access control mechanisms.

The first part tackles the issue of testing access control. By its intrusive nature, the access control policy mechanisms are frequently triggered by simply running the system main functions. Therefore, it appears to be possible to try and use functional test to validate access control mechanisms, since these tests trigger all the system functions. The first research issue to be studied in Chapter 3 is to evaluate the quality of functional test cases when used to test access control mechanisms by comparing them to specific security test cases build according to adapted security criteria. In order to do this, *mutation analysis* will be used as an evaluation criteria. To that end, mutation analysis will be adapted to the context of access control testing. Another research question is about comparing two test generation strategies; an independent test strategy where only specific security test cases are used and on incremental one that reuses the existing functional test cases and complete them to test access control mechanisms. Based on the empirical results, several conclusions are drawn. They show that functional tests are not sufficient to test access control mechanisms and that the incremental strategy is better that independent strategy.

Chapter 4 follows these findings and provides three complementary testing strategies for access control testing. Two of these strategies are related to test generation: a top-down approach based on pair-wise technique for generating security test cases, in addition to a bottom-up approach that selects and transforms functional test cases into security ones. These two testing strategies help validating existing or newly built systems. In order to make the access control policy evolve, it is important to be able to detect the hidden security mechanisms. A third testing process is proposed to detect then hidden security mechanisms, which represents an important problem that is a direct consequence of the nature of access control architecture. In legacy systems, access control mechanisms are sometime hard-coded in the application business logic. This harms the flexibility of the system and limits the possibility of making the access control policy evolve. To solve this important issue, a testing process based on mutation analysis in proposed to detect hidden mechanisms. Empirical studies where conducted for each of these three testing strategies, in order to evaluate and assess their effectiveness and their feasibility and provide some experimental trends.

To complete the process of testing the internal part of the system, chapter 5 focuses on the external part, namely the issue of bypass testing. The concepts of bypass testing are leveraged to provide a new tool, the bypass-shield that is to be used for both auditing and protecting web applications. The tool is build by analyzing the web application web pages and parsing all possible user inputs. The objective is to automatically be able to lift the client-side constraints and then to enforce them in the protection tool that is located on the server-side. For this approach as well, an empirical study on four popular web applications was conducted to prove the efficiency of this approach.

The second part of this thesis goes beyond testing and proposes new model driven approaches for building secure systems. Two approaches are proposed. The first one will be presented in Chapter 6. It is based on a generic metamodel for access control policy. It allows access control policies to be automatically deployed. In addition, framework implemented mutation testing in a generic way and allows testing artifacts to be automatically built. The main benefit of this approach resides in taking into account testing at an early stage of the modeling process and provides a universal certification process for testing access control mechanisms.

The second approach presented in Chapter 7 aims at solving the issue of hidden mechanisms. The objective is to be able to built fully flexible and reconfigurable access control mechanisms. The solution is based on MDE concepts, namely model composition and benefits from the advances in the field of model@Runtime to keep the access control policy synchronized with the system.

Contributions

Several parts of this dissertation have been published in international conferences [6-12], in a national conference [13], in international workshops [14-17], in addition to a Journal (The paper still under review. It is an invited paper selected as one of MODELS'08 best papers).

In addition, I contributed to imagine and built a new security tool, the bypass-shield. This tool is part of the ANR funded project DALI. I contributed to this project by working on this security tool.

Chapter 2
Background and state of the art

This chapter presents the background and the previous work related to the fields of application security, security modeling and security testing.
This chapter is focusing on three different dimensions that are:

- Software security: we present the main classical concepts of security, the "big picture" and then we highlight the main issues related to application security. All these points are detailed in Section 1 and Section 2.
- Modeling security: we present how model-based approaches are used for security. We focus specially on access control, which is the main topic of the thesis work. This part is detailed in Section 3.
- Security testing: we introduce software testing in general. Then we focus on the specificities of security testing in application by showing the main existing approaches. All this is presented in Section 4.

2.1 An overview on computer security

Computer security aims at protecting the information and the services from attacks which stretch from performing unauthorized access of information or services, tampering information, to targeting the services for shutting them down or slowing them.
The main objective of computer security is to guarantee confidentiality, integrity and availability (CIA). This objective is two fold:

- Protecting the information and the services
- Keep providing services running in an acceptable way and fulfilling users' requests.

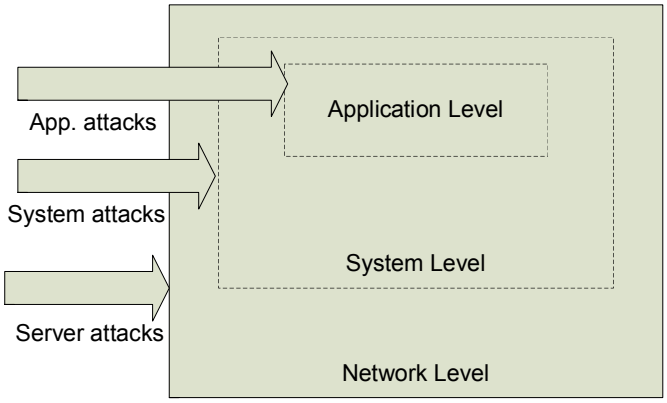


Figure 2 - Security levels

Computer security can be classified into three main levels as depicted in Figure 1. Each level has its own security threats and, on the shielding side, the corresponding security mechanisms to deal with these threats. There are three security levels:

- Network security: It involves tackling threats targeting the network. The main threats are: Denial-of-Service DOS, or Distributed DOS, network intrusion.

- System Security: It is related to the OS and includes protecting against all sorts of malware (virus, worm, spyware etc.). The protection mechanisms involve antivirus, anti-malware. OS level access control mechanisms, firewalls, etc.
- Application Security: It is composed of the threats targeting a specific application. It includes unauthorized access, information theft, and misuse of the application etc. The security mechanisms include Access control mechanisms, application level encryption/cryptography etc.

The field of computer security is very large. This thesis will focus on the application level security. The main objective is to provide new approaches for testing access control mechanisms at the application level and for bypass testing.
Next section will focus on application level security and present in details the threats and the security mechanisms.

2.2 Application level security

This section introduces the main concepts related to security in web application. In order to draw the big picture, it is important to begin with presenting what security is about. Then we present the main threats to security; bugs, flaws and vulnerabilities and leading to these threats. Finally, we present the techniques and the existing mechanisms to secure web application. This section focuses on access control which will be one of the main topics treated in this thesis along with bypass testing.
Security is an emergent, transversal and extra-functional property of software systems. It is mainly a quality that emerges from the whole software systems and is not related to only one of its components. It is transversal since it involves all the components and pieces of code composing the software system. Security has to be taken into account in all system code. Finally, security is an extra-functional property since it cannot be captured in only one piece of code. It involves all steps of software development process.
Figure 3 shows an overview of application level security. End users interact with the system through GUI interfaces. Security mechanisms are in charge of protecting the applications assets; by controlling access to functions and data, checking user inputs to prevent from attacks etc.

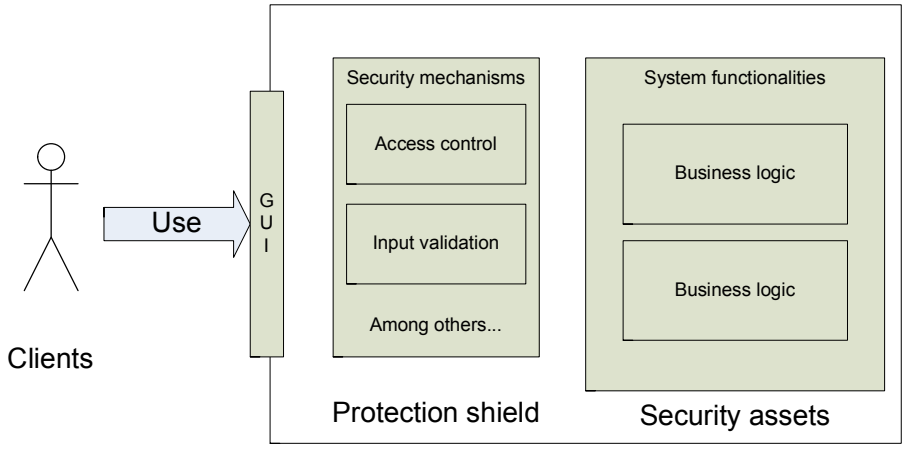


Figure 3 - Overview of application security

2.2.1 The security threats

Threats to security are due to *bugs* and *flaws*. Bugs are software defects. A bug in a software system leads to a failure. Some failures lead to disclosure of sensitive information (credentials, password, and important information about the underlying software). This is due for instance to an inappropriate handling of exception, or critical bugs in the security

mechanisms, in the access control mechanisms or in the implementation of the authentication procedure.
Security threats are also due to flaws. Flaws are security problems which may lead to *vulnerabilities*. Vulnerabilities are flaws that can be exploited by potential hackers. In other terms, vulnerabilities are exploitable flaws. Flaws are not necessarily exploitable, for example when:

- The flaw is located in internal functions that are not used or accessed directly by user.
- The flaw is located in a dead portion of code or in a component that is never executed.

An interesting taxonomy of the most important software errors is described in [18] . This repository presents the main errors leading to the most important and prevalent classes of attacks.
A common list of known and published vulnerabilities is maintained and regularly updated by the CVE [19]. This list contains all the known and published vulnerabilities targeting specific software systems (like browsers, email readers, etc.).
The OWASP community maintains the list of top 10 most prevalent flaws [20] for web applications. This list highlights the main weaknesses leading to security flaws. Lack of robust input validation is obviously the main cause of flaws (like injection flaws, XSS). In addition, weaknesses in the implementation of security mechanisms such as access control or misconfiguration of the application can lead to several security flaws (e.g. in the Top 10 list: broken authentication and session management, insecure direct object reference, and finally failure to restrict URL Access).
The most important vulnerability according to the OWASP classification is code injection which occur when user supplied data is sent to an interpreter to be included in a command or a query. There are many kinds of injection attacks; SQL, LDAP, XML, XPATH Injection etc. To show how this attack is performed, we present an example of SQL Injection.
SQL injection attacks exploit the fact that some web applications do not perform the input data validation, and create SQL query strings dynamically. The aim of SQL injection attack is to append SQL statements inside the application SQL query and execute a different query.
In order to understand the SQL injection, let us focus on the following example, this code is used inside a java class. (It is a “hello world” example always presented by many to show how SQL injection works):

```
String unsafeQuery = “Select * FROM users WHERE user = ‘“+ inputUser + “’ and password = ‘“ + inputPassword + “’;”;
```

A rogue user (or a hacker) may type this as *inputUser* and *inputPassword*:

```
InputUser?  ‘ or 1=1; --‘
inputPassword? Nothing
```

Now we end up with the following query:

```
unsafeQuery = “Select * FROM users WHERE user = ‘’ or 1=1; --‘ and password = ‘Nothing’;
```

The last part of the query that starts from “--“ is ignored. It is considered as comment because the string *“--“* indicates a comment. The query that will be executed is this one:

```
Select * FROM users WHERE user = ‘’ or 1=1;
```

This query is a tautology and will return the list of all users. In this case, the attacker succeeded to log in (illegally). He was able to actually modify the query and execute another one.

2.2.2 Security mechanisms

Securing an application involves:

- Implementing efficient security mechanisms.
- Adopting rigorous and secure development methods.

Security mechanisms involve, proper *authentication* [21-23], to guarantee that only authorized users have access to the application. In addition, the users access to resources, data or services have to be regulated through the use of *access control* [24] . This helps to guarantee that only authorized user to specific functions. Finally, once access is granted the way the system is used have to be controlled through *usage control* [25, 26]. This helps guaranteeing that the system is well used. The overall objective is to guarantee that the system users are well *authenticated* and that the system resources are correctly *accessed* and correctly *used*.

In addition to these security mechanisms, rigorous and known secure coding is important in helping develop reliable and flaw-free applications. In fact, vulnerable code and ways to avoid it are available in online security community website (for instance in the OWASP project website).

Furthermore, when secure coding guidelines are not followed, or in the case of a legacy system that needs to be secured, specific security mechanisms can be used to mitigate and protect against attacks. These security mechanisms involve some specific protection components that aim at protecting against specific attacks (Cross Site Scripting/XSS, SQL Injection etc.).

Access control

Access control allows protecting system resources against unauthorized access. Actors of a given systems are granted (or denied) access to the system resources according to a specified access control policy. This policy includes rules describing how the actors of the system (users, programs, processes, or other systems) may use the system resources.

Access control policies are usually built based on access control models. Then, they are implemented through access control mechanisms, which are in charge of encapsulating the policy, integrating with the application and enforcing the access control policy.

Access control models

Access control models define what security concepts need to be expressed and how rules have to be expressed. During the last decades, a multitude of models have been introduced in the literature.

In order to better understand these models, we will show some examples of policies built according to these models. We show examples of models, namely RBAC, MAC, DAC and OrBAC. We finish this section by showing some examples of others advanced access control models:

RBAC: Role Based Access control

RBAC [27] is the most popular and widely used access control model. Several products in the industry (Databases, OS, Business applications etc.) rely on it.

RBAC defines three entities, namely; *users*, *roles*, *permissions.*

RBAC associates users with roles on one hand and roles with permissions on the other hand. Two types of rules have to be defined:

- UserRole: rules which have two parameters: a user and a role.
- RolePermission rules which have two parameters: role and permissions.

To illustrate how to build an RBAC policy, we use the example of a library management system. This system defines three types of users: students, secretaries and a director. The students can borrow books from the library, the secretary manages the accounts of the students but only the director can create accounts.

POLICY LibraryRBAC (RBAC)

R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook)
R5 -> RolePermission(Personnel ModifyUserAccount)
R6 -> RolePermission(Director CreateUserAccount)

According to the standard [27], RBAC may include other features such as SOD (Separation of Duty). There are two kinds of SOD; static separation of duty (SSD) and dynamic separation of duty (DSD). Roughly, SSD forbids users to be able to have two roles, while DSD does not allow users to having two specific roles during the same session.

MAC: Mandatory Access Control
MAC [28, 29] is suitable for multilevel secure systems (especially in the military context). Next the definition of MAC (Taken from Trusted computer System Evaluation Criteria):
"A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity".
MAC entities are Subjects, Objects and Clearances. Subjects are usually Processes or threads (executing user commands), and Objects can be files, ports, etc. MAC policies express the access of subjects to objects according to their clearance and to classification of objects. Subjects with a high clearance are able to read all kinds of objects. However, they are not allowed to write in objects that can be accessed by low clearance subjects. A classification of clearance determines if the access is granted or denied. For example, if Subject S1 having clearance C1 requests reading Object O2 having clearance C2, then access if granted if C1 >= C2. Otherwise, if C2 > C1, access is denied.
Here is an example of a MAC policy. The following policy defines two users and two objects and the rules specify their clearances.

POLICY systemMAC (MAC)
R1 -> SubClearance(process1 low)
R2 -> SubClearance (process2 high)
R3 -> ObjClearance (report1 low)
R4 -> ObjClearance (report2 high)

DAC: Discretional Access Control
The definition of DAC [30] (taken from [31]) :
"A means of restricting access to objects based on the identity of `subjects and/or groups to which they belong. The controls are 'discretionary' in the sense that a subject with a certain

access permission is capable of passing that permission (perhaps indirectly) on `to any other subject.".

A DAC policy expresses a set of Subjects and Objects and access types. A rule is the combination of one Subject, one object and one access type. We consider DAC as used for file systems. Objects include files, directories or ports (or others) and Subjects include users or processes. The policy can be seen as a matrix where the values are access types. Access types include three access types (r: read, w: write and x: execute) and two special ones, which are *control* and *control with passing ability*. The control access type enables its holder to modify the users' access types to that object. In addition to this, the control with passing ability enables the user to pass this control ability to other users.

DAC is suitable to be applied to file systems. For example, applied to file systems, the access types of the DAC:

- r : permission to read the object
- w: permission to write
- x: permission to execute
- c: control permission, the ability to modify 'r w x' permission for this object.
- cp: control and the passing ability of control.

Here is a simple example of DAC policy. The following policy defines two subjects (Tim and Admin). Tim can read or execute file1, while admin has the right to read, write and execute the file in addition to the control and passing ability.

POLICY systemDAC (DAC)
R1 -> DACRule(Tim r file1)
R2 -> DACRule(Tim x file1)
R3 -> DACRule(Admin cp file1)
R4 -> DACRule(Admin r file1)
R5 -> DACRule(Admin w file1)

OrBAC: Organization based access control

OrBAC [32], OrBAC introduces the notion of organization, which defines a scope of application for access control rules, and it is very useful when dealing with interoperability between different organizations. It also adds the notion of context; which makes it possible to define dynamic policies.

OrBAC defines the following entities; organizations, roles, activities, views, context. It allows defining three kinds of rules:

- *Permission(Organization, Roles, Activities, View, Context).*
- *Prohibition(Organization, Roles, Activities, View, Context).*
- *Obligation(Organization, Roles, Activities, View, Context).*

In addition, OrBAC supports hierarchies over organizations, roles, views and even delegation [33] and administration of the policy [34].

To illustrate OrBAC policies, we use the example of the library management system.

POLICY LibraryOrBAC (OrBAC)
R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Permission(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

Other advanced access control models

As we have previously stated there are a multitude of access control models. In fact, a substantial work [35-37] focused on access control formalization that guarantees flexibility and allow policies to be easily modified and updated. In this context, we present some of the work related to this area. In [38], Bertino et al. proposed a new Access control model which allows expressing flexible policies that can be easily modified and updated by users to adapt them to specific contexts. The advantage of their model resides in the ability to change the access control rules by granting or revoking access based on specific exceptions. Their model provides a wide range of interesting features that increase the flexibility of the access control policy. It allows advanced administrative functions for regulating the specification of access controls rules. In addition, their model supports delegation, which enables users to temporarily grant other users some of their permissions (like a director would do during his vacations).
In addition, Bertolissi et al. proposed DEBAC [36] a new access control model based on the notion of event and that allows the policy to be adapted to distributed and changing environments. Their model is represented as a term rewriting system [39], which allows specifying changing and dynamic access control policies.

Access control mechanisms
Once the access control policy is expressed using one the available access control models, security mechanisms to implements these policies have to be created in the Business Logic of the system. The Business Logic is the part of the program that performs tasks that are specific to the application domain (like processing data, reasoning and computing results specific to the logic of the application). The architecture commonly adopted is depicted in Figure 4:

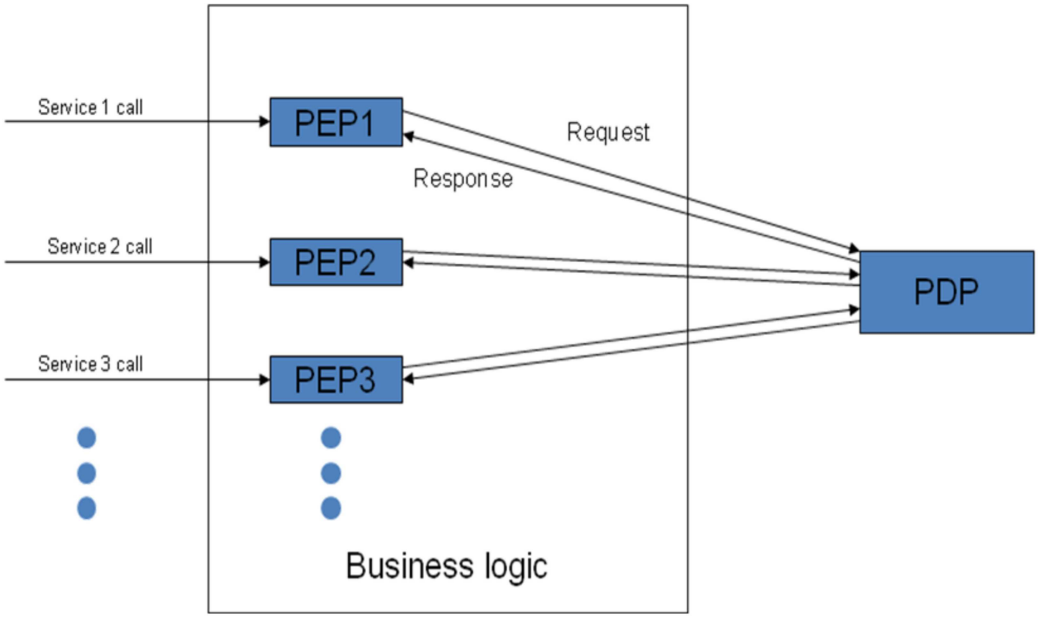


Figure 4 - Architecture of the Access control mechanisms

This architecture involves two main components, namely the PDP and the PEP:

- *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 access control policy, the PEP sends a request to the PDP to get the suitable response in the current context for the service requested by the user. 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 taken. It encapsulates the Access Control Policy and implements a mechanism to process requests coming from the PEP and returns a response which can be *deny* or *permit*.

To support the implementation of the security mechanisms and their integration in the application, several frameworks exist in the industry, like for instance XACML, JAAS or EJB. We present briefly the XACML framework that we will use when proposing a model-based framework for access control in the following chapters.
XACML, the eXtensible Access Control Markup Language is an OASIS standard which allows defining access control policies in XML files. Several tools support and help to read, write and analyze XACML files (like the sun XACML implementation and Google enterprise Java XACML [40]),
A policy in XACML stores a set of rules. A rule has an Effect property (permit or Deny) and may define a condition element. The root element is either a PolicySet or a Policy. The Target elements contain the set of Subject, Action, Resource and Environment. These elements contain the attribute for which the rule, the policy or the policy set is applied. When there are different possible responses (more than one target is matched), a combining algorithm is used to decide which rule to apply.
XACML offers a flexible but a very verbose language for specifying policies. Therefore, manually writing XACML files is error-prone, as shown by Xie et al [41, 42].
Moreover, it was not defined according to a specific access control model (RBAC, MAC, DAC or OrBAC). Adaptations are necessary to use XACML for RBAC or ORBAC policies. There exists actually an XACML profile for RBAC defined by OASIS [43] and for OrBAC [44] which is actually an extended version of RBAC profile. These profiles define the way an RBAC or an OrBAC policy can be expressed using the XACML language.

Specific security mechanisms
Researchers have proposed several specific security mechanisms to thwart specific security threats like XSS [45-47], cross-site request forgery [48-50], Clickjacking [51] or SQL injection attacks (SQLIA). To illustrate how these mechanisms are used, we detail two approaches targeting SQLIA; *SQLRand* and a learning based approach.

SQLRand: a radical shift
This technique was proposed by Stephen W. Boyd et al. [52]. They apply instruction-set randomization against code-injection attacks (mainly buffer overflow based attacks) to the problem of SQL injection. The idea behind the concept is to create randomized instances of the SQL query language by using random keywords. The SQL keywords (like select or update etc…) are appended by a random number called a key. All application queries are randomized inside code scripts or sources. Then a 'de-randomizing' proxy is used to recover the query (the proper SQL syntax) and send it the database management system. The malicious input is rejected by the proxy parser, as it uses unknown keywords (the assumption is made that the user will not be able to guess the value of the key).
For example, we have the following query:

SELECT gender, avg(age) FROM cs101.students WHERE dept = '$dept' GROUP BY gender

To randomize the query, a utility appends automatically the key (here it is 571):

SELECT571 gender, avg(age) FROM571 cs101.students WHERE571 dept = '$dept' GROUP571 BY571 gender

A rogue user may attempt to type "*' or 1=1; --*" the query became :

SELECT571 gender, avg(age) FROM571 cs101.students WHERE571 dept = '' or 1=1 ; -- GROUP571 BY571 gender

The SQLIA fails because when the parser tries to parse this query the "or" is identified as unknown keyword since it does not contain the key "571".

Figure 5 presents the architecture proposed by SQLRand approach (from [52]) :

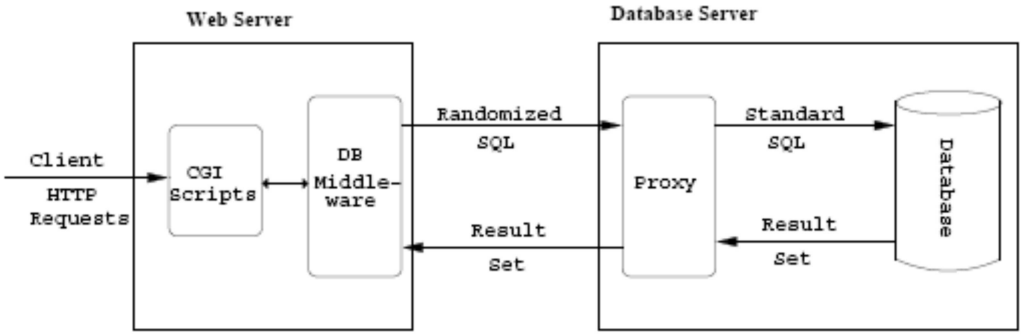


Figure 5 - SQLRand architecture

The drawback of this approach is that it requires developers to adopt a new programming paradigm. Moreover, it is difficult to apply to legacy software as it requires the medication of all applications queries.

A learning based approach

This approach [53] is based on the idea that an application always performs the same types and models of SQL queries and if a new kind of query is performed then this is an indicator of a possible SQLIA.

This approach works as follows:

- During a training period the security mechanism learns all the models of queries that can be performed by the application.
- The security mechanism monitors checks every query performed by the application to see if it is linked with an already known model. If it is not then an alert is raised.

The security mechanism is located between the application and the database server. Therefore, it is independent from the application language and platform. Figure 6 presents an overview of the system (from [53]).

One limitation mentioned by the authors for this approach is that it might not detect attacks when the structure of the malicious query matches the structure of another accepted query. In this case, the attacker changed the query in a way that makes it similar to another query normally used by the application. The security mechanism is not able to detect this attack. This potential shortcoming can be overcome by making the association at a finer level.

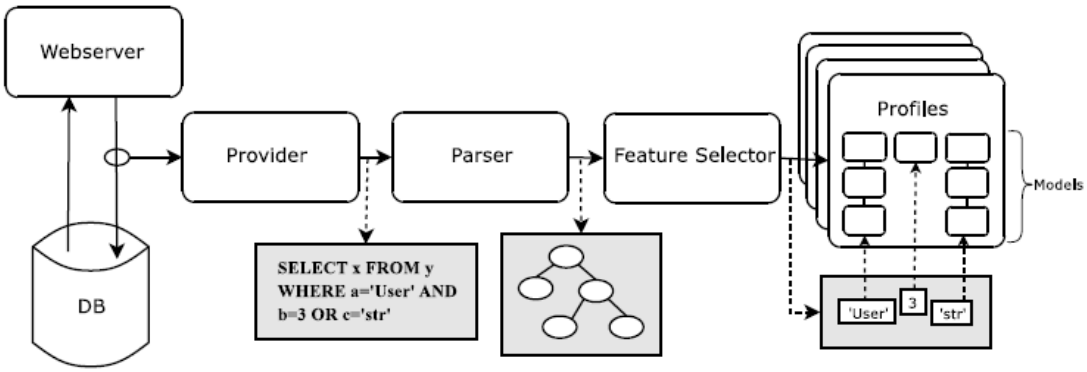


Figure 6 - Overview of the learning based approach

All these security mechanisms help protecting against specific attacks (SQL Injection, XSS etc). They can be included in legacy systems and plugged into the system even after it was deployed. However, no security mechanism is perfect and the best way to secure a system is to build it from scratch using the best security guidelines. It is important to take into account security at an early stage of the development cycle. Next section provides an overview of the state-of-the-art approaches for building secure systems.

2.3 Modeling security

MDE or Model Driven Engineering aims at helping organization to cope with the complexity of developing enterprise software products. It promotes the use of models and specific domain languages (DSL) as first-class artifacts. With the variability of requirements and the growing complexity of software, modeling approaches are suitable to overcome these difficulties. It is only in recent years that modeling has been applied to security. With the success of the UML (Unified Modeling Language) language, researchers have focused on how integrating security in the modeling process, and how providing efficient tools to deal with security at design level.

As shown in Figure 7 (from [54]), security aspects can be included to provide a secure process during the application lifecycle. In addition to security tools, manual audit of the application code and penetration testing which occur at the end of the development lifecycle, other security processes are used like risk analysis [55-57], the specification of security requirements and abuse cases.

At the early stage of the development, abuse cases (also called misuse case) are used to express how the system can be misused by potential attackers. The abuse cases [58, 59] are expressed using UML diagrams. Use cases are used to show the main actors, while sequence diagrams detail step by step how the attacks are performed.

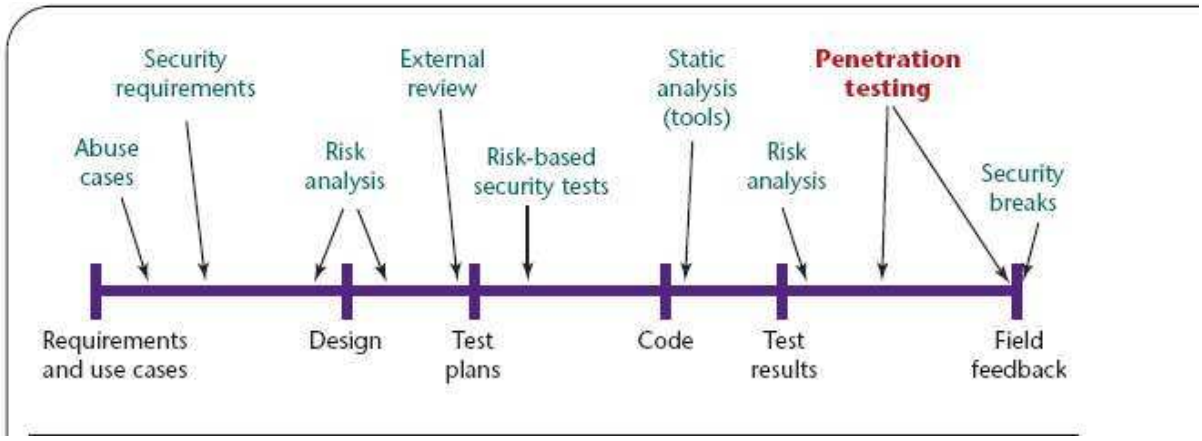


Figure 7 - Secure software development lifecycle

Figure 8 show a simple example of a sequence diagram illustrating an XSS attack. An email with a link is sent to victim user which takes him to the web application websites. The link contains malicious JavaScript. Once this script executes on the victim browser, sensitive information is sent the attacker. This attack is due to a vulnerability in the web applications which allows executing that malicious script.
This approach helps understanding how the attackers operate and therefore how the system should be adapted to protect against these attacks. In addition, these documents are used by security testers to create and perform security tests.

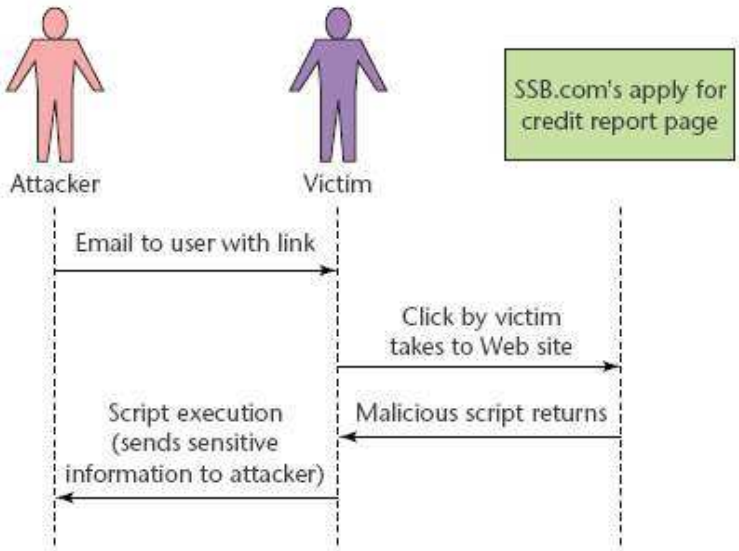


Figure 8 - Sequence diagram for an XSS attack

Moreover, several approaches were proposed for deriving the security requirements [60-62]. For example, Van Lamsweerde [63] proposes to express the security requirements by modeling attacks as anti-requirements. Then a process is followed to infer new countermeasures to be included in the requirements.
This approach is based on epistemic logic and the use of security patterns to express the attacks. The actors used in these models are the application users (the potential attacker, the admin, the maintenance personnel etc.) and the system. The method for deriving security requirements is as follows:

- The security concerns are expressed as objectives and expressed using epistemic logic by reasoning about the involved agents. These concerns are related to some security properties like for instance confidentiality, availability or integrity.
- The security threats are modeled as anti-goal. Attackers are therefore seeking to attacks the system to fulfill these anti-goals.
- The derivation process allows finding if there are some vulnerability that make the attack succeed. This derivation is performed by asking security experts and applications domain experts how an attacker that knows how the system works can be able to attack it.

This approach helps establishing the security requirements. It is interesting since it makes it possible to find new threats that are specific to the application domain at a high level of abstraction.

In addition to these approaches, substantial work focused on providing model based methodologies for security. Some approaches were proposed to help modeling access control formalisms such as RBAC or MAC in UML models. Doan et al. proposed a methodology [64] to incorporate MAC in UML diagrams during the design process. While RBAC was modeled using a UML diagram template [65, 66]. This template expresses the main RBAC concepts including the notions of user, role, permission and session. In addition, this template allows expressing the separation of duty constraints (both SSD and DSD) in addition to hierarchy over roles. This template is depicted in Figure 3 (taken from [65]). The template parameters are preceded with '|'. Actually, these parameters have to be instantiated later and mapped to application concepts which are already in the primary UML model.

In order to express multiplicity constraints, some OCL constraints are added. For example, as illustrated in the figure, there should be only one *object* and one *operation* to compose *permission*.

The next step is to instantiate this RBAC template by mapping the parameters to elements in the primary UML model (the model of the application). The final step involves composing these two models; the instantiated template and the primary model to end up with a model of the application incorporating the RBAC concepts. At this point, the RBAC policy has still to be expressed and object diagram templates are used for modeling RBAC policy rules.

All these modeling approaches allow expressing access control formalisms in design phases. These approaches include the modeling process of access control policies into the modeling of the application functionalities which harms the flexibility and controllability of the access control policy. In fact, separating the access control mechanism from the application allows more control (for instance, it is easy to modify the policy) and facilitates the validation of security (because security is centralized in known locations).

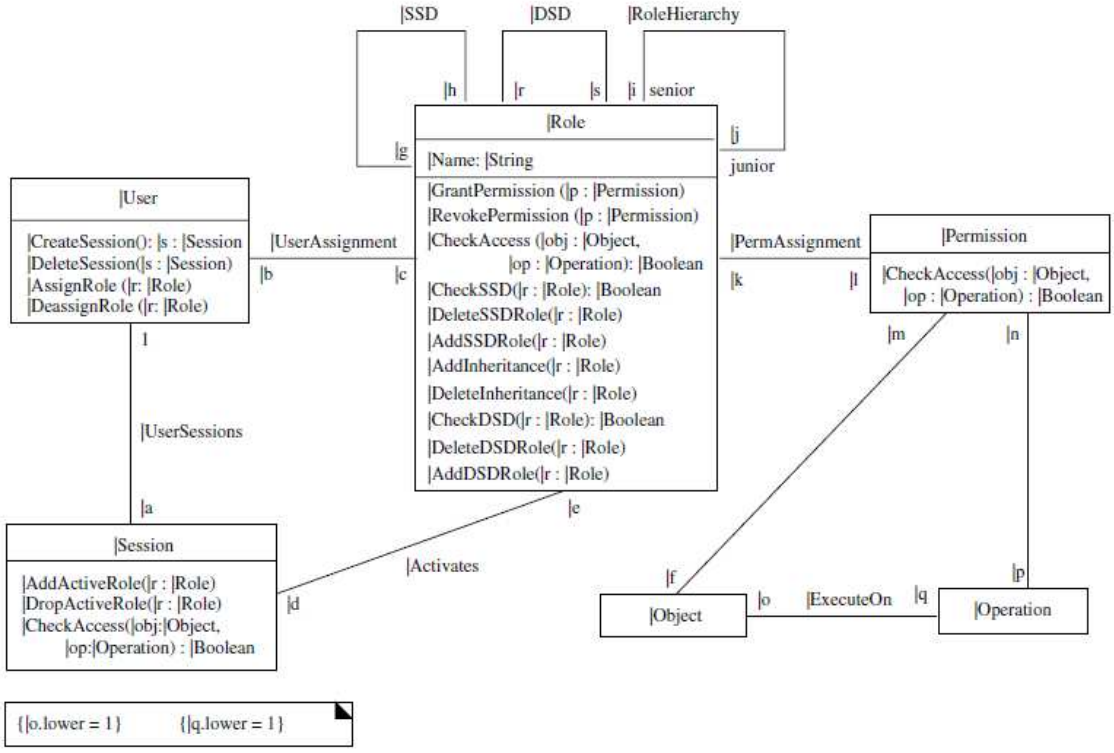


Figure 9 - RBAC template

An advanced approach is proposed by UMLsec [67, 68], which extends UML model security requirements in UML diagrams. More precisely the approach introduces a UML profile allowing completing UML diagrams (such as activity diagrams and statecharts). The main benefit of this approach resides in providing formal methods and techniques for a thorough analysis of security in UML diagrams. The security requirements that are included in the models help evaluating these models and finding possible flaws in the design.

In addition, Lodderstedt et al. propose SecureUML [69], which allows generating security components from specific models. The approach proposes a security modeling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure. More precisely, they use the Meta-Object facility to create a new modeling language to define RBAC policies (extended to include constraints on rules). They model policies for different examples of distributed system architectures including Enterprise Java Beans and Microsoft Enterprise Services for .net.

Aspect-based approaches have been applied to access control [70]. Ray et al. apply aspect oriented modeling (AOM) to integrate RBAC policies. RBAC policies are modeled in aspects that are then composed with the application primary model. This approach joins the approaches that model RBAC in UML diagrams, with the benefit of using AOM, for a flexible definition of access control policies and the automatic weaving into the application model.

There are two main limitations shared by all these approaches. They always focus on one specific access control model (RBAC or MAC etc.) and therefore cannot be easily applied to other access control models. The other main limitation is that they do not consider testing access control mechanisms or provide a framework or testing artifacts to perform testing. The process of testing is separated from the modeling and integration of the access control mechanisms. We will tackle these issues and provide a new model-driven process that can be

applied to several access control models and that provides testing artifacts to help performing access control testing.

2.4 Testing and Security Testing

In this section, we briefly present the main concepts around software testing and detail the mutation analysis technique. We adapted this technique to the context of security. Then we present some previous work related to security testing. Finally, we highlight the previous work on testing access control mechanisms.

2.4.1 Software testing

Software testing embraces most of the application domains and covers all the stages of the various existing development processes (V, Xtreme Programming, agile approaches). It is thus impossible to present this domain in depth in this thesis. In this thesis, we are consistent with the definition of software testing (taken from[71]):
"Software testing consists of the dynamic verification of the behavior of a program, on a finite set of test cases, suitably selected from the usually infinite executions domains, against the specified expected behavior".
Software Testing appears to be a powerful and efficient way for validating programs [72-75]. It can be used along with other existing techniques like model-checking and theorem proving or any other formal technique, when the suitable formal models are available. Software testing can be used by exploiting incomplete or partially ambiguous elements of specification, such as usage scenarios. In this section, we present the main concepts of software testing, and then detail a technique which is widely used to qualify a set of test cases, namely *mutation analysis.*

Main software testing concepts
The main goal of software testing is detecting errors in the software. When a test executes and fails, this failure is usually due to a fault which was introduced by the programmer. To be exact, a failing test case essentially reveals an inconsistency between what the system is supposed to do (from the point of view of the tester, admittedly derived from the specification) and what it effectively computes (from the point of view of the developer admittedly derived from the specification). The program is usually suspected as faulty, while the test case may be faulty (a wrong interpretation of the specification) or, ultimately, the specification itself can be suspected.

Software testing is tightly linked to the software development processes. Therefore, the testing techniques should be adapted to the way the system is developed. For instance, the V-model implies that a specific software testing technique have to be applied for each step. As illustrated by Figure 4, there are four kinds of software testing steps, namely unit testing, integration testing, system testing and acceptance testing. Unit testing allows validating low-level program units. The kind of program unit depends on the programming paradigm. For example it is the class for Object-Oriented programming, and the function/procedure for classical imperative programs. Integration testing targets validating the interactions or interfaces between modules, classes or the software component. It is an intermediary step before system testing which validates the whole system by including all the programs units and components which were previously validated by integration testing. Finally, acceptance testing checks that the systems meets the initial requirements and corresponds to the customer needs and expectations.
For each one of these testing techniques, there are tools which can be used by software testers like for example, the *JUnit* framework [76] used for unit testing in Java, *EasyMock* [77] or

Selenium [78] or *JWebUnit* [79] which allow tests to be automatically for web applications in order to perform system testing.

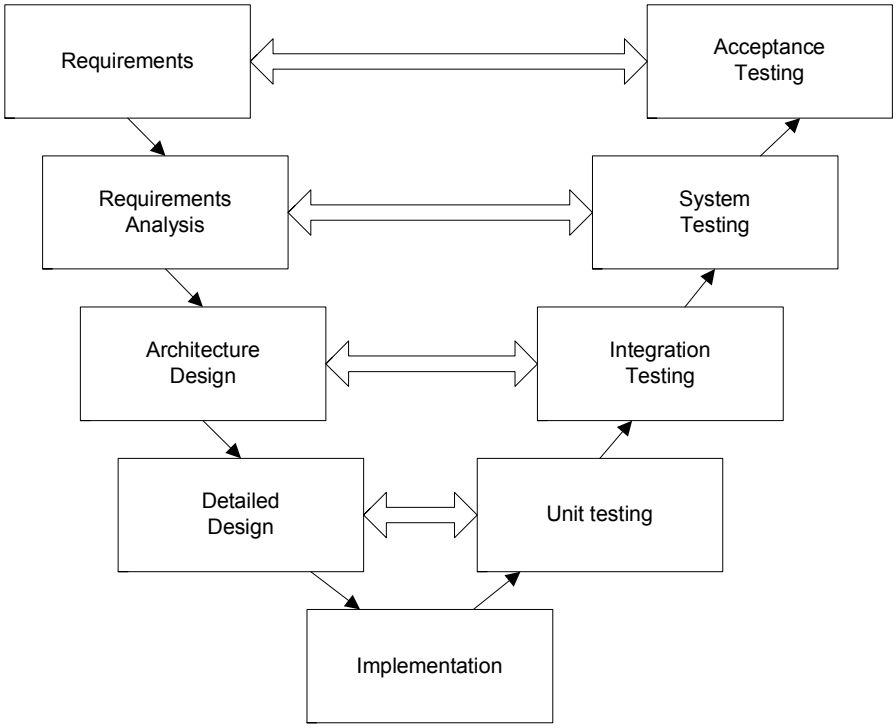


Figure 10 - The V-Model software development process

The main issues of the software testing research are:

- *Test selection:* how to establish relevant and efficient criteria to select the subset of the input domain which is relevant to represent the overall input domain of the system under test.
- *Test generation:* how to create a set of test cases corresponding to the test selection and how to automate the process of testing. Several different techniques are studied in the literature, such as model based testing (generating tests from a specified model, UML class diagram, state charts etc.).
- *Test oracle:* how to define the oracle function which is in charge of deciding whether the test case fails or succeeds.
- *Test qualification:* how to assess the quality of the test cases. There are several ways to do this; code coverage, mutation analysis etc.

In the next subsection, we introduce and discuss the concept of mutation analysis. This testing technique will be used and adapted to the context of security.

Mutation analysis

Mutation analysis aims at evaluating the quality of test cases. Since we never know with certainty whether the system is still faulty after test cases execution, the goal of mutation analysis is to "test the tests". Indeed, mutation analysis aims at testing whether the test cases are efficient to detect (seeded) faults. The idea is that test cases are efficient if the are able to detect errors in the program. One way to assess the quality of tests is to challenge them in detecting errors injected intentionally in the program. The final goal of this process is to be able to improve the tests and hopefully be able to detect real faults in the system.

The process it presented in Figure 5. Errors in the program under test (PUT) are injected systematically using *mutation operators*. This leads to having a set of faulty versions of the program which are called *mutants*. Each mutant includes a simple error. The tests are then

executed against the mutants. When test cases are able to detect the injected errors, the mutant is said to be *dead*, otherwise it is considered as *alive*. The quality of the test cases set is related to their capacity of killing mutants, which is estimated by the mutation score. The *mutation score* is the number of killed mutants over the overall number of non equivalent mutants. Indeed, an *equivalent mutant* is a mutant that behaves exactly as the original program, in other word the mutant is functionally equivalent to the original program. For example, when we replace x=a with x=a+0, the created mutant program semantic is exactly the same and it is impossible to create a test that kills this mutant. Hence, it is important to detect and remove all equivalent mutants.

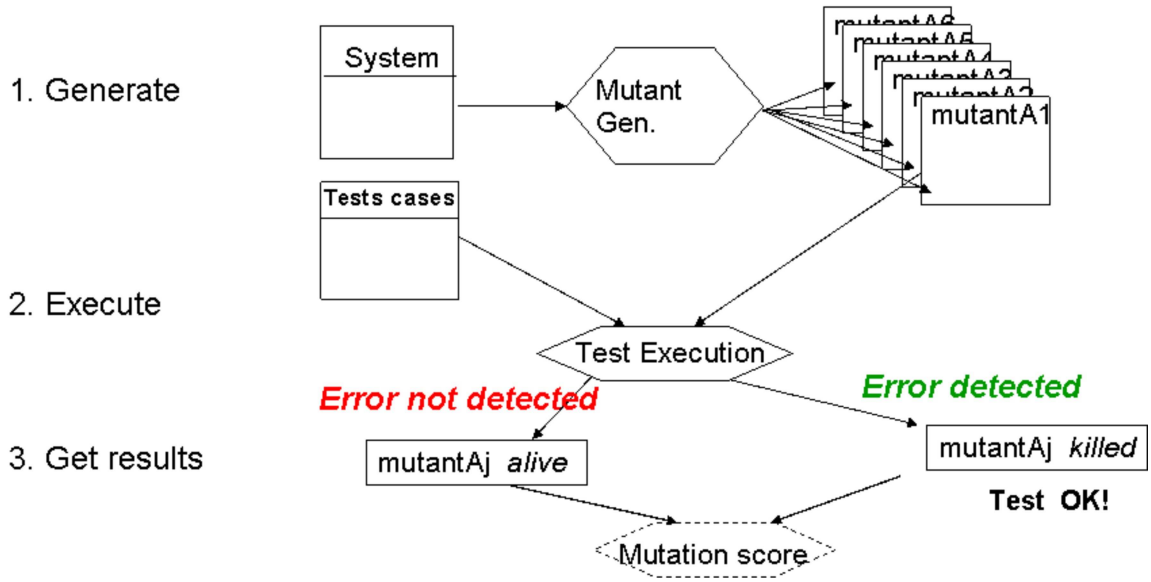


Figure 11 - The process of mutation analysis

Mutation operators are used to automate the generation of mutants. Each operator is specialized in seeding one type of errors (for example changing Boolean expressions, replacing a variable with another, replacing method call with another one etc.). Several sets of mutations operators were defined in the literature for specific language and programming paradigms; for Java [80-82], for C [83], for C# [84, 85]. The mutation operators aim at reproducing faults that programmers would do. The effectiveness the mutation analysis tightly depends on the quality of the mutation operators.

The validity of the process of mutation relies of two hypotheses:

- The competent programmer.
- The coupling effect.

The first hypothesis is based on the idea that programmers usually write code that maybe faulty but who is not far away from the correct program. The program they write may contain some small errors but the modifications need to correct the program are in most cases minor.

The coupling effect hypothesis [86] relies on the assumption that complex errors are usually a combination of small errors. For this reason, mutation operators always introduce small errors. This hypothesis is important for mutation analysis. The coupling effect is what motivates the use of one seeded fault for each mutant and more importantly that improving the tests to make them able to detect these seeded faults will allow to detect real faults.

Andrew et al. [87] have presented an empirical study that shows that this correlation is realistic.

To evaluate the effectiveness of mutation analysis, a good approach would be to compare it to other testing techniques. Mutation was compared to data flow coverage [88-90] . Offut et al. [88] showed through an experimental study that test cases built using mutation easily fulfill

data flow coverage criteria and detect more errors. In addition, they compared mutation to four testing techniques [91] (other three techniques in addition to data flow coverage). They found the same results. Mutation is better in detecting errors and produces less but more efficient tests.
Besides from qualifying tests, mutation can be used for:

- Evaluating testing criteria: to assess and compare different criteria for test generation. Several sets of tests corresponding to these criteria are generated. Criteria are to be compared based on their mutation score.
- Test generation: In this case it is called mutation testing. Test data is automatically generated in order to kill mutants.

Several tools implement mutations were developed; MuJava [92], PlexTest(For C++ mutation [93]). Heckle (For ruby programs [94]).
Recently, researchers used mutation to purposes other than testing, for instance in [95], were mutation was used to automatically repair programs.

2.4.2 Security testing

Security testing is a critical aspect of security [96] since it is the main way to assess that the security mechanisms have been correctly implemented. Security testing is a wide area of research [97, 98]. We start by introducing the most used testing technique, namely penetration testing. Then we present some approaches based on fault-injection. We also present bypass testing and some approaches targeting very specific security vulnerabilities (XSS and SQL Injection). Finally, we present previous work in the literature concerning testing access control mechanisms.

Penetration testing
Concerning Security testing, the widely adopted approach is known as *penetration testing* [54]. This approach, which is currently the most used in the industry, involves security experts who try to perpetrate attacks by playing the role of a hacker and trying to attack the system and exploit its vulnerabilities. The idea is to reveal the vulnerabilities and flaws in the system that attacks are likely to exploit. The testers may rely on existing testing tools (a list of these tools is maintained in the OWASP website [99]) and they perform known attacks. This approach is effective to uncover weaknesses, flaws and vulnerabilities in the code. However, it has its drawbacks. Firstly, it depends on the expertise of the security testers. Secondly, the manual approach is limited and does not help covering all the aspects of security (when dealing with large systems for instance).

Fault-injection based testing techniques
Other approaches have been proposed in the literature, which apply fault-injection based techniques for security testing.
Mathur et al. applied fault injection to the application environment [100]. The application environment is perturbed by modifying environment variables, files or processes used by the application under test. Then the application has to resist to this perturbation and must not have an insecure behavior that may lead to security flaw.
Gosh et al. propose an automated injection fault injection analysis approach [101]. The idea is to inject faults to the application data flow and internal variables in order to identify the pieces of application's code that have insecure behaviors when the state of the application is disturbed. This approach is more generally known as fuzz testing. It has been generalized to several platforms and there are tools available to perform fuzz testing on different platforms

(for instance: JBroFuzz [102] which is a web application fuzzier, Sulley [103] a fuzz testing framework).
The majority of these approaches are black box approaches since they do not consider the way the system is build and its internal code [104, 105]. Godefroid et al. [106] propose a white box fuzz testing approach. They use traces to monitor the execution of the program under test with well-formed inputs. Then, they are able to infer constraints over the inputs. Finally, these constraints are negated using a constraint solver and then are used to produce new inputs. They implemented their technique in a tool called SAGE and applied it to some windows applications and were able to find unknown vulnerabilities.
Fuzz testing aims at validating the correctness of the application and specially targets the input validation code which is in charge of checking inputs submitted by users. This activity is very important since user inputs are the main vector of attacks. Another interesting testing technique focusing on input validation is bypass-testing.

Bypass Testing

Bypass testing is the process of testing web applications by bypassing client-side input validation and triggering the server-side input validation (if it exists). Bypassing is possible either via some browser plugins (like Firebug or Opera Dragonfly) or by simply using code to build the requests to be sent directly to the server (using Java or C++ for example).
Bypass-testing is well known and used by security experts to perform penetration testing. Offut et al. formalized this concept of bypass testing in 2004 [107] and they defined its main characteristics. The CyberChair web application (a popular submission and reviewing system used for conferences) served as a feasibility case study to provide initial insights on the efficiency of bypass testing strategy. They manually tested it using bypass testing strategy and they succeeded to discover serious bugs. For instance, they were able to submit papers without authentication by exploiting bypass testing. They also created a proof-of-concept tool, they have applied to test a simple case study STIS (Small Textual Information System), (a web application they built). In their approach, they have proposed three different strategies for generating test data. All these strategies target testing the robustness of the web application by sending invalid inputs, set of inputs or by violating the control flow (by breaking expected execution scenarios).
Offut et al. applied their approach to an industrial case study [108], a web application developed by Avaya Research Labs. They were able to discover 63 failure using 184 test cases.
To protect against bypass attacks, the only existing solution is to perform bypass-testing to locate the flaws in the server side code and improve it. There no specific approach for protecting against bypass-attacks. This thesis will tackle the issue of bypass-testing and propose a new approach that allows both testing and protecting web application against bypass-attacks.

Specific security testing

Other approaches target testing the application to discover specific security vulnerabilities (like SQL injection or XSS).
One of the most advanced techniques in this area is Ardilla, a tool implementing a new and efficient approach, proposed by Kiezun et al. [109]. Ardilla aims at detecting SQL injection and XSS vulnerabilities. Malicious data, corresponding to attacks are sent to the server. These inputs are generated in a way to explore every control flow path. Then Ardilla marks this data coming from the user as potentially unsafe (tainted), tracks the flow of this tainted data in the application, and checks whether tainted data can reach sensitive locations in the code (for instance, statements executing database queries). Ardilla found several XSS and SQL

injection related vulnerabilities that were not known before. This approach appears to be quite similar to the SAGE approach [106]. The main difference is that it targets two specific security vulnerabilities.

Testing access control mechanisms

Several other studies proposed techniques and tools for automatically testing access control mechanisms based on XACML [41, 110] or RBAC [111]. Fisler et al. proposed Magrave a tool for analyzing XACML policies and performing change-impact analysis [112]. The tool can be used for regression testing to identify the differences between two versions of the policies and test the PDP.

In [42], Xie et al. proposed a new tool Cirg that automatically generates test for XACML policies using Change-Impact Analysis. Several researchers have generated tests from access control policies given by various forms of state machines [113, 114].

Concerning the use of mutation, Xie et al. [115] proposed a mutation fault model specific to XACML policies. However, they had to deal with the problem of equivalent mutants due to the fact that mutants are seeded at code-level. Since Xie et al.'s work aims at testing the Policy Decision Point alone, implemented with XACML, their testing approach does not execute the business logic of the system.

The same PDP-alone based testing approach is considered by Mathur et al. [111], who also mutates RBAC models by building an FSM model for RBAC and use strategies to produce test suites from this model (for conformance testing).

Several formal approaches were proposed for testing access control policies [116, 117]; or to apply model-checking approaches on access control [118, 119]

Mallouli et al [116] proposed a framework for specifying and testing access control policies in OrBAC. They proposed a tool called SIRIUS, to automatically generate test sequence. The system under test behavior has to be specified using an extended finite state machine (EFSM). Then, the access control rules are included into the EFSM through specific algorithms (one algorithm for each kind of rules). To perform conformance testing, test sequence are generated to check that the access control rules are well implemented.

2.5 Conclusion

This chapter introduces the background and presented previous work related to security; especially security modeling and security testing. The remainder of this thesis will focus on these two research issues.

Concerning Security testing, there are two main subjects; which are access control testing and bypass testing. It is interesting to notice that the related work on access control testing (Xie et al. [41, 42, 110, 115] and Mathur [111]) focuses on validating the PDP against the security requirements. The idea is to check that the expressed policy (in XACML or in RBAC) is conforming to the requirements. This task is very important but does not validate that the whole security mechanism (PDP+PEP) is correctly implemented. The next chapters will consider the whole security mechanism and will provide new approaches for testing these access control mechanisms.

Previous work on bypass testing [107, 120] provides a powerful method for detecting weaknesses in the server side and to helping correct these bugs or flaws. It is however important to be able to provide a security tool that protects against these attacks. Modifying the code is costly and sometimes impossible when the code is not available. It would be very helpful to have a tool that protects web applications is a transparent way. Chapter 5 tries tackles this issue and provides a new tool called the *bypass-shield* that is automatically built to protect against bypass-attacks.

Concerning security modeling, there are several interesting approaches as previously detailed [65, 66, 68-70]. However, these approaches do not consider the testing process. They do not provide any framework that facilitates the testing of the access control mechanisms. Moreover, most of them supports only one access control model (usually RBAC, MAC or DAC). They cannot be applied easily to other access control models. Furthermore, As far as we know, no previous work dealt with the problem of maintaining the alignment of access control policies with the business logic. When the access control policy evolves, it is crucial to check that the business logic actually supports this evolution, and eventually make the needed changes. Chapter 6 and Chapter 7 deal with these issues and provide two practical solutions. The fist one is a framework for specifying, deploying and testing access control mechanisms. It supports many access control models by using a metamodel. In addition, testing artifacts are built to support access control testing. The second solution provides a model-driven process for building and integrating fully flexible access control mechanisms, making it possible to maintain the alignment between the evolving policy and the actual system. All these approaches are detailed in the next chapters.

Chapter 3
Security testing vs. Functional testing: Going beyond functional testing

Before investigating new research approaches for access control testing, it is important to evaluate to what extent it is helpful and efficient to reuse existing functional testing to test security.
This chapter tackles this key research question about how far functional testing can guarantee the correctness of security mechanisms. This question is treated through studying how access control policy can be tested using traditional testing techniques.
This chapter focuses on access control policy testing as a specific target for software testing. It proposes test criteria and analyzes two strategies for security test cases generation. On one hand, existing functional test cases are enhanced in order to consider security concerns in their verdict. On the other hand, test cases specifically targeting the access control mechanisms are selected from the access control policy. In order to validate these approaches for test generation, we define fault models for access control policies and adapt mutation analysis to access control testing. It is important to highlight that this fault model will be reused for dealing with other security issues. This model will be improved, generalized and used as a powerful tool when tackling other issues.
Three empirical studies, which are taken from Java web applications, are performed to obtain experimental trends. Results confirm that security testing is a necessary and specific task in order to obtain high confidence in security mechanisms, implementing an access control policy.
This chapter conclusion presents the key research directions and issues that will be treated in details in the next chapter.

3.1 Introduction

The access control policy for an organization addresses constraints on data or functions that can be accessed by external users (or programs) and by members of the organization. Such constraints are expressed using one of several access control models (DAC[30], MAC[28, 29], RBAC[27, 121, 122], TBAC[123], OrBAC [32]) These models allow the permissions or prohibitions to be described for any of the resources of the system (it may configure a firewall or define who can access a given service or data in a database). The most advanced models express rules that describe permissions or prohibitions that apply only in specific circumstances, called contexts. For instance, in the health care domain, physicians have special permissions in specific contexts, such as emergency. Also, some models provide means to specify different access control policies to be applied to the various parts of an organization (sub-organizations). The access control policy can thus specify what permissions and prohibitions should be in the system, depending on the contexts, roles and views.

The software development process involves building a system according to both the functional requirements and the extra-functional ones, including the access control policy. This process is a sequence of activities, from high-level analysis to design and implementation. In most cases, the deployment of an access control policy is not automatic and the correctness of its implementation has to be verified or tested. For example, it is possible to run test cases to get some evidence that the security mechanisms are correct with respect to the requirements (access control policy).

Access control policy testing as security testing is a necessary but hard step. We propose building test cases from the access control policy specification and run those tests on the system to reveal security flaws or inconsistencies between the policy and the implementation.

When generating the test cases, it is important to consider that the access control policy is strongly coupled to system's functionality: testing functions includes exercising many security mechanisms. The issue is to determine:

- Whether testing functions provides enough confidence in the security mechanisms,
- How to improve this confidence by selecting test cases specific to security.
- How to transform these tests and adapt them to security testing.

We propose two strategies for generating access control policy test cases: in complement of existing functional test cases or independently from them.

When generating test cases in a specific context such as security testing, the first step involves analyzing which faults (in our case security flaws) can occur in that specific context, and to define test criteria, to select and generate test cases. The definition of a fault/flaw model for access control policies leads to an adaptation of mutation analysis [124].

We define what access control policy testing means and we propose test criteria to generate test cases from an access control model. We use mutation analysis to obtain an experimental basis and compare the criteria and the strategies. This involves producing faulty versions of a system, each of them differing from the original one by the presence of a flaw in one of its security mechanisms. Good test criteria should lead to the generation of test cases able to detect these faulty versions, called mutants. We perform three empirical studies, to compare and discuss the criteria and strategies and highlight the specific issues related to access control testing.

This work aims at demonstrating that testing an access control policy is a task distinct from (but tightly related to) functional testing and can benefit from the existing functional tests, which can be selected, transformed and reused to effectively test security. The corollary contributions are the security flaws model (mutation operators), Access control test criteria and strategies, and mutation-based process for selecting functional tests and transforming them into security tests. Section 4.3.1 introduces the context and defines important notions of security testing. Section 3 presents an example and the chosen access control model called OrBAC. Section 4 proposes test criteria to guide access control test cases selection. In section 5, we adapt mutation analysis adapted for access control testing. We present the fault model for access control policies and the set of mutation operators. Section 6 details the empirical study, conducted on three applications, that allows addressing various issues, which arise when testing security.

3.2 Process and definitions

In this section, we present the overall process to derive test cases from requirements and propose definitions of the main notions.

Requirements for a software system include the functional description of the system as well as many extra-functional concerns (performance, 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 access control policy 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 access control policy may introduce specific concepts, and reuse most of the concepts and functions defined in the business model. The new concepts introduced for security implementation are taken into account in the refinement process, either during design or at deployment/coding steps. The access control 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 access control 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.

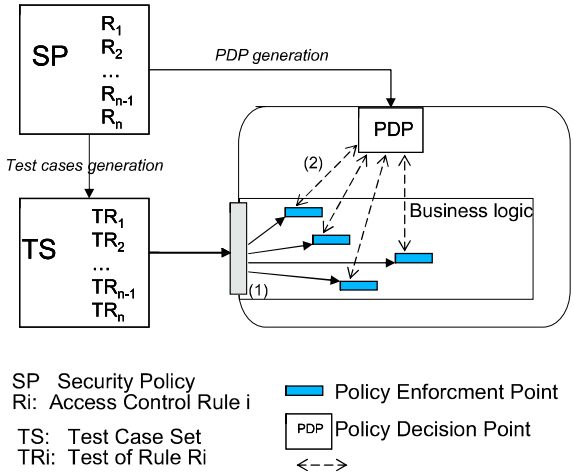


Figure 12 - Context for testing a secured system

The Figure 2 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 access control test cases are obtained using the access control model, while the functional test cases are derived only from the use cases and business model. Access control test cases are not only dependent on the access control policy but also refer to the use cases and the business model.

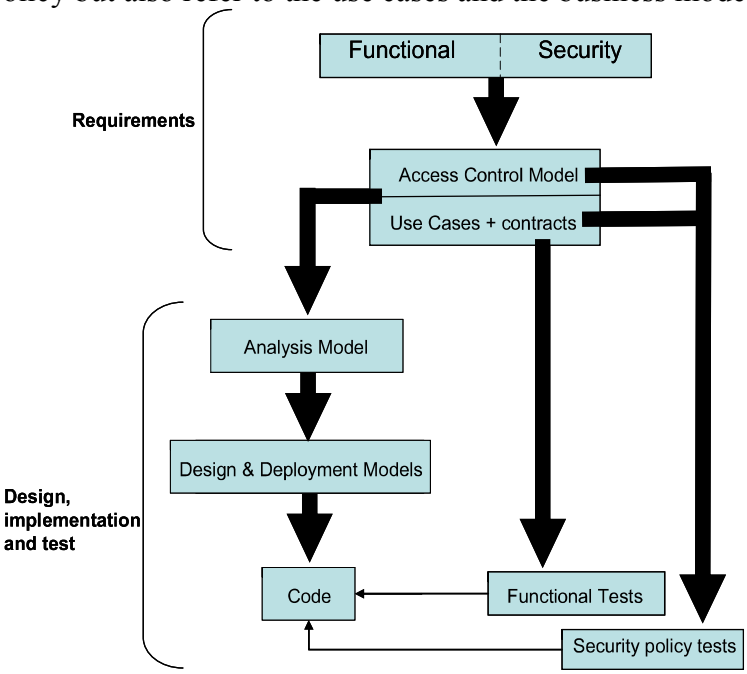


Figure 2 - Access control policy tests generation

3.2.1 Definitions

The following definitions help clarifying the main concepts and the differences between functional and access control testing:

- *System/functional testing:* 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 [125, 126]. By opposition to access control policy tests, we call these tests *functional*.

- *Access control policy testing:* the process of generating and executing test cases that are derived specifically from an access control policy. The objective of access control testing is to reveal as many security flaws as possible.
- *Security flaws:* occur when the interactions between PEP and the PDP are 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. manually adding an unexpected call to the PDP, erroneous parameters). 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.
- *Test case:* A test case includes 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 (access control, names of the tested access control rules) for access control ones.
- *Access control oracle function:* The oracle function for an access control 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 correctly executed. For instance, for an actor of the system who is allowed to access a given service, the functional test's intent involves having 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 an access control test. In practice, a security test will not contain all the assertions used in a functional test (that checks that the action was correctly executed).

As a running example, we consider a library management system (LMS). Its purpose is to offer services to manage books in a public library.

We use OrBAC [32, 127] as a specification language to define the access control rules (a set of rules specifies an access control policy). 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 language.

3.2.2 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 reserve 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

orders books or adds 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 being fixed, this book cannot be borrowed but a user can reserve it. The director of the library has the same accesses than the secretary and can 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.

3.2.3 Modeling an access control policy

In parallel of the design model, it is possible to model the access control policy from the requirements in terms of a set of access control rules.

A rule can be either permission or prohibition or an obligation. Each rule involves six parameters (called entities): a *status flag* indicating permission, prohibition or obligation, an *organization*, a *role*, an *activity*, a *view*, and a *context*.

The notions of Organization and Obligation rules are part of the OrBAC formalism. We choose to ignore these two notions. In most applications, the notion of organization is not useful since there is no interoperability within the application with other organizations or sub-organizations. In addition, obligation rules are different kind of rules representing the actions that actors must perform. They are usually implemented through some specific security functions independently from the system and they have to be validated separately (like obligations to perform specific tasks, sending mails, filling out forms etc.).

The domain involves *role names* RN, *activity names* PN, *view names* VN, and *context names* CN. An *access control policy* is thus set of rules defined by $Policy \subseteq S \times RN \times PN \times VN \times CN$.

The signature of an access rule is thus:

Status(Role, Activity, View, Context)

To express an access control policy, roles have to be defined. In the requirements we distinguish two main categories of roles: the users and the administrative personnel. We define the `Borrower` role which captures the main logical role played by a 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`. Then we define the services, called activities, which are constrained by security rules. All users can perform three activities: `borrow`, `reserve` and `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. From the LMS requirements, we identify the entities displayed in Figure 13. The graphical representation is the one from MotOrBAC, the tool implementing the OrBAC model. 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) that are restricted with access control, we identify the notions 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.

Several access control rules can then be expressed. For example, users are allowed to borrow books only when the library is open. This rule is defined as follows:

`Permission(Borrower, BorrowBook, Book, WorkingDays)`

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 (Borrower, BorrowBook, Book, Holidays)
Permission(Administrator, ManageAccess, PersonnelAccount, default)
Permission(Administrator, CreateAccount, BorrowerAccount, default)
Permission(Secretary, ConsultBorrowerAccount, BorrowerAccount, default)
```

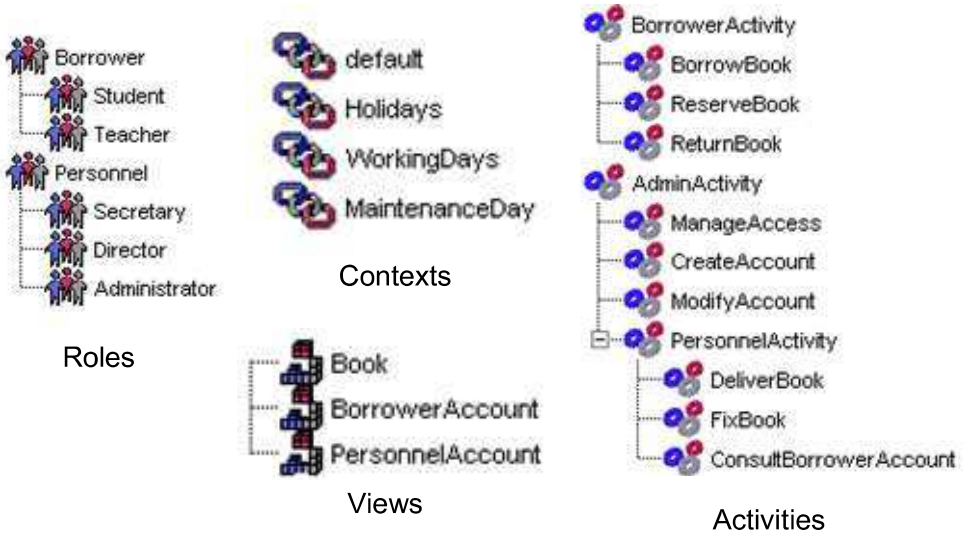


Figure 13 - OrBAC entities for the LMS

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

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

Based on hierarchies shown in Figure 13, 6 additional rules are automatically derived. The BorrowerActivity is replaced by each possible sub-activity (borrower 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(Borrower, BorrowerActivity, Book, WorkingDays)
Prohibition(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(Borrower, BorrowerActivity, Book, WorkingDays)
Prohibition(Teacher, BorrowerActivity, Book, WorkingDays)
```

We used OrBAC because it is supported by a tool called MotOrBAC. It can automatically detect the conflicting rules during the definition of an access control policy, using Prolog. One

way to solve the conflicts involves assigning priorities to rules. The rule with the highest priority is executed.

In addition, separations of duty (SoD) constraints are dealt with within the MotOrBAC tool. This allows the problems occurring due to SoD to be treated early by separating conflicting roles.

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.

3.3 Access control test cases selection

For access control 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 present an example of an access control test case that shows the difference between access control and functional test cases. Afterwards, we propose several test criteria to select test cases from an OrBAC specification. We also consider two strategies to produce efficient access control test cases w.r.t. the criteria.

3.3.1 Access control test cases

The following example shows the access control test case for the rule `prohibition(borrower, return_book, maintenanceDay)` In this case the role is `borrower`, the activity is `return`, the view is `book`, and the context is `maintenanceDay`. The code contains a preamble that puts the system into a desired state (the book was previously borrowed by that user) before the execution of the actual test, and the evaluation of the access control test (the oracle).

```
// test data initialization
// log in a student
std1 = userService.logUser("login1", "pwd1");
// create a book
book1 = new Book(“book title”);
// book needs to be borrowed before returned
borrowBookForStudent(std1,book1);
// context
contextManager.setTemporalContext(maintenanceDay);
// run test
try { returnBookForStudent(std1,book1);
// security oracle
// SecurityPolicyViolationException is expected because an access control rule was not applied - test failure
 fail(“ SecurityPolicyViolationException expected,   returnBookForStudent with student = “ + std1 + “ and book = “ + book1); }
catch( SecurityPolicyViolationException e) {
 // ok security test succeeded log info
 log.info(“test success for rule :   prohibition(borrower,return_book,maintenanceDay)”);
```

3.3.2 Test criteria

In addition to functional tests, three security tests criteria, used to select access control test cases (for testing OrBAC policies) are compared:

- *All-primary-rules criterion* - is satisfied if and only if 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.

- *All-concrete-rules criterion* (Concrete) is satisfied if and only if a test case is generated for each implemented 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 all-concrete-rules criterion is stronger than the all-primary-rules criterion, since it forces the coverage of all concrete rules.
- *Robustness criterion* – used to select *Robustness access control test cases* that exercise the default/non specified aspects of the access control policy. These test cases are selected to kill mutants generated with a specific mutation operator (ANR) that will be presented in the next section.

Security test cases obtained with all-concrete-rules or all-primary-rules should test aspects which are not the explicit objectives of functional tests. For instance functional tests do not activate all security mechanisms related to the prohibition rules.

3.3.3 Test strategies

We study whether the functional test cases can be used for access control testing. Reusing functional test cases implies adapting them for explicitly testing the access control policy. The intent of the functional test becomes *security* and details the access control 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 by the PDP. This allows detecting hidden mechanism. For example if the PDP is bypassed due to hidden mechanisms, this is the only way to detect that permission has been actually granted by the PDP. In addition, the test checks that authorization hook placement is correct. In other terms, it is possible to check that the PEP did really call the PDP for granting or denying the authorization.

So, we consider two 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 involves 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 involves selecting functional, access control test cases and advanced access control test cases independently.

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

3.3.4 Test Qualification

Figure 14 shows the process for improving an access control test cases suite so that it reaches a satisfying level, in terms of ability of detecting security flaws. When flaws have been corrected by initial test cases, the test qualification loop involves 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 access control policy 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 an access control policy if it kills all the security mutants. The improvement process ends when the 100% mutation score is reached. The improved access control test cases can then be reapplied on the non-mutated code to check whether actual flaws are detected.

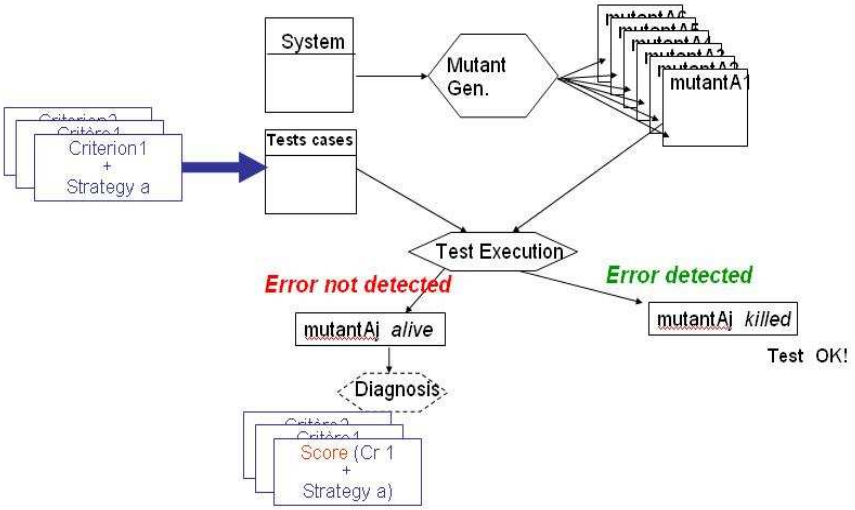


Figure 14 - Qualifying security tests

In next section, we explain in details how mutation is applied to security tests qualification, and especially to compare access control test selection criteria.

3.4 Mutation analysis applied to Access control testing

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.

In this section, we discuss the issue of adapting mutation analysis to access control testing. We propose adapted mutation operators and discuss their meaning. We also discuss the effectiveness of mutation analysis applied to access control policy testing.

3.4.1 Defining operators for access control policies

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 access control policy (and the verdict difficult to define).

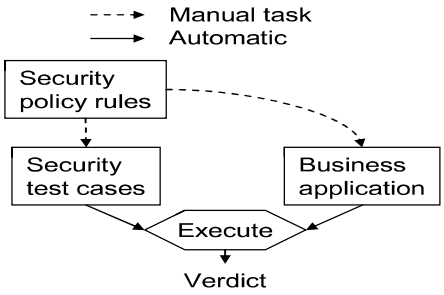


Figure 15 - Testing the access control policy

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 based 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 involves transforming mutation operators defined with OrBAC into faults in the implementation.

For this work, we assume that OrBAC access control policy is encapsulated in the PDP (which is the usual expected architecture). This way, it is easy to automatically generate the mutants, by modifying the policy that is stored in the PDP. In some cases this generation may be difficult and needs to be done manually. In that case, for 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 one of the objectives of the case study, which is to select a sufficient subset of mutation operators.

Next, we present a set of mutation operators that for OrBAC rules. We illustrate these operators with examples.

The security mutation operators are divided in 4 categories:

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

3.4.2 Type changing operators

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

- PRP: Prohibition to permission
- PPR: Permission to prohibition

These operators simulate faults that occur during the implementation. The obtained policy allows a forbidden activity or prohibits an authorized one. We show some examples obtained by these operators:

Example 1: Rule used:

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

Rule to use instead:

```
Prohibition (Secretary, ConsultBorrowerAccount, BorrowerAccount, default)
```

Example 2: Rule used:

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

Rule to use instead:

```
Permission(Student,GiveBackBook, Book, Holidays)
```

3.4.3 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:

- Change role
- Change activity
- Change view
- Change context

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

Rule to use instead:

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

`ReserveBook cannot replace ModifyAccount` because `ReserveBook` cannot be attached to the BorrowerAccount view. The same problem occurs 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:

Example 1: Rule used:
```
Permission (Administrator,ModifyAccount, BorrowerAccount, default)
```
Rule to use instead:
```
Permission (Secretary,ModifyAccount, BorrowerAccount, default)
```
Example 2: Rule used:
```
Permission (Student, BorrowBook, BorrowerAccount, WorkingDays)
```
Rule to use instead:
```
Permission (Student, BorrowBook, BorrowerAccount, Holidays)
```

These 2 operators are useful because they will simulate cases where the access control policy is too permissive or too restrictive. The first example allows a user (the secretary) to do the same activity allowed for another user. The second example 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).

3.4.4 Hierarchy-following parameter changing operators

A mutation operator can be used to change hierarchies by replacing a parameter by its parent or one of its descendants.

- Change role hierarchies
- Change activity hierarchies
- Change view hierarchies
- Change context hierarchies

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. The following examples show the impact how this mutation operator. In the first example, the borrower role is replaced with student role which limits the scope of the rule to students and denying access to the other hierarchy-following parameters (the teachers). While the second example, it is the activity that is involved which leads to allows only borrowing books and prohibiting the other activities (reserving and returning books).

Example 1:
Change role hierarchies, rule used:
```
Permission (Borrower,reserveBook, Book,WorkingDays)
```
Rule to use instead:
```
Permission (Teacher,reserveBook, Book,WorkingDays)
```
Example 2:
Change activity hierarchies, rule used:
```
Permission (Student,BorrowerActivity, Book,WorkingDays)
```
Rule to use instead:
```
Permission (Student,BorrowBook, Book,WorkingDays.
```

The hierarchy-following parameter changing operators that will be retained are: ‘change role hierarchies’ (named RPD) and ‘change activity hierarchies’ (named APD).

3.4.5 Add rule operators

Instead of replacing an existing rule, the adding rule operator introduces a new rule. When creating a new rule the compatibility between the selected view and activity is enforced to guarantee the semantic correctness of the generated rule. The goal of this operator is to simulate cases where the implementation does something in addition to the requirements. This is a typical security fault which 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 access control policy. It is important to note that this operator generates a lot of mutants.

Examples of added rules

```
Permission(Teacher,consultPersonnelAccount,PersonnelAccount,MaintenanceDay)
Permission (Secretary,ManageAccess , PersonnelAccount,MaintenanceDay)
```

Next table presents the retained operators:

Table 1 - Retained Mutation operators for OrBAC rules

Mutation Operator	Description
PRP	Prohibition rule replaced with permission
PPR	Permission rule replaced with prohibition
RRD	Role is replaced with different role
CRD	Context is replaced with different context
RPD	Role (a parent) replaced with one of its descendants
APD	Activity replaced with one of its descendants
ANR	Add new rule

3.4.6 The effectiveness of mutation analysis

The mutation process we propose simulates errors in the access control policy. Injecting errors in the policy seems counter-intuitive, since it is unlikely to find errors in the original policy, this one being automatically included in the PDP. In fact, the idea is that if there is some discrepancy between the access control policy and the functional/business application, it is due to wrong interactions between the security mechanisms and the functional code. Errors can be located in the mapping between the functional concepts and the security concepts or in the PEPs, since the calls to the PDP are implemented manually.

It is interesting to highlight the fact that the policy mutants imply faults on both the PEP and in the mapping. We illustrate this in the following example with the LMS policy:

PPR mutant:Replace Student with Personnel in R4

POLICY LMS-RDD-R4-Student-Personnel

Ri -> Permission(Library, **Personnel,** Borrow, Book, WorkingDays)

Mapping errors: RoleMapping(RoleClass, RoleEntity)

Instead of RoleMapping(StudentClass Student) => RoleMapping(StudentClass Personnel)

Detection of a “mapping error”: This example illustrates how mutation in the policy impacts the mapping. In this case, the error in the mapping is easy to locate and many mutants can help finding it. In the example, the policy combined with this error in the mapping makes it possible for students and personnel to borrow books. After testing, if this mutant remains alive, this means that there is an error in the mapping of the student role. Therefore, this mutant allows us detecting this mapping error.
Detection of a PEP-to-PDP interaction error: Mutation allows detecting erroneous interactions between PEPs and PDP. If a call to a particular rule is not correctly enforced, the mutant corresponding to this rule will not be killed, allowing by the way the erroneous call to be detected.
Test cases, which detect mutant policies, necessarily exercise all the security mechanisms (PEP), and may even detect some hidden security mechanisms as shown in [11]. However, our approach does not allow us detecting all possible hidden accesses that serve as a backdoor. In this case, there is an undocumented code that allows access to a certain user or in a certain condition. Mutating the policy will not help detecting such a backdoor. The only way to detect these backdoors is through monitoring the behavior of the application and checking how access is granted to users.

3.5 Empirical studies and results

The case study applications have a typical 3-tiers architecture widely used for web applications. We used 3 applications:

- LMS: a library management system presented in section 2.
- VMS: The virtual meeting system offers simplified web conference services. It is used in an advanced software engineering course at University of Rennes 1. The virtual meeting server allows the organization of work meetings on a distributed platform. When connected to the server, a user can enter or exit a meeting, ask to speak, eventually speak, or plan new meetings. Each meeting has a manager. The manager is the person who has planned the meeting and has set its main parameters (such as its name, its agenda, etc). Each meeting may also have a moderator, appointed by the meeting manager. The moderator gives the floor to a participant who has asked to speak.
- ASMS (Auction Sale Management System): The ASMS system 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.

Table 2 gives some information about the size of the 3 applications (the number of classes, methods and lines of code), and the number of access control rules for each system.

Table 2 - The size of the three case study applications

	# classes	# methods	# LOC	# rules
LMS	62	335	3204	41
VMS	134	581	6077	106
ASMS	122	797	10703	130

3.5.1 Generated mutants

Table 3 shows the number of generated mutants per operator for the three applications. The ANR operator generates many mutants since it adds a non-specified security rule. This number may vary, depending on the completeness of access control rules, as shown in the table. The fewer rules are specified, the more mutants this operator generates. The number of

ANR mutants thus reflects that all possible cases have not been specified in the access control policy. To our experience, this is quite usual: the specification focuses on the most critical and important cases, and often considers that default behavior is acceptable. Testing these cases forces the default policy to be exercised and highlights lacks in the specification. On the other hand, there are few mutants generated from hierarchy-following parameter changing operators because the specifications do 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 3 - # Mutants per mutation type and operator

Operator category		Op.	LMS	ASMS	VMS
Basic Mutation operators	Type changing	PPR	22	89	36
		PRP	19	41	70
	Parameter changing	RRD	60	650	530
		CRD	60	520	318
	Hierarchy-following parameter changing	RPD	5	20	20
		APD	5	0	20
Rule adding operator		ANR	200	736	432
Total			**371**	**2056**	**1426**

3.5.2 Issue 1: functional tests for validating the security policy

The first experiments aim at studying whether the functional test cases can actually reveal errors in the access control mechanisms. While the functional tests, which cover 100% of the code, necessarily execute access control mechanisms, they focus on the functional success/failure of test sequence executions. When reusing functional test cases with the objective of testing security, the intent changes and 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 that it should exercise.

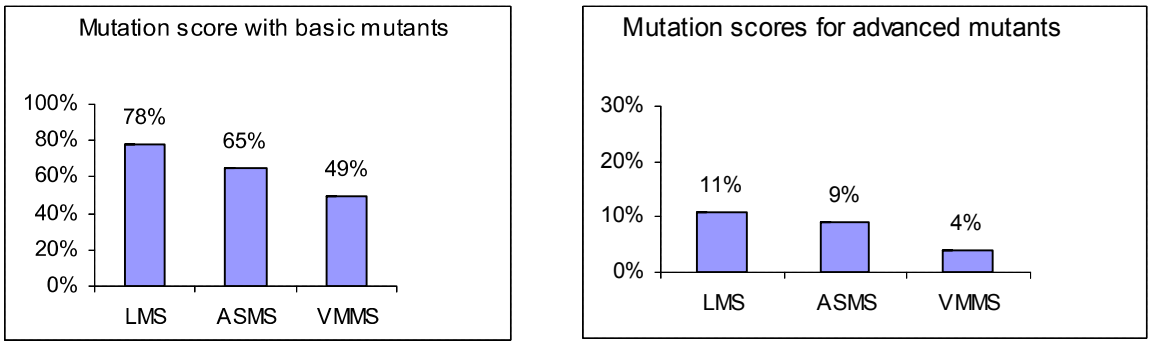


Figure 16 - Mutation scores for functional tests

Figure 16 shows that the functional test cases, adapted to security, can kill many (78%, 65% and 49%), but not all, basic mutants (produced by basic mutation operators). We notice that the functional test cases are not efficient (only 11, 9 and 11% are killed) for killing advanced mutants. This proves that these tests are not adapted for testing the robustness of the security mechanisms. This limitation is actually due to the fact that the ANR operator generates security flaws that are outside the scope of the specification. In conclusion, 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 security mechanism.

3.5.3 Issue 2: Comparing security test criteria

Figure 17 shows the mutation result for All-primary-rules and All-concrete-rules security tests. Only all-concrete-rules criterion allows reaching 100% mutation score. However, all-concrete-rules require more tests as showed in Table 5.

In addition, we notice that the two criterion cover a few number of the ANR mutants (only 17%, 16% and 32% at best). 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.

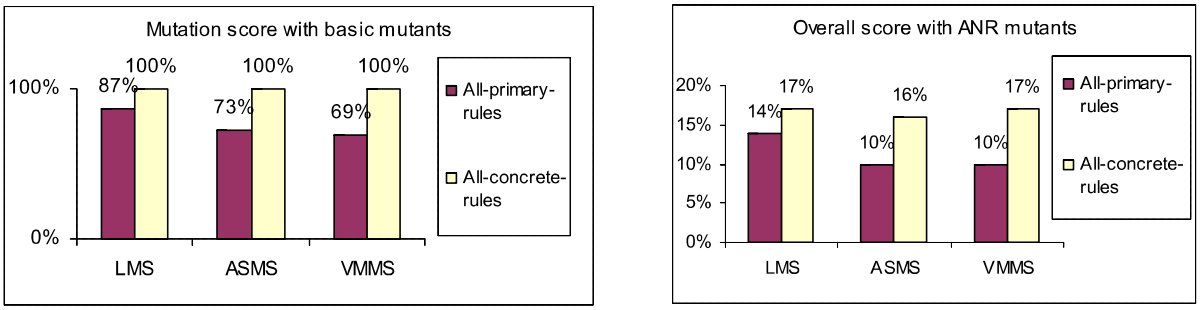


Figure 17 - Mutation scores for All-primary-rules and All-concrete-rules security tests

The CR2 criterion is necessary to provide a full coverage of basic mutants, and. 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), we will propose in section C an empirical ranking of the mutation operators to focus only on the hard-to-kill mutants.

Table 4 - Number of tests cases by category

	LMS	ASMS	VMS
All-primary-rules	20	39	29
All-Concrete-rules	35	110	106

3.5.4 Issue 3: Robustness vs. basic security tests

The *robustness security test cases* are explicitly generated in order to kill all the ANR mutants. In fact, the robustness test cases are generated by testing 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 access control policy.

For example, if the following rule is not defined:

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

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

When generating robustness security test cases, the 100% mutation score with ANR

mutants is used as a test criterion. 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.

The issue here is to compare the test cases selected using the all-concrete-rules criterion and the robustness security tests which are generated in order to kill all the ANR mutants. Table 5 presents the overlap between these two approaches. It is interesting to note that the robustness security test cases kill up to (59, 69 and 72%), of the basic security mutants. On the other hand, the test cases selected with all-concrete-rules kill up to 17, 16 and 32% of the ANR mutants. The effort to kill the ANR mutants is much more important (154, 614 and 384 test cases) than for killing the basic security mutants (35, 110, 106). Tests based on the all-concrete-rules criterion and robustness security test cases are thus not comparable, and are both recommended, the first is used to efficiently test the specification and the second to cover non-specified cases (robustness, default access control policy).

In conclusion, the robustness security tests cannot replace all the tests based on all-concrete-rules.

Table 5 - Overlap of CR2 and adv. test cases

		#test cases	Basic security mutants	ANR
LMS	**All concrete rules**	35	100%	**17%**
	Robustness Tests	154	**59%**	100%
ASMS	**All concrete rules**	110	100%	**16%**
	Robustness tests	614	**69%**	100%
VMS	**All concrete rules**	106	100%	**32%**
	Robustness tests	384	**72%**	100%

3.5.5 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.

Figure 8 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.

It appears, as highlighted in Figure 18, that the independent generation of basic security and

advanced test cases is the most costly compared to incremental strategies (note that for incremental strategies the functional test cases are not counted). Reusing functional test cases and completing them is the most interesting strategy, being stated that the cost of adaptation of functional test cases cannot be estimated. The *incr. from CR2* strategy is an interesting compromise, since the functional test cases are not reused, but the robustness test cases are added to complete the CR2 test cases. With this incremental strategy, the saved testing effort varies between 11% (LMS) and 25% (VMS) compared to an independent test generation. It has to be noticed that, depending on the expected security level of trust, a decision has to be taken concerning the need for testing the access control robustness (by killing ANR mutants), given the additional cost that is required.

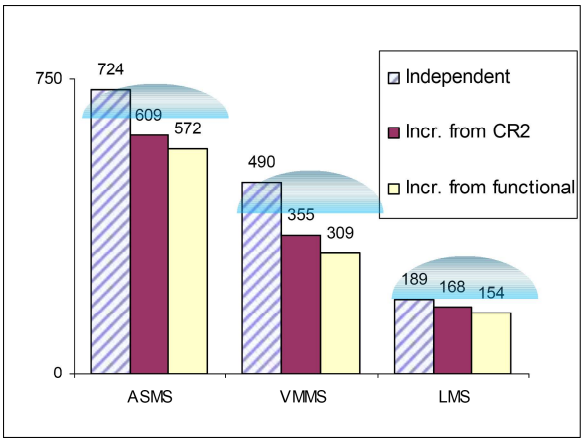


Figure 18 - Independent vs. Incremental strategies

3.5.6 Issue 4: ranking mutation operators

Our objective is to analyze the interest of each mutation operator. As for the principles of selective mutation [128], 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 empirical results. The generation of mutant security mechanisms is fully automated, allowing the presented experimental protocol to be applied. The *first step* involves 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 if the test cases it includes have a 100% mutation score and, if when a test case is removed, the mutation score decreases.

It has to be noticed that this notion of *minimality* has to be distinguished from *optimality*. A test suite is optimal if it is minimal and no other minimal test suite exists which is smaller. The optimal solution is often not known. 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* involves 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 and only 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 19 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.

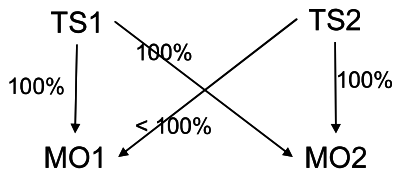


Figure 19 - MO1 operator strictly subsumes MO2

Definition: mutation operators ranking (>).

MO1 > MO2 if and only if:
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.

Table 6 - Sizes of minimal test suites

Operator category	LMS	ASMS	VMS
Rule type changing operators (PPR-PRP)	36	110	106
Rule parameter changing operators (RRD-CRD)	36	110	106
Hierarchy-following parameter changing operators (RPD-APD)	4	17	10
Rule adding operator (ANR)	154	634	384

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-following parameter changing operator).

The results for the size of the minimal test suites are displayed in Table 6. The relationships between minimal test suites are the following:

TS(RRD-CRD) = TS(PPR-PRP)

TS(RPD-APD) $\subset$ TS(PPR-PRP)

| TS(PPR-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.

Table 7 - Test suites overlap for PPR-PRP and ANR operators

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

The ANR and PPR-PRP operators are not comparable with this ranking. Some test cases are shared by both minimal test suites (21 test cases). Table 7 shows the overlap between the test suites, in terms of respective mutation scores for the LMS. 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.

Operator used: RRD
Initial rule: Prohibition(Secretary,ManageAccess, PersonnelAccount,Default)

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

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

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 tests are already generated to kill mutants without conflicts.

Figure 20 - Testing mutants causing conflicts.

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 because there are two test cases able to detect this mutant. By giving priority to mutant rule, an additional error is injected by not applying the conflicting rule. Table 8 presents the number of mutants which generated conflicts per operator for LMS. Figure 20 shows an example of such a conflict. It illustrates that two test cases, generated to kill mutants without conflicts, kill this mutant with conflict.

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) correspond 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 does 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)
- 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.

Table 8 - # of generated mutants with and without conflicts

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

In fact, these mutants that are with conflicts could be removed and the explanation is interesting, since it corresponds, for access control testing, to the 'coupling effect' analyzed by Offutt et al. [86]. 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.

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
2. Rule type changing operators
3. Rule parameter changing operators
4. Hierarchy-following parameter changing operators

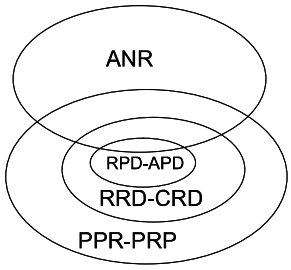


Figure 21 - Relation between mutation operators

Figure 21 displays the operators ranking, and highlights 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.

3.6 Conclusion

We have identified a set of issues related to access control testing and have showed the specificities of this testing task through several experiments. In particular, we have illustrated how functional and security testing can be tackled as complementary activities.

We propose a methodology for test cases selection, with various test adequacy criteria based on the access control rules or on mutation. It also distinguishes test selection for testing the "nominal" access control rules from the advanced test selection that aims at testing the robustness of the access control mechanisms.

The case studies highlight the issues a test expert has to deal with when facing the objective of testing an access control policy for a real system. In particular, two qualitative aspects arise: the possibility, to adapt functional test cases to test access control mechanisms, and the interest of advanced security tests, regarding the important additional effort it requires. More fundamentally, the aspect of security mechanism testability in relation to system architecture is critical. The way security mechanisms are distributed 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. We believe that the testability of the system tightly linked to the way it was designed and deployed.

In the next chapter, we propose new approaches aiming at automating:

- The process of testing: As shown in this chapter, functional test can be reused to test security mechanism. An interesting issue involves the automation of this process, which means the automated selection of test target. Then, in order to automate the process of generation, we propose to select the subset of functional testing to be automatically transformed into security tests.
- Detecting the hidden mechanisms: These hidden mechanisms harm the flexibility and make the system rigid to the evolution of the access control policy. We use mutation to automatically detect these hidden mechanisms.

Chapter 4
Testing strategies for access control policies in applications

In this chapter, methodologies that target the process of testing access control policies will be presented. Three complementary testing approaches are introduced. The first two tackle the issue of test generation, while the third one deals with detecting the hidden mechanisms. We conducted an empirical study to evaluate the effectiveness of these approaches.
The first work aims at proposing solutions to automatically generate security testing. This objective is reached through two different methodologies.
The first one is a *bottom-up* approach that is a corollary of the previous chapter findings. In the previous chapter we showed that the functional tests execute some of the security mechanisms and hence activate and test some access control rules. The idea involves selecting and transforming these functional testing in order to be used to test security. To do this, we use mutation to compute which rules are activated by the functional tests in order to select them and adapt them by completing their oracle functional on the fly. This is a bottom-up approach since in starts from the functional test to actually end up with security tests.
The second work proposes a model-based approach for generating *test targets*. Test targets are the set of rules to be tested. We use pair-wise testing to automatically generate these test targets. Then these test targets are used to automatically generate security test cases using code templates. To assess the quality of test targets we use mutation. In fact, we use pair-wise and there are several strategies for generating test targets. The effectiveness of these strategies is compared using mutation. This is a *top-down* approach because it starts from the access control model to generate (automatically) security tests.
While top-down and bottom-up strategies deal with test generation for a given version, they do not help with the issue testing policy co-evolution and more importantly detecting hidden mechanisms. The third approach aims as tackling the issue of hidden mechanism. These hidden mechanisms cannot be detected by the two previous approaches. Thus, we propose a new approach that instruments mutation for detecting these hidden mechanisms.

4.1 The bottom-up approach: testing security with functional tests

In the previous chapter, we showed the importance of the coupling between the security and the functional aspects. This coupling results in having functional tests triggering some of the security mechanisms. However, these functional tests do not include a security oracle, which means that they do not perform the suitable verifications of the security mechanisms. This limitation implies the need to adapt these tests by adding the security oracle. This process includes two steps; selecting the subset of functional tests that actually trigger the security mechanisms and adding the appropriate oracle. In order to do this task, we propose an automated approach that allows selecting the subset of functional tests, finding which rules are tested by each test and adding the specific security oracle.
In this section, we start with presenting the overall process. Afterwards, we detail the different steps. Finally, we show some empirical results and discuss the effectiveness of reusing functional tests.

4.1.1 Overview of the approach

Figure 22 presents the methodology. Firstly, we identify the test cases which are not impacted by the security policy and remove them from the initial set of test cases. Secondly, we analyze the impacted test cases in order to precisely associate the access control 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 policy enforcement point (PEP) is executed and that the access is granted or denied consistently with the security policy.
In the first two steps, the analysis is performed dynamically. Firstly, 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 mutation analysis which requires executing the test cases for each mutant. We systematically inject errors in every security rule in order to create mutant policy decision points (PDPs). This analysis is a kind of sensitivity analysis [129], and allows locating precisely the relationships linking a test case to access control rules in the PDP. If the behavior is the same when executing the test case on the initial and modified PDP, then this means that this test case 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 access control rules that are covered by this test case.

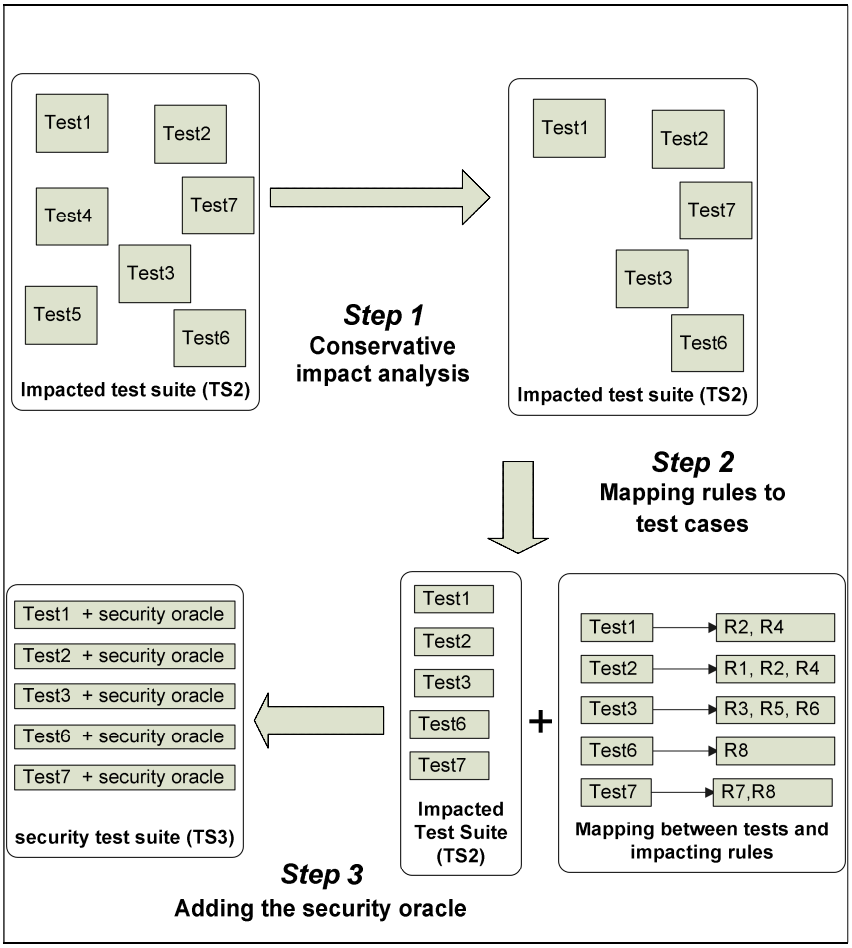


Figure 22 - Overview of the three analysis steps

We assume that the PDP is *controllable*. Controllability means that the rules in the PDP can be easily modified. This property allows modifying the rules and checking the sensitivity of the test cases execution to changes in the PDP. The access control rules can be implemented in several ways, for instance in a database or in XACML files.

4.1.2 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 access control policy*. One test case is considered to be impacted by the policy 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.

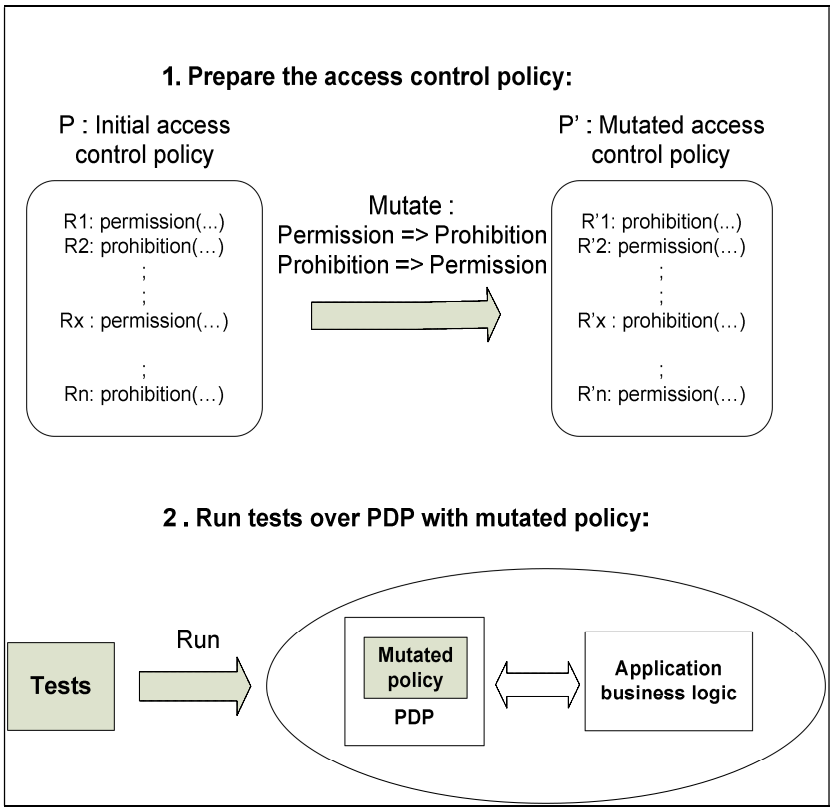


Figure 23 - First step: test cases selection

As shown in Figure 23, 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 one as well. We end up with a mutated access control policy that contains only faulty rules (a negative access control policy). In the PDP, we use this policy instead of the initial one. Then tests are run on the system using this new mutated policy. Tests that exercise functionalities protected by the security mechanism are expected to fail due to the errors injected in the policy. These tests are selected and considered as impacted by the security policy because they fail when 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.

4.1.3 Relating test cases and security rules

To perform this second dynamic analysis step, we applied mutation adapted to access control policies. In this section, we present in details the approach, which involves finding which rules are impacting which tests.

The objective is to map test cases to the access control rules they trigger. This second is used to build these relations. While it could be done with other techniques (e.g. by comparing the execution traces, regression testing techniques), we decide to use the mutation approach to

make this analysis possible. It has the advantage of being non intrusive in the business logic. No assumptions are needed concerning the PEP "observability" and the way the security mechanisms are implemented 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 a "sensitivity analysis" [129] by modifying the PDP access control rules and checking whether the execution of a test case is impacted by that 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 [115] or even manually if the number of rules is small.
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 terms, it means that the initial execution triggers the PEP related to the access control rule (recall that a PEP is a security mechanism which explicitly calls the PDP). This test is thus impacted by this particular access control rule.

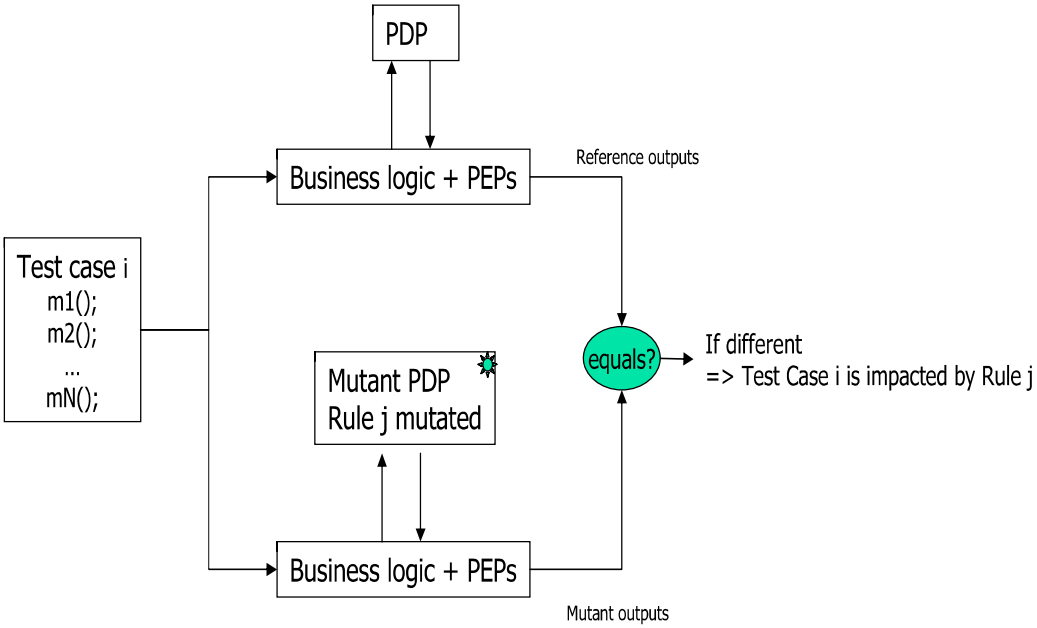


Figure 24 - Second step: dynamic analysis of test cases impact on security mechanisms

When a test case TC is executed, the PEP sends a request to the PDP that evaluates an access control rule R_j which impacts the behavior of the test case: using mutation, we can determine that rule R_j is evaluated by the PDP when the test case is executed.
For detecting the impacted test cases by an access control rule, we use PPR (prohibition to permission) and PRP (permission to prohibition) mutation operators. The ANR (Add New Rule) operator is not used either since existing functional test cases are unlikely to test the robustness of the security mechanism 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 to advanced security test cases 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 is mutated.

4.1.4 Oracle modification in the functional test case

We propose three different levels of quality to implement the oracle function. As stated in Table 9, the first level of oracle just checks that no obvious inconsistency exists between the current access control policy and the implementation of the system. Level 1 oracle function is "black-box", which means that 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 9 - The three quality level of oracle

Oracle level	**Assumptions**	**Description**
1	no (black box)	Check that an exception is raised (for prohibition rules) or that no exception is raised (for permission rules)
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)	- (2) - 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 more advanced checking of the PDP. For instance, it allows checking that the PDP functions are called as expected (with the right parameters).
Level 3 oracle extends level 2 and checks that no unexpected interactions occur between the PEP which is expected to send a request to the PDP. Level 3 oracle is thus a good way to detect interaction faults.
We automated the weaving of level 2 oracle functions in the impacted test cases using AspectJ. To make this process possible, the access control 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 (aspect oriented programming), 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 access control policy (specification). Figure 25 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).

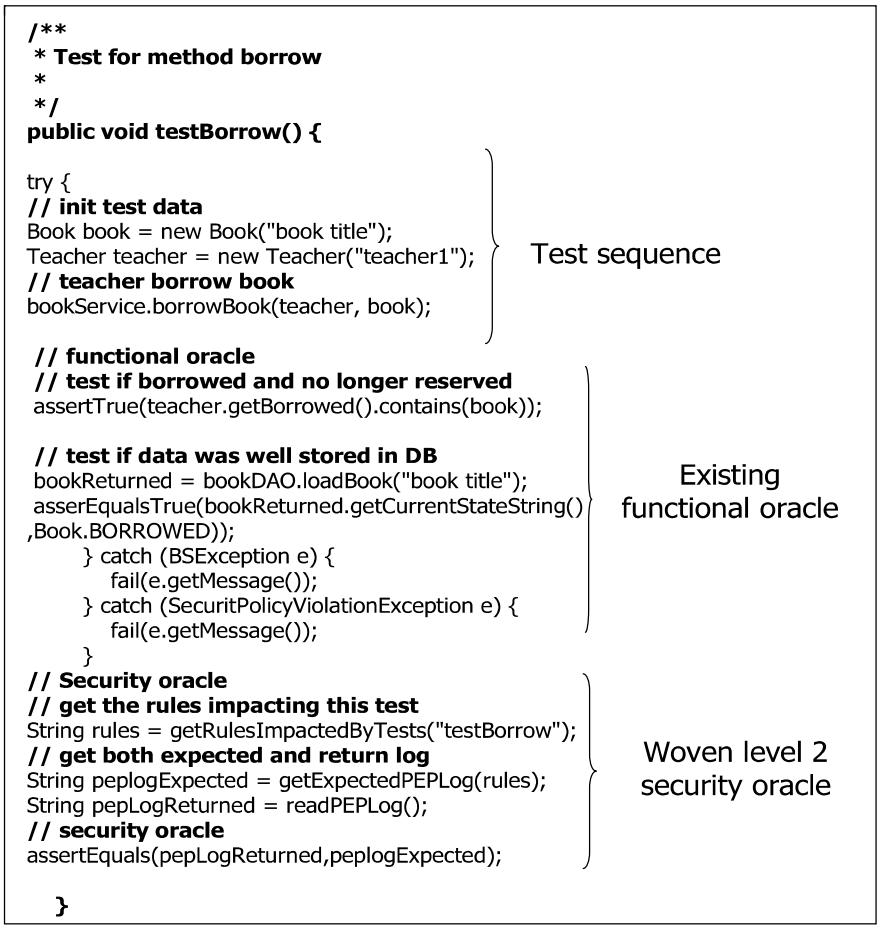


Figure 25 - Woven security oracle function

4.1.5 Proof of concept and empirical study

In this section, we show the various steps of the process and results obtained after applying the approach on the 3 case studies (presented in the previous chapter).

First step: impact analysis results

Table 10 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 10 - 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 11, 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 11 - Execution time saved with the first analysis

Application	% Saved time
VMS	20.4 %
SMS	34.6 %
LMS	15 %

Second step: test impact analysis results

Table 12 shows some examples of the results that we get using this analysis. For each test case, the analysis pinpoints the exact rules that are impacting it.

Table 12 - 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()	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 deeply the results and observe how many tests are impacted by only one rule or two (and so forth). Table 13 shows 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 13 - 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

4.1.6 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 security tests,

(2) Manually adapting existing tests (without step 1),

(3) Manually adapting 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 [126]. 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 14 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 it allows reducing significantly the work of analysis which is needed to determine manually which functional test is impacted by a security rule, the remaining task involves 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 is clearly high.

Table 14 - 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

4.1.7 Summary and conclusion

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 automated approach when compared to a manual one.
In the next section, a complementary approach will be presented. This top-down approach focuses on generating tests using a pair-wise technique.

4.2 Top-Down approach: Model-Based testing of access control policies

This approach aims at providing a model-based testing methodology targeting access control policy testing. We begin with describing the test methodology and define the main concepts. After giving the necessary background in terms of the considered access control and combinatorial testing, we present the testing methodology. Then, we present the experiments for assessing the quality of generated test suites. Finally, we discuss these results and conclude.

4.2.1 The considered access control model

As a running example, we re-use a library management system that was already introduced in the previous chapter. For this work, the model that will be used relies on hierarchical roles, hierarchical couples of activities and views (since each activity is associated to a view) and hierarchical contexts. Requests include a subject instance, a resource instance, an action instance that the subject wants to exercise on the resource and a context instance.
The application of actions to resources is requested by a subject in a given context. Intuitively, for a request, the policy decision point checks if there is a rule that matches the request: the subject instance of the request is part of the role defined in the rule; the (activity, view) pair is part of the rule, and the context instance is part of the context defined in the rule. In that case, the rule is said to be applicable to the request. If there is no such rule, the PDP returns a default decision that is either permission or prohibition. Finally, if there is more than one rule, then the rule with the higher priority is chosen. While our approach for generating tests obviously relies on the semantics of a particular access control model when the oracle is consulted, it can nonetheless be applied to any access control model.

4.2.2 The combinatorial testing

When testing configurations that involve multiple parameters, one fault model consists of assuming that failures are a consequence of the interaction of only two (three, four, etc.) rather than all parameters. Combinatorial testing [130, 131] relies on precisely this idea and aims at defining test suites such that for each pair (triple, quadruple, …) of parameters, all pairs (triples, quadruples, …) of values for these parameters appear in one test case in the test suite. If the assumption (the fault model) can be justified, the eminent benefit of this strategy lies in the rather small size of the test suite. For instance, without any further constraints, the number of tests necessary for pair-wise testing is the sum of the number of all possible parameter values in the system.

4.2.3 Adapting Pair-Wise test methodology

The problem to be tackled is the generation of a set of test cases for assessing in how far an application implements a given access control policy. We use this section to describe the test methodology and the technologies involved. There are two different ways of generating test targets; one regardless of any policy, the other depending on a policy.

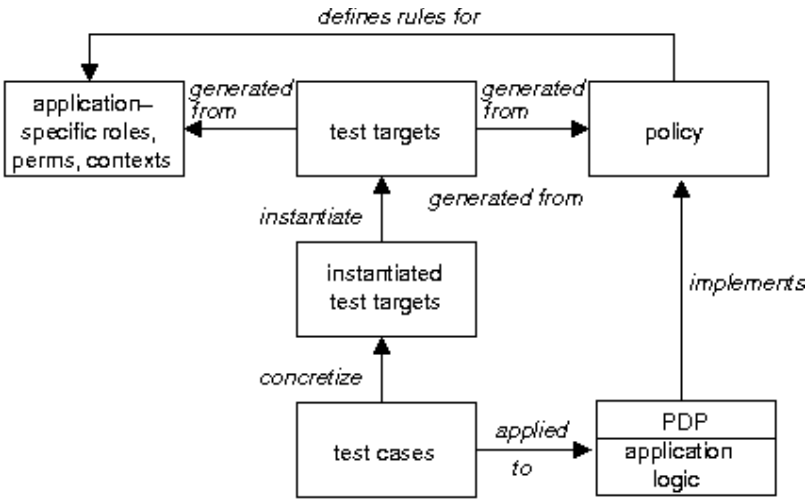


Figure 26 - The overall process

The process is depicted in Figure 26. Policies are defined for a domain that involves, among other things, application-specific *roles*, *activities*, *views* and *contexts*. Different means are employed to generate *test targets* from policies and partial domain descriptions. A test target includes a role name, an activity name, a view name and a context name. By picking a subject that is associated with the role name, a couple of activity and view permission that is associated with the activity name and view name, and a context that is associated with the context name, we obtain an *instantiated test target*. Test targets and instantiated test targets are defined at the level of abstraction of the policy and the partial domain description. They are hence too abstract to be directly executed on a real piece of software. Test cases, in contrast, are executed. These test cases implement the instantiated test targets and take into account the business logic of the application under test. The methodology that we define in this section is about the generation of (instantiated) test targets.
The generation of test targets can be done in at least two ways. One is to generate them regardless of any policy. A second strategy takes into account the policy.

Ignoring the Policy
Roughly, the *generation of test targets without policy* only takes into account information on roles, activities, views, contexts, and their hierarchies. In this case, the combinatorial testing is applied to all nodes of the three hierarchies as shown in Figure 27.

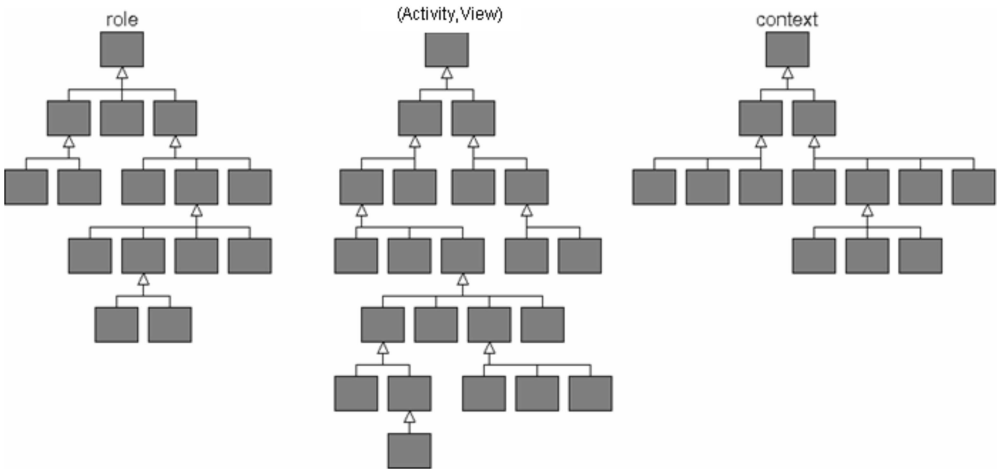


Figure 27 – The first generation strategy

Using the Policy

The *generation of test targets from access control policies*, in contrast, proceeds as follows. We will consider each single rule in turn. The part of the rule that is relevant for test case generation is the triple (r, (a,v), c) that includes a role name, a couple (activity,view), and a context name. Each element of that triple is a part of one of the three hierarchies, and they correspond to boxes with thick borders in Figure 28. Combinatorial testing is then employed to generate *n*-wise coverage:

- For all role names below r.
- With all (activity,view) names below (a,v).
- With all context names below c.

All these nodes depicted as grey boxes in Figure 28, top. Since a rule is either a permission or a prohibition, it appears sensible to also test all those nodes which are not explicitly specified (this complementary set is depicted as grey boxes in Figure 28, bottom).

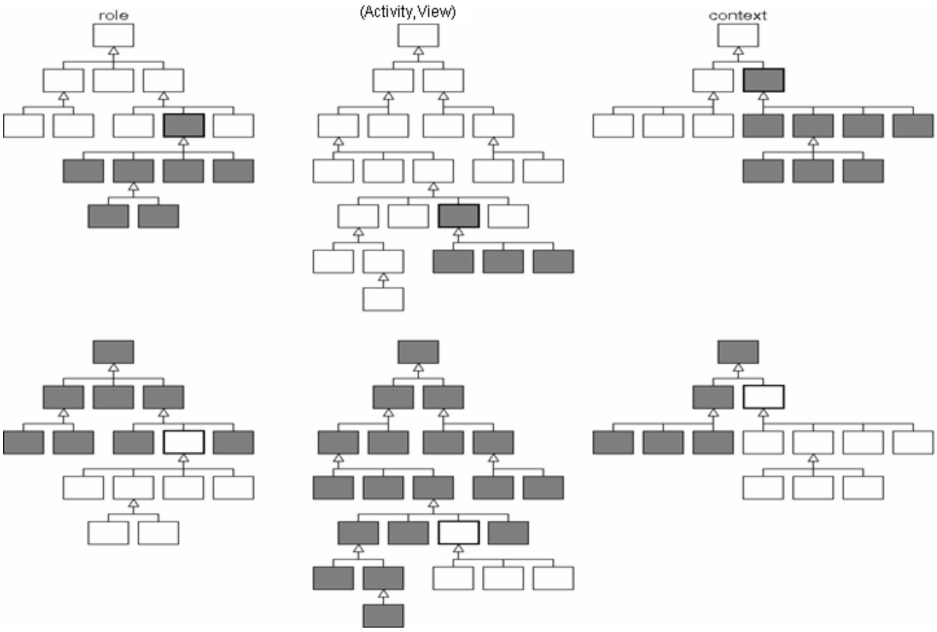


Figure 28 – The second generation strategy

For the three elements of a rule, we can then specify whether or not the respective component of a test target should conform to the element of a rule. As an example, consider a rule that constrains role r, (activity,view) (a,v), and context c (boxes with thick borders in Figure 29). The test targets that correspond to the grey boxes in that figure specify *a role different from r*, a *(activity,view) that derives from (a,v)*, and a *context that differs from c*. In sum, for each rule, this gives rise to eight combinations that can be used for the generation of test targets.

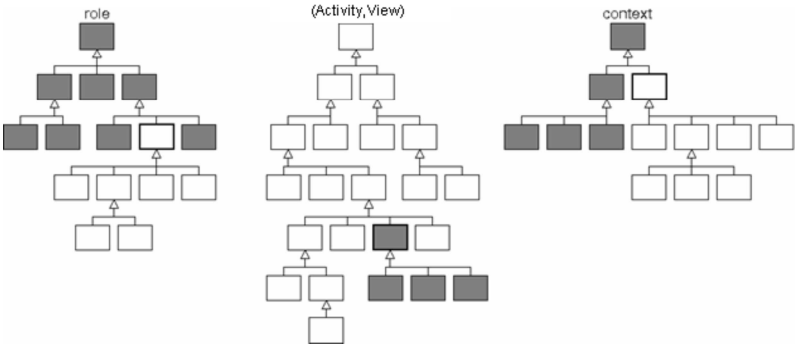


Figure 29 - The second generation strategy using different rule elements

Comparison

Not using the policy at all may appear counter-intuitive: if there is a policy, why not using it for test case generation? Not using a policy seems reasonable because regardless of the policy, *all kinds of requests* should be tested, and these only depend on the roles, activities, views and contexts, but not on the policies. The essential difference between the two strategies lies in the number of tests that correspond to existing rules. In most cases, some permissions or prohibitions will not be defined explicitly in the policy. Respective decisions are then taken by referring to the default rule. In the first approach, corresponding test targets may be generated more or less at random. In contrast, many test targets are explicitly generated by the second approach for these implicit rules.

4.2.4 Concrete Test Cases

Concrete tests differ from abstract test targets in that the latter do not take into account the application logic at all. The problem then is that specific actions cannot be applied to specific resources in all states. For instance, a book cannot be returned before it has been borrowed. When concretizing test targets, this information must be taken into account. The information, however, is likely available in the requirements documents, in the form of sequence diagrams or similar descriptions (for examples use cases, statecharts etc.). Then, the derivation of test cases involves writing a preamble that puts the system into a state where the access rule is applicable as far as the state of the application is concerned. The automation of this procedure is highly useful. This way, testing becomes a push-button technology.

The following example illustrates concretization. Consider the following rule for the LMS:

```
prohibition(borrower, return_book, maintenanceDay, 4)
```

For any of the above generation strategies, assume that the **test target** contains:

- A role `borrower`.
- A couple of activity and view `return_book`.
- A context `maintenanceDay`.

A possible **instantiated test target** is:

```
prohibition(std1, book1, maintenanceDay, 4).
```

However, this still is too abstract to be executed. Concretization leads the following code that contains:

- A preamble that puts the system into a desired state (by borrowing the book).
- Execution of the actual test.
- The oracle function.

```
// test data initialization
// log in a student
std1 = userService.logUser("login1", "pwd1");
// create a book
book1 = new Book("book title");
// activity
// book needs to be borrowed before returned
borrowBookForStudent(student1,book1);
// context
contextManager.setTemporalContext(maintenaceDay);
// security test
// run test
try {
 returnBookForStudent(std1,book1)
 // security oracle
 // SecurityPolicyViolationException is expected because an SP rules is not
respected
 // test failure
 fail(" SecurityPolicyViolationException expected,   returnBookForStudent
with student = " + std1 + " and book = " + book1);
}
catch( SecurityPolicyViolationException e) {
 // ok security test succeeded log info
 log.info("test success for rule :
prohibition(borrower,return_book,maintenanceDay)");
}
```

4.2.5 Implementation

With the exception of the generation of Java code, our test generation procedure is fully automated. The system takes as input a policy and the respective domain description a set of (role, (activity,view), context) together with the strategies to be applied. For each strategy, it returns the generated test targets. *N*-wise test generation is performed by a tool that is publicly available at www.burtleburtle.net/bob/math/jenny.html.
In terms of the experiments, mutants of the policy are generated using the mutation technique presented in the previous chapter.

4.2.6 Experiments

The objective of the experiments is to assess the quality of tests generated by the different strategies. We consider the three case studies used in this chapter (LSM, ASMS and VMS). In addition, we consider the access control policy in a **hospital** with different physicians and staff. It defines who manages the administrative tasks and who performs which medical tasks. Its policy is defined in 37 rules with 10 roles, 15 couples of (activity,view), and 3 contexts. In contrast to the other systems, the hospital system was not implemented in an application.

Experimental protocol
We start with generating tests with the two strategies. In addition to this, we generate tests purely at random. We count the number of generated test targets and measure the generation time. This time turns out to be negligible (less than a second). In a second step, we assess the quality of the test suites by using mutation testing.

Generation of test targets
The **first strategy** takes into account only the domains. Combinatorial testing is applied. In order to measure the influence of randomness when pairs are chosen, the generation process is repeated thirty times for each strategy and for each case study.

The **second strategy** does take policies into account. For each of the eight sub-strategies, we generate test targets for each single rule. We make sure that they are all independent from each other: we do not first generate two sets of role names, two sets of (activity,view) names and two sets of context names and then pick the 8 combinations from these buckets but rather regenerate them in each turn. This experiment is also performed thirty times (hence 240 experiments per rule). In each of the thirty experiments, we remove redundant test targets from each of the eight test suites.
The **third strategy** also does not consider policies. However, test targets are chosen fully randomly, without pair-wise testing. In this case too, we remove redundant tests. This generation strategy is also applied thirty times.

Assessment of tests
The quality assessment proceeds as follows (we only consider the level of test targets here). We consider each of the four policies in turn. We apply the first five mutation operators to every rule. The generated test suites are then applied to all mutants, and we record the mutation score.

Observations and Discussion
Figure 30 to Figure 33 show box-whisker diagrams of the mutation scores for the four case studies. Strategy 1 (no policy, only domain considered) is labeled 1-pol1-NNN, and 1-rnd-NNN displays the results of the respective random tests (strategy 3). NNN denotes the number of generated test targets both for strategy 1 and for the corresponding random experiment.

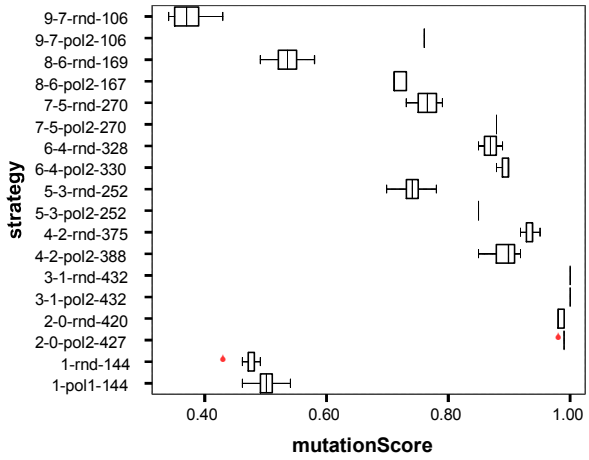


Figure 30 - Meeting System (432 exhaustive tests)

The eight sub-strategies of strategy 2 are labeled 2-0-pol2-NNN, 3-1-pol2-MMM, …, 9-7-pol2-PPP. The first digit indicates the strategy's global index. The last digits correspond to the number of test targets generated for that strategy. The second digit corresponds to the chosen sub-strategy: Let c denote whether, for every rule, contexts are chosen from the specializations of the provided context or from the complement of that set. Let (a,v) denote whether, for every rule, contexts are chosen from the specializations of the provided (activity,view) or from the complement of that set. Finally, let r denote whether, for every rule, contexts are chosen from the specializations of the provided role or from the complement of that set. The second digit then is the decimal representation of the binary number r(a,v)c. For instance, 010 (2 in decimal terms; Figure 29) corresponds to: (1) context

chosen from the sub-contexts of that provided in the rule, (2) (activity,view) from the set of all (activity,view) couples that are not below the (activity,view) provided in the rule, (3) role below the role that is specified in the rule. In the following, we will denote sub-strategy xyz (x,y,z are 1 or 0) of strategy 2 by 2.xyz.

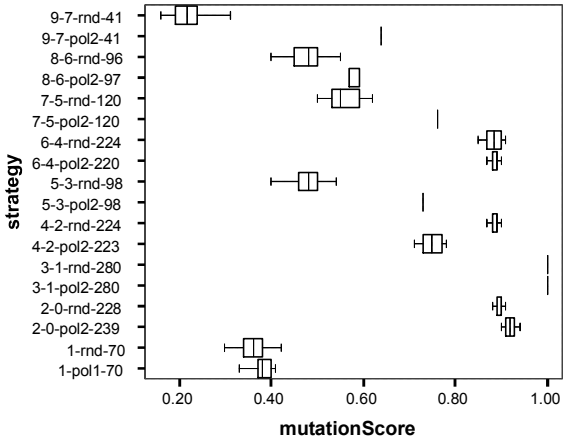


Figure 31 - Library System (280 exhaustive tests)

Overall, the variance of the mutation scores in all experiments is rather small. The number of tests for comparable strategies may slightly differ (e.g., for strategy 2.0 in all four experiments). This is a result of the influence of randomness on the pair-wise testing approach. However, as the variance is small, we will ignore this effect.

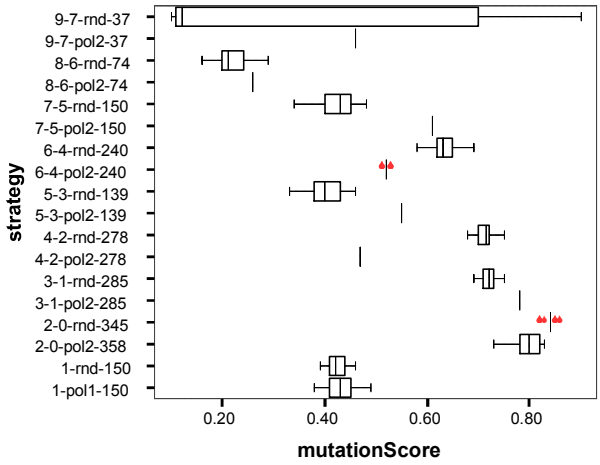


Figure 32 - Hospital System (450 exhaustive tests)

Furthermore, the eight sub-strategies of strategy 2 result in different numbers of test targets. This is a consequence of the role, (activity,view) and context hierarchies: the number of nodes above or below a node need not be identical. Consequently, pair-wise testing is applied to variables with different domains.

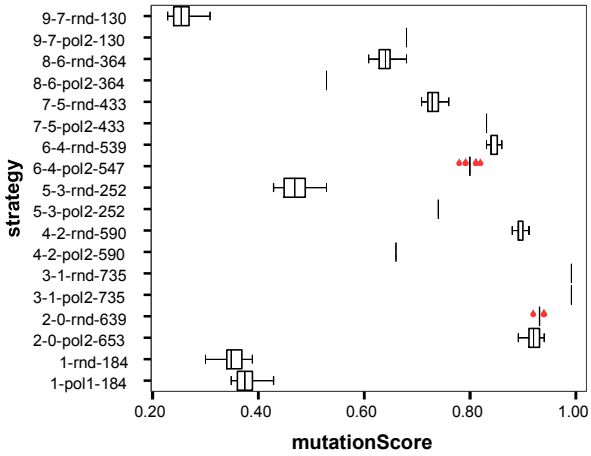


Figure 33 - Auction System (736 exhaustive tests)

Strategy 1 turns out to be as good as random testing, and, with the exception of the hospital system, worse than all sub-strategies of strategy 2. The number of tests is comparatively low: an average of 29% of the cardinality of the exhaustive test set. However, because of better strategies, pair-wise testing that does not take into account a policy is, in terms of our assessment criterion, a strategy that can safely be discarded. This result indicates that taking into account an access control policy when generating tests on the grounds of pair-wise testing is highly advisable. This is a result that we did not expect and that is probably due to the use of mutation testing for assessing tests.

With the exception of the hospital system, **strategy 2.1** yields exhaustive tests. This will, in general, not be the case for all policies. In our examples, however, hierarchies are rather flat (which means that the negation of roles and permissions yields, for every rule that is used for test target generation, almost all other roles and permissions, respectively). At the same time, the number of contexts is very low, which, taken together, leads to a high probability of generating exhaustive tests (note that this is only almost the case for the auction system). The exception of the hospital system is explained by its policy and the mutant generator: for one role, there are no rules, and in contrast to the test target generator, the mutation generator creates mutants only for those elements that occur in any rule (and that are not only provided in the policy's domain description).

When comparing different strategies to random tests, we get the following aggregated results (bold rows indicate superiority of the respective strategy):

Table 15 - Strategy 1 and 2 vs. the random strategy

Strategy	Better than random	Equal	Worse
1	0	4	0
2.0	1	2	1
2.1	1	3	0
2.2	0	0	4
2.3	**4**	**0**	**0**
2.4	0	2	2
2.5	**4**	**0**	**0**
2.6	2	1	1
2.7	**3**	**1**	**0**

As we can see in Table 15**, Strategies 2.3, 2.5, and 2.7** perform better than random tests**, strategies 2.0, 2.1, and 2.6** perform similarly to random testing and **strategies 2.2 and 2.4** perform worse than random testing.
However, we also have to take into account the number of tests that achieve these results. Table 16 shows the relative number (x 100) of test cases that achieved the mutation scores, i.e., number of tests divided by cardinality of exhaustive test set. We also show the averages.

Table 16 - The percentage of generated test of strategy 2

2.0	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**2.7**	**System**
99	100	90	58	76	63	39	25	Meeting
85	100	80	35	79	43	34	15	Library
80	63	62	31	53	33	16	8	Hospital
89	100	80	34	74	59	50	18	Auction
88	**91**	**78**	**40**	**71**	**50**	**35**	**17**	**Average**

Strategies that take into account positive context definitions (i.e., do not make use of the complement set: strategies 2.1, 2.3, 2.5, 2.7) provide better or identical results than random tests. Out of these, 2.3 and 2.5 perform their results with 40-50% of the tests, and 2.7 with only 17%.
Conversely, strategies that take into account the complement set of the contexts (strategies 2.0, 2.2, 2.4, 2.6) provide worse results than random tests. Three of them require 71-88% of the tests; only strategy 2.6 requires a mere 35%.
Strategies that do not negate roles and (activity,view) couples at the same time (strategies 2.2 and 2.4) perform worse than random tests. They do this with a comparably high number of tests: 71-78%.
With the exception of strategy 2.7, strategies that either negate or do not negate roles and (activity, view) couples perform as good as random tests (2.0, 2.1, 2.6). Strategies 2.0 and 2.1 require 88%-91% of the tests while strategy 2.6 (that may also be classified as performing better than random tests) only requires 35% of the tests.
In terms of the number of necessary tests that tends to be relatively high when compared to the exhaustive test set, strategies 2.3, 2.5, 2.6 and 2.7 appear promising. In terms of the mutation scores, 2.1, 2.3, 2.5 and 2.7 appear promising. The intersection consists of **strategies 2.3, 2.5 and 2.7**.
When compared to random testing, **strategy 2.7** (positive role, positive (activity,view), positive context) performs particularly well – good mutation scores are obtained for a rather small number of tests (17%). The number of test targets for strategy 2.7 equals the number of rules. The reason for this equivalence are the rather flat hierarchies in all example systems, and as it turns out, if a rule is defined for a non-leaf node of a hierarchy, then there are always complementary rules for all sub-nodes. Since redundant tests are removed in all strategies, this explains that exactly the number of rules is obtained. This result suggests that simply using one test per rule (possibly with exactly the elements that define the rule) provide good results.

4.2.7 Summary and conclusion

Because of the many degrees of freedom in the policy language of our examples (default rule, specification of both permissions and prohibitions, priorities), we think it is too early to draw generalized results in terms of which strategy is better. We also conjecture that this depends on the policies ratio of permissions and prohibitions, and on the depth of the different hierarchies. However, we believe that our work very clearly suggests that:

- Using policies for test generation with pair-wise algorithms is preferable to only using domain knowledge (roles, (activity,view) values, and contexts);
- While it may be too early to decide on the best strategies, there are notable differences between the different strategies that use combinatorial testing. This suggests that research into combinatorial testing for access control policies is a promising avenue of research;
- Our generation procedure that uses pair-wise testing is stable in the sense that it is not subject to random influences, as suggested by the small variances obtained in a thirty-fold repetition of our experiments.

Our conclusions are of course subject to several validity threats. The main problem with any generalization is obviously the small number of systems and the small number of hierarchies and rules for each policy. A set of four domain definitions with four hierarchies is unlikely to be representative of all possible hierarchies.

In the next section, we will tackle the problem of hidden security mechanisms. In fact, the two testing approaches; the top-down and the bottom-up approaches are not able by their own to detect the hidden security mechanisms. There is a need to provide a tailored solution for this problem. This solution will be presented in the next section.

4.3 Detecting hidden security mechanisms

Recall that from an access control point of view, a software system is composed of three parts: the business logic, the PEP and PDP. Conceptually, the PDP is the decision logic that checks whether or not access to a resource can be granted while the PEP is the place where the policy is enforced in the application code. Technically, PDPs can be implemented in a multitude of ways, including configurable dedicated components, explicit pieces of code, and by imposing architectural constraints.
The business logic consists of the program code which is executed to implement the functional requirements. The policy is enforced through PEPs that are in charge of sending requests to the PDP which allows or denies access to the requested service.
There are some approaches like SecureUML (that we presented in the state of the art) that allow to automatically generate respective configuration files as well as application code from a set of related models. This kind of approach, however, only works in model-based development processes and generates applications from scratch. It is thus not applicable to legacy systems. If legacy systems' security policy rules (SP rules, which are for us the access control policy rules) are modified, the code of these systems often has to be changed as well. However, locating necessary modifications in the code is far from trivial. The reason is that the implementation of security mechanisms (PEP+PDP) is often interwoven with that of the business logic and can furthermore be implemented explicitly or implicitly.
To be able to tackle this problem, it is important to understand the nature of these security mechanisms implementing the security policy. There are three kinds of access control mechanisms; visible explicit mechanisms, hidden explicit mechanisms and implicit mechanisms. Explicit mechanisms are implemented by dedicated pieces of code, and can be either visible or hidden. A security mechanism is *visible* in a legacy system if there is a traceability link from (a part of) the security policy to the security mechanism. Otherwise, the mechanism is *hidden*. Due to the lack of documentation and traceability, the location of the code implementing a hidden mechanism may be lost, and so may the knowledge of how it works. This problem especially occurs for large and old legacy systems. Finally, *implicit* mechanisms are a result of technical constraints imposed by the architecture, platform or implementation of the business logic.
While the business logic of a legacy system may implement a given security policy, it may equally restrict the evolution of the security policy. This is a result of hidden explicit security mechanisms and implicit security mechanisms. Given all possible modifications of a security policy, we will use the term *flexibility* to denote the ability of a legacy system to evolve without tedious analysis and modifications of the program (detection and modification of the hidden and implicit security mechanisms, refactoring of the business logic).
Whenever a security policy evolves:

- The explicit security mechanisms are modified and have to be tested.
- Hidden security mechanisms may be in conflict with the new access control rules: they must be located and adapted.
- The business logic may be in conflict with the new security policy: the reasons for this conflict are the implicit mechanisms, which must be determined, and the design and the code may have to be modified and an adapted refactoring performed to make the legacy system more flexible.

When (black-box) testing legacy applications against new security policies reveals a mismatch between policy and implementation, locating the cause is usually difficult. Refactoring the design and re-programming some parts of the system may be prohibitively expensive. In some cases, it might even become impossible to deploy the new security policy.

The long-term goal of this research is a methodological and technological support for the evolution of legacy systems in the context of changing SPs. We take a first step by tackling the following problem. Given a legacy (or newly developed) system and a currently applicable SP, can we assess the coupling between business logic and access control logic? How difficult will it be to implement policy modifications?
Our solution involves a test-driven assessment methodology for the flexibility of legacy systems. We derive tests from policies. Then, we apply small mutations to the original policy in order to simulate incremental evolutionary steps. Then we derive tests from each mutated policy. Essentially, these tests are requests together with their expected results (deny or grant). Then, we disable the *visible security mechanisms* in the application. This is feasible precisely because of the existing traceability. Without hidden and implicit mechanisms, any request should now be granted. If implicit and hidden PDPs are present, in contrast, not all requests will be granted. By applying the generated tests to the system and comparing the actual with the expected access decisions, we extract information on implicit and hidden mechanisms.
Our test-driven approach provides an understanding of the technical reasons that restrict the evolution of the security policy for legacy systems. Because test cases are executable artifacts, they are precious means to help pinpoint those parts of the business logic that are not flexible w.r.t. the evolution of security policies.
The contribution of this work is the proposed methodology as well as its empirical validation.
The remainder of this section is organized as follows. Next, we present the background and the objectives, namely the detection of hidden and implicit security mechanisms. Then, we present the methodology, a two-stage test-driven assessment method. In the first step, the current system is analyzed based on exhaustive testing and in the second step, the system is assessed w.r.t. the possible evolutions of the current security policy. Finally, we present the experimental results and conclude.

4.3.1 Background

Before detailing the proposed test driven methodology to estimate system flexibility, we need to provide a few important concepts concerning the nature of security mechanisms.

Visible security mechanisms

Implementing control over the access to resources of a system can be done in a variety of ways. A classical architectural pattern is one component that is dedicated to access control. Typically, it works as a filter between the interface of the system and the control logic functions (services) or data (e.g. access to the database). If it exists, and if it is mentioned in the documentation or located through traceability links, such a component can be easily modified to implement a new security policy. The security mechanism is *explicit* since it has been created with the objective of controlling access. It is also *visible* since its location in the legacy system is known. For instance, PDPs of this kind can be specified with Sun's XACML. This allows the management of access control independently from the system. All requests pass through the PDP before getting to the system.
In order to illustrate explicit visible mechanisms, we consider that the PDP contains a `SecurityPolicyService` class which is called by the business services classes and returns the decision of the request (allow or deny). Figure 34 presents the point of view of the caller. The business services code of `borrowBook` automatically calls the visible PDP via the `SecurityPolicyService.check` method.

```
public  void borrowBook(User user, Book book) throws
SecuritPolicyViolationException {
// call to the security service
ServiceUtils.checkSecurity(user,
LibrarySecurityModel.BORROWBOOK_METHOD,
LibrarySecurityModel.BOOK_VIEW,
ContextManager.getTemporalContext());
// call to business objects
// borrow the book for the user
book.execute(Book.BORROW, user);
// call the dao class to update the DB
bookDAO.insertBorrow(userDTO, bookDTO);}
```

Figure 34 - An explicit and visible security mechanism

Regression testing reveals some hidden and implicit mechanisms

The ideal case for dealing with a system evolution is regression testing. Regression testing illustrates the problems that are caused by hidden and implicit security mechanisms.

If policies evolve, testing can be used to assess that the new security policy is correctly implemented. In the ideal scheme of regression testing, when security policy evolves, the tests cases are first applied to the current system and then are partially reused on the new system..

If only the access control policy evolves, this does not impact the business logic. The evolution of a security policy involves adding and removing access rules and adding or removing roles, activities and contexts. The new PDP is modified according to this evolution. The correctness of the PDP modifications must be ensured using regression testing. Test cases corresponding to deleted rules are removed.

This ideal regression scheme rarely applies. In most cases, the unchanged business logic can be in contradiction with the new access control policy, due to implicit or hidden mechanisms.

The first case of mismatch occurs if an explicit security mechanism is not visible, i.e., is *hidden* in the legacy system.

To illustrate these mechanisms, Figure 35 adds a hidden mechanism to the example presented in Figure 34. In the body of the method, after this security call has been executed, a new check is done which forbids borrowing books during week-ends. Disabling the first mechanism will not prevent this hidden prohibition from being executed. If the PDP component is modified in order to allow borrowing books during week-ends, the hidden mechanism will have to be located and deleted.

```
public void borrowBook(Book b, User user) {
// visible mechanism, call to the security policy service
SecurityPolicyService.check(user,
SecurityModel.BORROW_METHOD, Book.class, SecurityModel.DEFAULT_CONTEXT);
// do something else
// hidden mechanism
If(getDayOfWeek().equals("Sunday") || getDayOfWeek().equals("Saturday")) {
// this is not authorized throw a business exception
Throw new BusinessException("Not allowed to borrow in week-ends);}   …}
```

Figure 35 - Explicit hidden security mechanism

A second kind of mismatch is due to implicit constraints which restrict the access, due to the way the system has been designed, implemented and deployed.

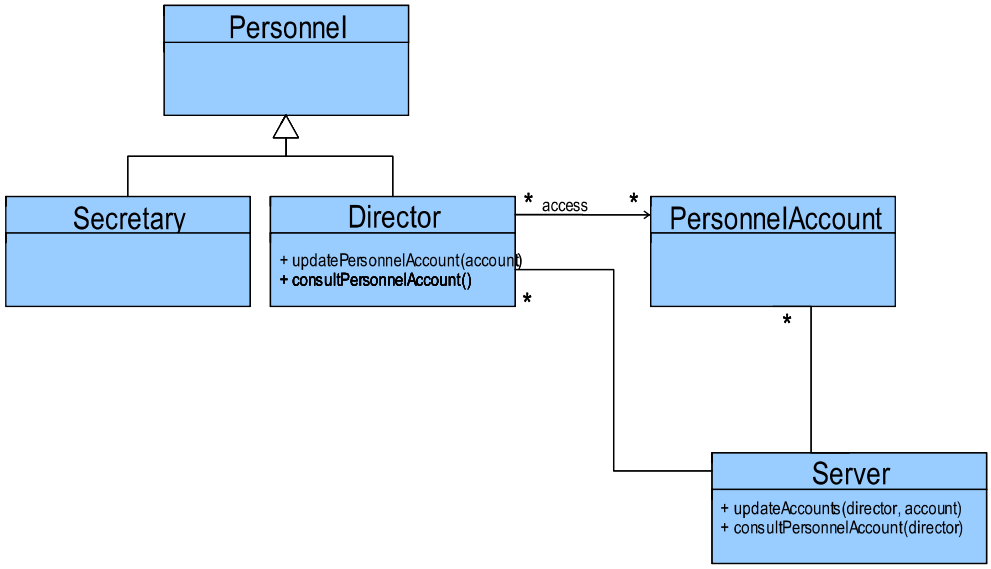


Figure 36 - Implicit security mechanism

By construction, a `Secretary` cannot directly call `updatePersonnelAccount`. A new access rule may specify that the director's secretary should now have access to personnel accounts. Since the program design does not have any direct reference from the secretary to the personnel accounts, we cannot add the rule "permission(secretary, update, account, always)". A simple refactoring would consist of moving the association "access" and the methods to the level of the class `Personnel`. Such a refactoring would also allow any `Personnel` instance to access the personnel accounts, which may be an unexpected change. So, the program will have to be carefully modified in several places to implement the desired evolution of the access control policy.

A last, more complicated case occurs when some access to a resource is granted by a hidden mechanism. In such a case, if the new security policy restricts this access, the visible PDP is modified but, depending on the execution flow, the user may still have his access granted due to the hidden mechanism.

In conclusion, the implementation of an evolving system policy is constrained by hidden and implicit security mechanisms. Even with a new PDP which implements the new security policy, these mechanisms may cause the system execution to fail. We will use testing for the early detection of such inconsistencies in the legacy system.

4.3.2 Flexibility analysis of the system to assess control policy changes

The difficulty of implementing an evolving policy is obviously related to the number of changes that must be applied to the current legacy system to enforce the new policy. Modifying the existing code may be more or less difficult, depending on the system size, on the programming languages (OO, Cobol etc.) and on the quality of documentation and design models. However, the mere modification of an existing application is costly because the legacy system business logic has to be analyzed.

We argue that testing offers a pragmatic way of dealing with this problem and helps estimate the cost of a planned evolution. The test-driven evolution process consists of:

- Assessing the current legacy system.
- Measuring its flexibility w.r.t. (micro-) evolutions.
- Diagnosing which functions and resources of the system are causing flexibility problems.

Exhaustive test-driven assessment of the current legacy system

The first question is to what extent the business logic implements hidden and implicit security mechanisms. These "hard-wired rules" may be redundant or even in conflict with some of the visible security mechanisms. The ratio of access control rules which are "hard-wired" in the business logic can be measured. We propose an exhaustive testing approach to assess the degree of hard-wiring of the security policy in the business logic. Exhaustive testing means that a test input is generated for each combination of roles, activities, views and contexts. The system flexibility is thus measured independently from a given security policy but it depends on the roles, activities, views and contexts.

We start by disabling the visible security mechanisms of the legacy system under test. This can mean either:

1. All requests should be granted now (universal permission),
2. Or all requests should be denied now (universal prohibition).

Recall that in contrast to function p which encodes the oracle, function s encodes the actual decisions taken by the PDP. Let s_+ denote the function that corresponds to the visible PDP that is disabled in the sense of universally **granting** access (i.e. **permission**). If there are no hidden or implicit mechanisms, we would expect:

$s_+(i)$=yes for all $i \in I$.

Similarly, let s_- denote the function that corresponds to the visible PDP that is disabled in the sense of universally **prohibiting** access. If there are no hidden or implicit mechanisms, we would expect:

$s_-(i)$=no for all $i \in I$.

- Let us first apply an input i to the **original SUT** (no changes in the PDP). If s(i)=p(i), the PDP behaves as expected. If s(i)≠p(i), then something is wrong – the legacy system does not do what it is supposed to do, i.e., what is described in the policy. In our setting, we may assume that this does not happen. We assume that the current system is consistent with its actual security policy.
- Let us now apply an input i to the SUT with the **visible PDP universally granting access**.
  a. If $s_+(i)$=yes, the result is inconclusive.
  Thus we cannot tell whether or not there is a hidden mechanism. Let *per1* be the number of test executions in this category.
  b. If, in contrast, $s_+(i)$=no, we have detected an implicit or hidden mechanism.
    i. If s(i)=no, we know for sure that disabling the visible PDP didn't have the desired effect. One scenario is that a request i1 is passed to the visible PDP and then forwarded to a hidden PDP which denies access. Disabling the visible PDP may mean that i1 is directly passed to the hidden PDP.
    ii. Symmetrically, let us assume that s(i)=yes. This means that disabling the visible PDP (in the sense of universally granting access) all of a sudden leads to a prohibition, a case that seems somewhat unlikely. However, one possibility is that in the original system, i1 is passed to the explicit PDP which transforms it into a distinct request i2 that is sent to the hidden PDP. The hidden PDP grants access to i2, and hence in sum, s(i1)=yes. Now, if the visible PDP is replaced by universal permission, this may mean that i1 is directly sent to the hidden PDP which denies access.

  Either way, a hidden mechanism is detected. Let *per2* be the number of test executions in this category.
  Note that this case is independent of the value of p(i). We don't need to know the actual policy.
- Finally, let us apply an input i to the SUT with the visible PDP universally prohibiting access.

a. If $s_-(i)$=yes, we have detected an implicit or hidden mechanism. In this situation, the visible PDP is somehow bypassed. Let *pro1* be the number of test executions in this category.

   i. If s(i)=yes, one scenario is that in the original system, a request i1 is first passed to the hidden mechanism which is directly granted, that is, control is transferred to the program logic without consulting the visible PDP. Modifying the visible PDP then obviously does not have any consequences.
   ii. If s(i)=no, one possible, albeit admittedly construed, scenario is that a request i1 is passed to the hidden PDP in the original system. Assume that this hidden PDP logs the reception and forwards i1 to the visible PDP who grants access. The positive response is passed back to the hidden PDP who decides that access cannot be granted. However, the hidden PDP may be configured in a way that a negative response of the explicit PDP (which is what we get by universal prohibition) is transformed into a positive one. This scenario appears rather unlikely but is not impossible.

b. If, in contrast, $s_-(i)$=no, the result is inconclusive. Let *pro2* be the number of test executions in this category.

   i. There may not be a hidden mechanism.
   ii. On the other hand, there may well be a hidden mechanism whose functioning is masked by the universally prohibiting visible PDP.

Note that this again is independent of the value of p(i).

The presence of hidden mechanism can be proven in those situations where $s_+(i)$=no and where $s_-(i)$=yes. Furthermore, the expected output p(i) does not matter in cases (2) and (3). The consequence is that the expected output part of a test case does not matter for our assessment. We "only" need to generate the input data for a test case but not the oracle. Furthermore, note that these results are independent of whether or not there is a default rule for policy evaluation, simply because the expected outcome p(i) does not matter.

In order to measure the flexibility of a system, let N be the number of exhaustive tests that is run. Clearly, N=per1+per2 = pro1+pro2.

The number of test cases that indicate the existence of hidden and implicit mechanisms is hence per2+ pro1. Conversely, the flexibility of a system is defined as the ratio

$$System\_flexibility = \frac{\text{per1} + \text{pro2}}{2\text{N}}.$$

A value of "1" means that our testing approach did not detect any hidden or implicit security mechanism (only inconclusive verdicts are emitted). A value of '0' means that every visible security mechanisms is doubled by a redundant mechanism which is hidden or implicit. When the ratio is close to zero, hidden and/or implicit security mechanisms are detected. These make the whole system rigid and its evolution problematic.

The problem with this approach to measuring system flexibility is that it forces all combinations of possible requests to be executed. It means, that a sequence must be produced for each possible input i. The number of test cases is:

$$N = |RN| \times |PN| \times |VN| \times |CN|,$$

This may be huge, depending on the number of roles, activities, views and contexts.

4.3.3 Test Selection for assessing system flexibility

In order to overcome this problem, we now take into consideration the old and the new policies. The expert responsible for the policy evolutions may want to know how much effort

will be needed in terms of code and design refactoring. So only the hidden and implicit mechanisms that would block an evolution have to be detected.

Test case criterion
In the previous chapter, we studied several test criteria to derive test cases from a security policy, including an analysis on the grounds of mutation analysis. We identified an efficient criterion which consists of deriving at least one test case per concrete access control rule.
In the following, we use this criterion for our experiments. The approach could be easily adapted to other criteria. The underlying assumption is that we have a set of test cases which are able to exercise and test each access control rule.

Decomposing policy evolutions
The evolution of a security policy can be decomposed into micro steps as shown in Figure 37:

1. δ+ for relaxing a policy (addition of a permission or removal of a prohibition),
2. δ- for restricting a policy (addition of a prohibition or removal of a permission).

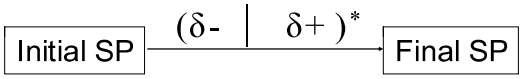


Figure 37 - Decomposition of the SP evolution into micro steps

Let *SP* be a security policy, i.e. a set of access control rules: permissions and prohibitions. Each micro-evolution results in a new security policy. We denote by SP^- the set of policies resulting from the restriction of the initial security policy (addition of a new prohibition, removal of a permission), and by SP^+ the security policies obtained by relaxing the initial policy (addition of a new permission, removal of a prohibition). We assume that a policy SP consists of two disjoint sets *Perm* and *Pro* that contain the permissions and prohibitions, respectively. Hence, $SP = Perm \cup Pro$. The set *NewPerm* contains all those permissions that can be added to the security policy. Similarly, *NewPro* contains all the prohibitions that can be added to the policy. Activities and views are not enumerated here for the reason that some activities cannot be used with every view. When adding a new rule we hence reuse activities with compatible views only.

$$NewPerm = \bigcup_{\substack{(\_,\_,a,v,\_)\in SP \\ r\in RN, c\in CN}} \{(permission, r, a, v, c)\} - SP$$

$$NewPro = \bigcup_{\substack{(\_,\_,a,v,\_)\in SP \\ r\in RN, c\in CN}} \{(prohibition, r, a, v, c)\} - SP$$

Activities and views are not enumerated here for the reason that activities are linked to specific views. When adding a new rule we reuse activities with compatible views. SP^- and SP^+ are then defined as follows:

$$SP^+ = \bigcup_{pro\in Pro} \{SP - \{pro\}\} \cup \bigcup_{nperm\in NewPerm} \{SP \cup \{nperm\}\}$$

$$SP^- = \bigcup_{perm\in Perm} \{SP - \{perm\}\} \cup \bigcup_{npro\in NewPro} \{SP \cup \{npro\}\}$$

Measuring system's flexibility for a given SP
Based on a given security policy and on all possible evolutions, we are now ready to estimate the overall flexibility of a legacy system. By applying micro-evolutions, we also expect to get an idea of the functionalities and resources which are the subject of hidden or implicit access restrictions. The analysis of flexibility should help pinpointing those parts of the system which are flexible when the SP evolves.

Figure 38 depicts a test-driven technique to provide such an estimate. From an initial security policy:

1. All possible security policy micro-evolutions are built.
2. The associated visible mechanisms are disabled for micro evolutions.
3. A test case is generated to test the evolution and applied to the business logic for each micro-evolution.
4. If the test case fails (detects an error), it means that the micro-evolution cannot be supported by the existing business logic without analysis and possibly refactoring. The legacy system cannot easily evolve in that direction.

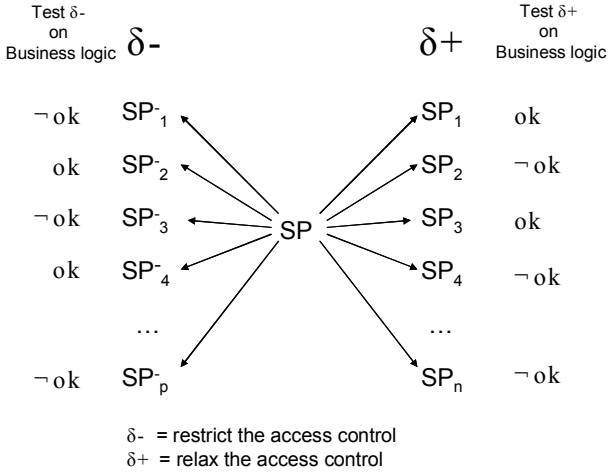


Figure 38 - Micro-evolutions of a legacy system access control policy

Let TSP^+ (resp. TSP^-) denote the test cases set built for testing each δ+ (resp. δ-) micro-evolution. We denote by tsp(pass) a pass verdict for a test case tsp, the overall flexibility of the legacy system for a given security policy SP, tested with the test suite TSP, is equal to:

$$flexibility(SP,TSP) = \frac{\left|\left\{tsp \in TSP^+ / tsp(pass)\right\} \cup \left\{tsp \in TSP^- / tsp(pass)\right\}\right|}{\left|\bigcup_{i=1..n} SP_i^+\right| + \left|\bigcup_{j=1..p} SP_j^-\right|}$$

$$rigidity = 1 - flexibility$$

Example: Consider 20 test cases used to test the 20 possible micro-evolutions: If five of them are executed and detect a failure (either a prohibition when permission is expected or vice versa), the flexibility for micro-evolutions is equal to 0.75. One evolution in four cannot be done unless we deal with hidden mechanisms..

Locating rigidity in the system

At each step, the analysis of micro-evolutions provides a useful diagnosis of those parts of the legacy system which are constrained by hidden or implicit security mechanism. The parts which cause the system to be "rigid" can be classified. The analysis we propose is two-fold.

1. Access to resources analysis: measuring the flexibility related to each resource (view) of the legacy system to detect problematic resources.
2. Access to functions analysis: measuring the flexibility related to each function (activity) interacting with a resource.

First, the degree to which a given resource is "protected" by hidden or implicit mechanisms is obtained. This is an estimate of the flexibility related to a given resource. For a view v (which corresponds to the data and resources of the system), we define:

Resource_flexibility(v)= percentage of test cases which pass when testing a rule related to the view (resource) *v*. Let TSP*v* be the test cases set exercising the rules related to a view *v*, we have:

$$resource\_flexibility(v) = \frac{\left|\{tsp \in TSPv / tsp(pass)\}\right|}{\left|TSPv\right|}$$

Second, the flexibility of system functions can be estimated in the same way. For an activity *a* (corresponding to a system function), we have:
Function_flexibility(a)= percentage of test cases which pass when testing a rule related to the activity (function) *a*. Let TSP*a* be the test cases set exercising the rules related to an activity *a*, we have:

$$Function\_flexibility(a) = \frac{\left|\{tsp \in TSPa / tsp(pass)\}\right|}{\left|TSPa\right|}$$

Test-driven evolution in practice
The flexibility measurement we propose provides an analysis of the micro-evolutions that the business logic may accept without refactoring or any modifications it. In practice, it is not necessary to make a complete analysis of the current system flexibility when a new policy is defined. Let SPinit be the initial security policy, and SPtarg the target security policy. The evolution from SPinit to SPtarg can be decomposed into micro steps (maybe with many possible solutions)

$$\Delta(SPinit) = SPt\arg = \delta_n^{-+} \circ \delta_{n-1}^{-+} \circ \delta_{n-2}^{-+} \ldots . \delta_1^{-+}(SPinit)$$

$$\text{where } \delta_i^{-+} \text{denotes either a } \delta^- \text{ or a } \delta^+ \text{ micro - evolution}$$

$$\text{and } \circ \text{ a function composition.}$$

The test driven process is similar to the test driven development principles promoted with XP. This test driven evolution technique involves repeatedly:

- Writing a test case associated to the δ_i^{-+} micro-evolution.
- Detecting a potential rigidity in the legacy system.
- Fixing the problem to pass the test.
- Implementing and documenting only the micro-step in the PDP.

This pragmatic approach assists the safe evolution of a legacy system.

4.3.4 Experiments and discussion

Like in the previous chapter, we applied our approach on the three case studies: LMS; ASMS and VMS.

VMS results
Table 17 presents the results for the VMS. The overall flexibility of the initial policy is 0.35, which means that 35% of the security rules should be modifiable without code and design refactoring. It also reveals that the remaining is constrained by hidden/implicit security mechanisms.

Table 17 - # of flexible and rigid rules for VMS

	Flex. rules	Rigid Rules	System flexibility
results	**20**	**36**	**0.35**

To obtain a more accurate diagnosis, we consider which resources and views are the subjects of hidden/implicit mechanisms as shown.

Table 18 - # of flexible and rigid rules for each resource/view for VMS

Resource\View	Flex. rules	Rigid rules	All	Flexibility
Meeting	**12**	**36**	**48**	**0.25**
PersonnelAccount	**6**	**0**	**6**	**1**
UserAccount	**2**	**0**	**2**	**1**

Table 18 leads to the understanding that any evolution concerning `PersonnelAccount` or `UserAccount` is possible and that the problems of rigidity concern the `Meeting` resource. We can go further in the analysis and check which functions/activities are flexible in terms of security policy micro-evolutions. The results presented in Table 19 show that the functions which are more rigid are related to the `Meeting` manipulation. It also reveals that some of the functions related to this resource are flexible (like opening a meeting).

Table 19 - Flexibility rate for each function/activity for VMS

Function/Activity	Flexibility
updatePersonnelAccount	**1**
updateUserAccount	**1**
askToSpeak	**0.13**
leaveMeeting	**0.14**
overSpeaking	**1**
closeMeeting	**1**
setMeetingAgenda	**0.14**
setMeetingModerator	**0.14**
speakInMeeting	**0.14**
setMeetingTitle	**0.14**
deleteUserAccount	**1**
openMeeting	**1**
handover	**1**
deletePersonnelAccount	**1**

ASMS results

We obtained the following results for the ASMS. The system flexibility value is very close to the previous example and the diagnosis allows determining that only the `Comment` resource can support security evolutions without restriction. Concerning the other resources, `UserAccount` cannot evolve at all without modifying the code. Only 16% of the possible evolutions can be applied without problems to the `Bid` resource and 31% for `PersonnelAccount`.

Table 20 - # of flexible and rigid rules for ASMS

	Flex. rules	Rigid Rules	System flexibility
results	**12**	**22**	**0.34**

Table 21 - # of flexible and rigid rules for each resource/view for ASMS

Resource\View	Flex. rules	Rigid rules	All	Flexibility
Comment	5	0	5	1
Bid	1	5	6	0.16
PersonnelAccount	5	11	16	0.31
UserAccount	0	5	5	0

The activities/functions which are the cause of these rigidity problems can be located more precisely with Table 22. For both ASMS and VMS, only δ+ evolutions were problematic, which means that no hidden mechanism has been detected which grants permission while a prohibition is expected. No hidden mechanisms were detected, which may be explained by the fact the systems are new (not real "old" legacy systems), the business model designs are object-oriented, and that the PDPs are centralized in dedicated classes. In such cases, the main rigidity is caused by design constraints, i.e. implicit mechanisms, which only restrict the δ+ evolutions (relaxing access).

Table 22 - Flexibility rate for each function/activity for ASMS

Function/Activity	**Flexibility**
updatePersonnelAccount	1
consultBid	0
consultComment	1
postComment	1
consultOldBids	0
updateBid	0
deleteBid	0.69
deleteComment	1
deleteUserAccount	0
consultPersonnelAccount	0
deletePersonnelAccount	0

LMS: Library Management System

The library management system is interesting since it is fully flexible. In fact, based on the return of experience of the two first studies, we built it to be flexible: the business model contains a `Role` class and allows the access to be fully controlled from the PDP. The principle for building a business model with no implicit mechanism may consists of making the access control concepts explicit (reification) in the business model.

4.3.5 Summary

As far as we know, no previous studies focused on the problem of automatically identifying implicit/hidden access control mechanisms in legacy systems. Xie et al. [132] propose a machine learning algorithm to infer properties of XACML policies. This approach focuses on the PDP and infers the policy properties by analyzing request-response pairs. Their approach does not consider the whole system (PDP + system). As pointed out by [133], guiding the

systems security policy evolution is a challenging issue. Our test-driven technique suggests that testing is a pragmatic technique to detect and locate (either using exhaustive or security policy based testing) hidden/implicit security mechanisms which might make the legacy evolution a nightmare for domain experts. The approach is still dependent on the generation of SP test cases.

4.4 Conclusion

In this chapter we presented three complementary approaches to deal with access control testing. Firstly, we showed how to select and transform functional tests into security tests. Secondly, we presented a model based approach for generating security tests. The approach is based on an adapted pair-wise technique. We performed an extensive empirical study to compare between the different strategies and to select which is the best criterion to be used to generate the test targets. Thirdly, we pointed out an important problem related access control testing. Due to the way access control mechanisms are included in the application, there are always hidden security mechanisms that represent a threat to the evolution of the access control policy. We presented an approach based on mutation that helps detecting these hidden mechanisms. All these three techniques were applied to case studies and this allowed us to show their effectiveness.

In this chapter, we focused on testing the internal security components of the system, namely the access control mechanisms. There is however an important task that needs to be done to complete this effort. It is important to validate that the interface and provide a tool to make it secure automatically. Next chapter will tackle this issue. We will focus on web application and tackle an important issue that is specific to web application. We will tackle the problem of bypass testing. We will propose a technique to perform bypass testing and propose a new solution to protect against bypass attacks.

Chapter 5
Automatic analysis of web applications for shielding and bypass testing

The previous two chapters focused on the issue of access control policy testing. They provided an overview of the problem and several approaches related to test generation, test selection and test transformation in addition to an approach for detecting hidden mechanisms and guiding the evolution of the system.
This chapter is complementary to the previous chapters as it tackles the issue of testing the external part of web application, namely web interfaces. More precisely, this chapter tackles the issue of bypass testing, which is related to user input validation.
User input validation is a critical task to limit attacks on web applications. In typical client-server architectures, validation is performed on the client side. However, this is not sufficient because it can be bypassed by attackers. User input validation thus has to be duplicated on the client-side (HTML pages) and the server-side (PHP or JSP etc.).
This chapter presents a new approach for shielding and testing web application against bypass attacks. We automatically analyze HTML pages in order to extract all the constraints on user inputs in addition to the JavaScript validation code. Then, we leverage these constraints for an automated synthesis of a shield, a reverse-proxy tool that protects the server side. To validate the web applications and demonstrate the effectiveness of the shield, we have built a bypass testing tool that generates test data violating the extracted constraints. We have implemented these processes in a tool suite. We conducted an experimental study on five popular open-source web-applications to illustrate the different flaws our test data can detect and evaluate the impact of the shield on performance.

5.1 Introduction

One important property shared by most web applications is the client-server architecture, which roughly divides the application code in two parts. The main part executes on the server, while the client part includes a browser in charge of the interpretation of the HTML and JavaScript code (other components exist like flash, java applets, ActiveX etc.).
In this architecture, the input validation of web application is performed both by the client and server sides. In practice, many validation treatments are under the responsibility of the client. Decentralizing the execution of input validation allows alleviating the load of inputs to be checked by the server-side. Incorrect user inputs are detected by the client-side code (HTML and JavaScript code) and not sent to the server.
This architecture implicitly assumes that the client is expected to check the validity of its inputs before calling the server, while the server is responsible for its outputs. This is a perfect example of design-by-contract [134], which relies on the assumption that both parts are trustable. However, the assumption that the client is trustable is dangerous, as recalled by J. Offutt:[107] “Validating input data on the client is like asking your opponent to hold your shield in a sword fight”. It is not possible to trust the execution of the validation on the client side. That is why it is highly recommenced to duplicate the validation process and perform it at the server side. In addition, input validation is an important security issue. The SANS top 25 [18] reports that one of the main vectors of attacks is input validation. Relying on the client will weaken the input validation. In fact, a malicious user is able to modify the JavaScript code using some plugins (like Firebug or DragonFly). These tools enable for the potential attacker to *bypass* the client-side by modifying the HTML and JavaScript code and thus disabling the client-side input validation. Therefore, Hackers can bypass the client-side input

validation and send malicious requests to the server-side directly. Furthermore, the server cannot detect that client-side input validations have been disabled or hacked. An analysis of bypass-based attacks has been initially proposed by Offutt et al. [107, 120], demonstrating that the n-tiers architectures may lead to security vulnerabilities, or at least to robustness problems for the server side.
As a basic counter-measure, it is recommended to carefully filter and check user inputs. In this chapter, we propose an automated process which either allows:
- auditing the server-side weaknesses/vulnerabilities (in that case the server-side application code needs to be manually adapted) through systematic bypass testing [107].
- or shielding it by building a reverse proxy security component, called Bypass-Shield that captures the client-side validation constraints, extends them, and enforces them. The shield implements the contracts between client/server as an independent component, making a design-by-contract applicable in the context of web application security and robustness.
The common mechanism we use for both analyses is a (semi-)automated extraction of client-side validation constraints (HTML and JavaScript).
On one hand, bypass testing involves systematically violating these constraints. Then requests are built to include these erroneous (robustness testing) or malicious data (security testing). Finally they are sent to the server and may lead to robustness and security problems.
On the other hand, shielding the application involves building an "in-the-middle" component, which is the trustable intermediate which guarantees that contracts are fulfilled by the client (because located on the server-side).
The remainder of this chapter is organized as follows. Next section presents the background concepts. Then, we present the overall approach and describe the process. Afterwards, we detail each of the three processes included in the approach, respectively, the client-side analysis for collecting the constraints, the Bypass-Shield and the automated bypass-testing.

5.2 Context and motivations

This section introduces the main concepts used in this chapter. It presents the input validation architecture used in web applications and the client-side validation techniques.

5.2.1 Definitions

Client-side: it includes the part of the web application that is executed by the client browser (Firefox, Internet Explorer etc.). The client-side contains the HTML code, the cookies, the Java applets, and the flash programs etc. that are executed by the client machine. They are sent by the server and the client is in charge of interpreting this code to be displayed (HTML code) and executing it.

Server-side: The server side contains the core application code (which is often called business code), the web server which is the framework in which the business code runs. Several technologies of web servers exist (Apache, IIS etc.). The server-side may contain a database.

Form: It is located in The HTML code. It is a set of tags defining fields to be filled by end users. Then, this data is usually sent to the server-side to be treated. The server responds according to the form data and the requested service (store, search, register, delete etc.).

Input Validation: The process of validating user inputs. It can be performed in the client-side and in the server-side.

Bypass Testing: The process of testing web applications by bypassing client-side input validation and triggering the server-side input validation (if it exists) [107]. Bypassing is possible either via some browser plugins (like Firebug or Opera Dragonfly) or by simply using code to build the requests to be sent directly to the server (using Java or C++ for example).
Client-side constraints: They are constraints expressed by HTML code (like *MaxLength*) or JavaScript functions. They are used to check user inputs. These constraints are part of the client-side input-validation process. They enforce limitations and tailored conditions on the user inputs.
Robustness bypass testing: Aiming at challenging the robustness of the web application server-side by directly providing erroneous inputs. These inputs violating the client-side constraints are sent to the server-side. The final goal is to uncover robustness problems and improve the server-side code.
Security bypass Testing: Targets the evaluation of server-side security. The data sent in this case represents typical attack patterns, including some known attacks (code injection attacks). Some predefined attack patterns are injected and sent the server, which is expected to filter, sanitize or block these malicious data. The final objective is to highlight security issues that need to be treated.

5.2.2 The input validation architecture

The traditional client-server architecture defines a distributed model which involves two different places where the application code executes.

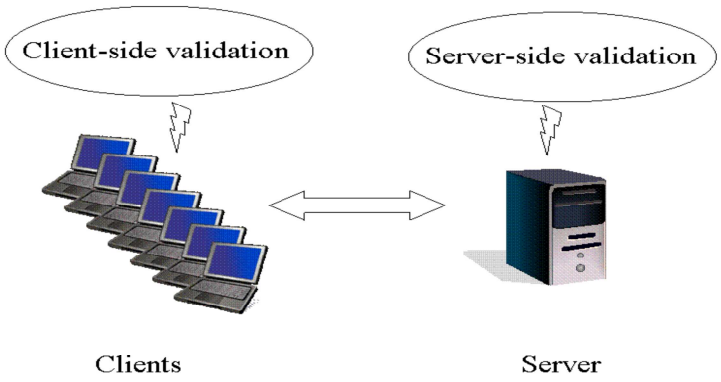


Figure 39 - Input validation model

Figure 39 depicts the input validation model currently used in web application. The input validation is executed firstly in the client-side. Once the data is validated through client-side validation, it is sent to the server that performs this same input validation in addition to other complex and more advanced validation. For instance, the server may need to access to a database to check the validity of some user inputs.
Erroneous data is detected at an early stage, at client-side and is not sent to the server. Therefore, the code executes on the client machine and the server does not intervene.

5.2.3 Client-side validation techniques:

Client-side input validation is implemented through two different kinds of code:

- Hardcoded HTML code.
- JavaScript code.

Hardcoded HTML code allows defining a set of predefined constraints. These constraints are implemented by expressing the corresponding tag property. For instance, the max length constraint has to be expressed within the input tag (like `maxlenght=20`). Other constraints

are expressed by choosing one particular tag. By construction, it constrains the kind of user inputs. Check boxes can only be checked or unchecked. In radio button group, only one can be checked.
JavaScript code makes it possible to express more advanced and adapted constraints. Using JavaScript code which includes conditions, loops and regular expressions (among other code facilities) it is possible to express any constraints on the user inputs. This JavaScript code can be executed before submitting the form to the server-side. Erroneous inputs are rejected by the JavaScript code and not sent to the server. Then a message is displayed to the end user to indicate the erroneous inputs that should be corrected.
To give the intuition of a typical JavaScript constraint, we present the following JavaScript code that allows checking that the input string conforms to a regular expression:

```
function checkEmail(myForm) {
if (/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/
.test(myForm.emailAddr.value))
{return true;}
alert("Invalid Email");
return false; }
```

5.3 Overview of the approach

From the same initial treatment, the parsing of the web page, the process we propose allows the derivation of a reverse-proxy (Bypass-Shield) and the creation of robustness and security test cases to validate the shield. Figure 1 shows an overview of this process.

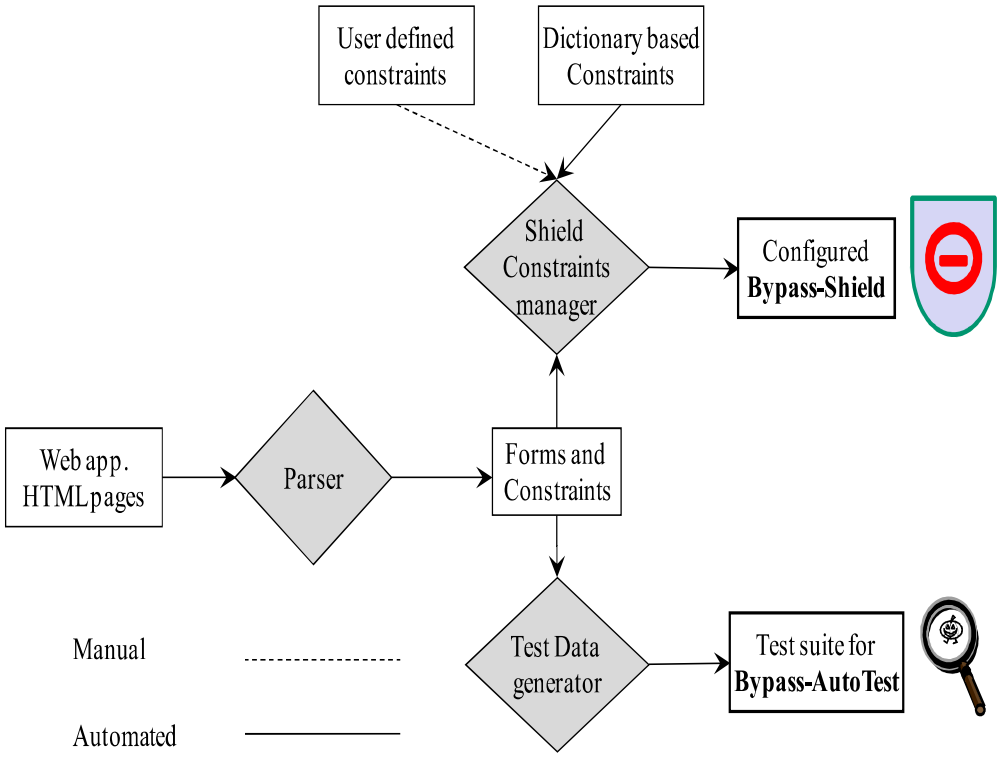


Figure 40 - Constraint-based Testing and shielding web applications

The constraints are Boolean expressions that evaluate to true if the value is correct and false when it is not (the input value is violating the constraint).The overall process involves three main steps.

The First step involves parsing systematically all pages in order to collect forms along with their respective inputs and constraints. Then these client-side constraints are stored in a file. The result of this step is used in the next two steps. The hard points and originality of this first step are:

- To exhaustively analyze a website in depth. This means taking into account the login process to access all website pages. In addition to an automated crawler, manual navigation is needed to completely parse the website.
- To deal with JavaScript code used for validating user inputs.

The second step aims at shielding the web application using the initial set of constraints collected at step 1. It results in a reverse-proxy, called Bypass-Shield, which intercepts and checks the inputs from the client as well as server responses. The collected constraints are completed with automated and manual constraints. The automated constraints are based on a dictionary listing a set of constraints to be applied to specific inputs (for instance emails have a specific format). The shield contract manager uses the name of the input to find any available predefined constraint. This task leaves out untreated inputs. The manual addition of constraints completes the automated process by providing a user-friendly tool to add new constraints. The obtained constraints are included in the Bypass-Shield which will intercept user requests and check the validity of user submitted inputs.

The third step includes a test generation process, based on the information collected at step 1. The constraints are used to generate test data for bypass testing. The idea of bypass testing is to generate data which systematically violate the client-side constraints. As a result, we obtain a test tool, Bypass-AutoTest, which allows auditing how the server reacts when receiving every kind of invalid data. Bypass-AutoTest has been implemented to check that the Shield works as expected and prevents attacks issued by the client-side.

Step2 (Shielding) and step 3 (Auditing through testing) can be used independently or altogether. In the first case, the shield allows protecting the server without modifying the server's code. The advantage is that the security controls are centralized in an independent component, which is responsible for the contracts between client and server. In the other case, the audit allows identifying the server robustness weaknesses and security flaws.

We distinguish between these two kinds of issues. From a pure testing point of view, the oracle, the general interpretation of the results and the impact differ. This means that the intent and the oracle are not the same.

The robustness oracle analyzes the server responses to find error messages (like java stack trace) or unexpected behavior (returning the same page without showing warnings), while the security oracle seeks to find any information or behavior that will harm the security of the application. For instance, the security oracle checks that the server responses does not reveal any critical information that can be used by hackers, or that the server does not behaves in an insecure way.

5.4 Client-side analysis for constraints identification

This work focuses on bypass-attacks exploiting forms, and does not handle attacks exploiting other attack vectors (like the cookies or HTTP headers). The goal of the HTML analysis is to collect all the user inputs from client-side web pages. User inputs are mainly forms which are filled by the end users. This task is fulfilled using three complementary techniques:

- Automated crawling of the application pages
- Manual navigation in the website to explore all possible scenarios
- Automated navigation using functional test built using testing framework like (HttpUnit or Selenium).

In fact, automated crawling of the web is too simple to allow reaching all the application html pages. Modern web application use partial page refresh, asynchronous request and have often just one URL throughout. Visiting all the links will not allow reaching all the HTML pages. In addition, the behavior depends on the client. For these reasons, we complete the automated crawling by two strategies, the manual surfing and the execution of functional tests.
In this section, we show how the automated crawler works and how the bypass shield is used to collect the HTML and finally how we deal with the JavaScript constraints.

5.4.1 The automated crawler

The crawler allows exploring all the available web pages by visiting all the links. The parser can be configured to use a login and password. This allows going beyond the login web page and exploring the entire web application pages. Furthermore, the parser can be configured to avoid visiting some links that will disconnect the user from the web application. This feature is implemented in a generic way using regular expression to create the set of links to be ignored. In addition, the parser only visits the pages that are in the base URL. This leads to avoid leaving the web application and parsing other websites/web pages.
This crawler allows collecting and storing all the forms along with the associated constraints.

5.4.2 Manual navigation and the use of functional tests

During this step testers are asked to run functional tests or navigate manually throughout the web application and to explore all the possible scenarios. During this step the bypass-shield is set in monitoring mode and collects and analyzing the code that is sent to the client. As shown in Figure 41, the shield intercepts the web pages that are sent to the client and analyzes them in order to collect the forms.

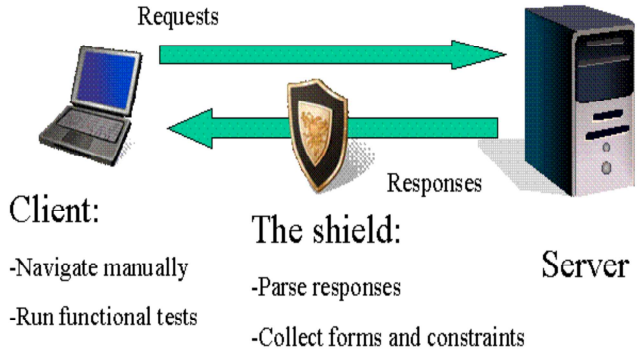


Figure 41 - Collecting constraints using manual navigation and functional tests

The forms and constraints that are collected are stored with the others forms previously collected using the crawler. The process of extracting the HTML is common to the crawler and the manual step. Next, we show how the HTML code is analyzed and how the constraints are extracted.

5.4.3 Collecting HTML constraints

The process of collecting the list of user inputs along with their constraints from the HTML code is presented in Figure 42.
Each web page is analyzed to locate all the forms. These forms are parsed to collect their inputs. The input may contain some predefined HTML constraints. In Figure 42, the max length attribute defines a constraint on the length of the age attribute. At this stage, only HTML constraints are treated. A separate and parallel process allows dealing with JavaScript code. It will be presented in the next section.

Once the form and its inputs are collected, the tool creates a set of objects for the Form and the inputs. We have modeled the types of inputs and the constraints as classes. This approach allows querying forms and inputs and facilitates the test data generation, the construction of the test suite and the configuration of the bypass-shield.

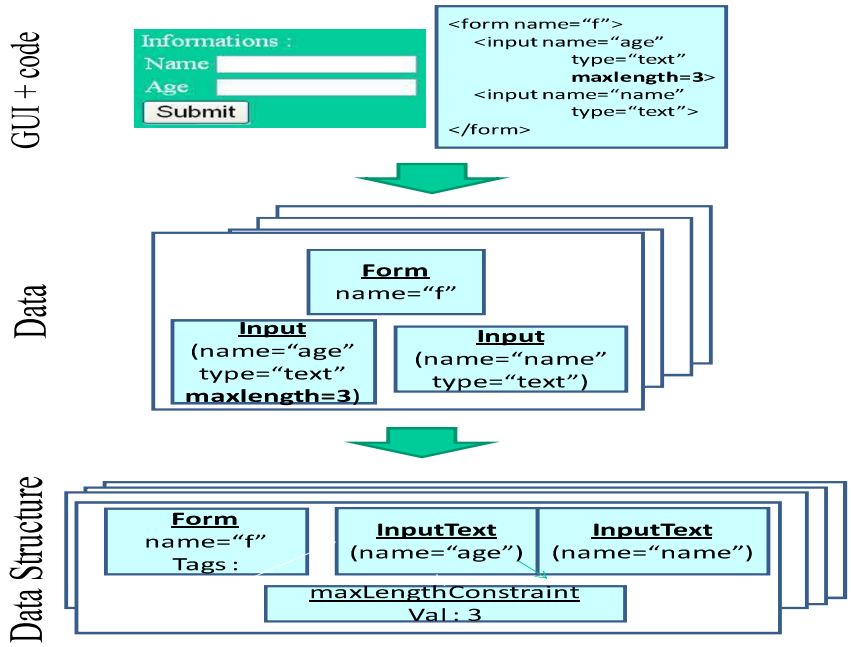


Figure 42 - Collecting HTML constraints from the GUI

Each input is categorized based on its type. Table 23 shows these predefined constraints. For instance, the text input corresponds to InputText object. Each constraint is extracted from the inputs and stored depending on its type.

Table 23 - HTML predefined constraints

Constraint name	Description
FormMethod	Method is either Get or POST and should not be modified.
Disabled	The input is disabled and hence is not sent.
MaxLength	Maximum input size.
MultipleValue	The value should be one of the values set
ReadOnly	The input is read only and cannot be modified.
RequiredValue	The input value is required and cannot be empty
SingleValue	The input has a single value, Null or that single value

5.4.4 Interpreting JavaScript

As we have mentioned previously, the JavaScript is not directly parsed. The difficulty of dealing with JavaScript code is due to its grammar which is complex, and this makes the semantic analysis very hard to automate. The solution that we propose involves running the client-side JavaScript validation code itself inside the shied. Instead of inferring the semantic of the JavaScript constraints, we actually run the JavaScript code inside the shield

automatically when a form is submitted. We lift the JavaScript code from the client, and rerun it automatically in the shield. Figure 1 shows an overview of this process.
The main goals of this process involve:

- Locating the JavaScript code implementing constraints on user inputs: the JavaScript validation code is usually triggered just before submitting the form (using for instance the *onsubmit* attribute) or attached to specific text input events (like *onblur* when the user finishes typing and leaves a text input).
- Extracting and storing this code: We should keep a mapping between the JavaScript code and the related form or input.

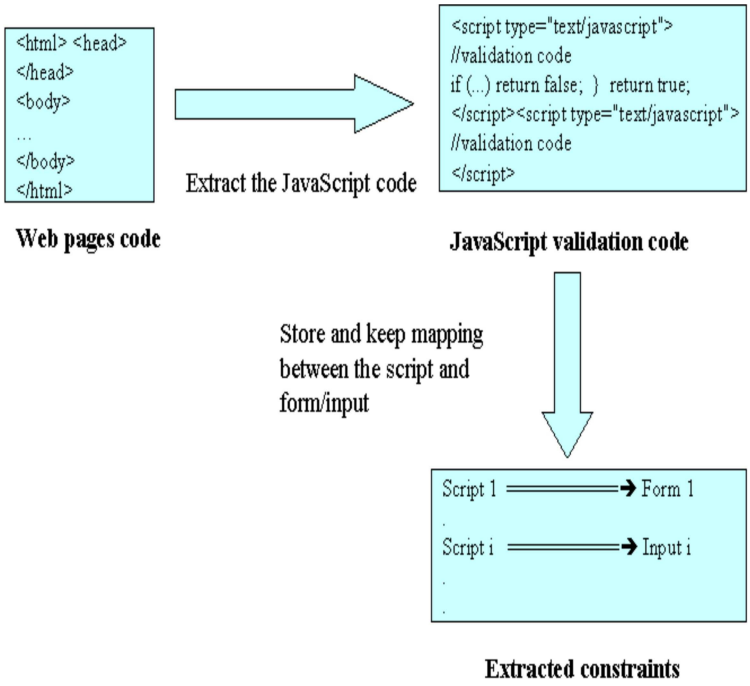


Figure 43 - The process for extracting JavaScript validation code

This process runs in parallel with the extraction of HTML static constraints. As shown in Figure 43, we extract for a given webpage the JavaScript code related to the input validation. Then we keep a mapping between the JavaScript code and the related form or input.

5.5 Server-side shield: a shield tool for protecting against bypass attacks

In this section we present the bypass-shield and its components. First, we introduce the contract manager tool that allows the addition of constraints to the set that has been generated in previous step. Then, the bypass-shield is presented in details.

5.5.1 The contracts manager

The *contracts manager* allows adding new constraints in order to complete the set of constraints extracted from the client's HTML code. Security engineers can add constraints manually through this manager and it also adds new constraints automatically. Table 24 presents the set of primitive constraints provided by the manager:

Table 24 – Predefined constraints

Constraint name	Description
Interval	The value should be within the defined interval
MinLength	Minimum input size
RegEx	The value conforms to the given regular expression
Date	The value has a date format
NumberFormat	The value is numeric
ListOfValues	This value is among a list of values
Required	The input has to be filled
Range	The value is within an interval

The contracts manager is able to automatically inject constraints using a dictionary file, in which the user defines a set of RegEx constraints. These constraints are automatically mapped to input according to their tag names. For instance a tag with the name email will take the following RegEx constraint:

```
([a-zA-Z0-9_]|\\-|\\.)@(([a-zA-Z0-9_]
|\\-)+\\.)[a-zA-Z]{2,4}
```

This constraint forces the email addresses to satisfy a specific format. The manager automatically adds this constraint for each email tag, even if the email format was not enforced in the HTML code. This verification is usually added using the JavaScript code. On the basis of the dictionary, the manager can thus partly compensate the fact that we don't analyze JavaScript code.
Once the configuration file that is used by the bypass-shield (it is a binary file storing the constraints) is filled with constraints it is fed to bypass-shield which is in charge of protecting the side-side part from bypass-attacks.

5.5.2 The bypass-shield

As shown in Figure 44, the bypass-shield aims at protecting and serves as a barrier against the attacks. It is installed as a reverse proxy on the server side of the web application. Therefore, all the requests are intercepted by the bypass-shield and checked.
For each request, the bypass-shield performs the following steps:

1. Intercept the request
2. Extract the user inputs and locate the corresponding form that was filled out by the user.
3. Check and validate the input according to the related constraints and run the related JavaScript validation code.
4. Accept the request and send it to the server side application or reject and send an error message to the client.

Only requests containing user inputs are checked. The URL requests are passed to the server. The server is expected to respond by sending back the webpage (the code) of that URL.

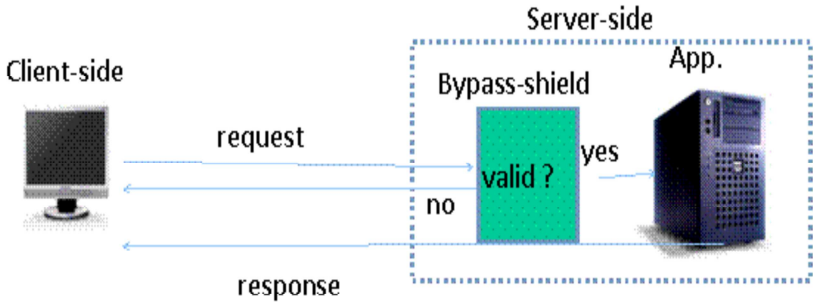


Figure 44 - Overview of the shield

The user inputs are extracted from the selected requests. In order to locate the corresponding form, the algorithm tries to find among the stored forms (they are stored in a binary file) the one having the same inputs (same number and same name) and the same action URL (the URL to which the inputs are sent). The HTTP request contains all the names of inputs along with their values. The following example illustrates how the algorithm extracts the input names from the request (in this example they are the name, the phone and the zip code). The action URL is simply the request URL without the inputs part.

The request:
http://www.mysite.com/account.php?name=Tim&phone=0234234354&zipcode=75000
The extracted inputs: name, phone and zip code
The action URL: http://www.mysite.com/account.php

Once the form corresponding to the request is located, the bypass-shield performs the validation of the inputs using the related constraints. All the constraints should be respected. If the inputs do not satisfy the constraints, the request is not forwarded to the server and an error message explaining the problem is sent to the user. In addition, the JavaScript code that is related to the form or one of its inputs is executed on these inputs (using a JavaScript execution engine). The result is a Boolean value (true or false) that means: accept or reject the input data. When all constraints are satisfied and JavaScript validation is succeeds, the request is forwarded to the server.

5.5.3 Impact of constraints enforcement on security

By validating client-side constraints, the shield prevents some code-injection based attacks like SQL injection on numeric fields, by enforcing constraints on numeric fields so it becomes impossible to bypass this constraint and perform any code-injection attack. In addition, it makes it harder for attackers to do long SQL injections when the field length is limited.
By ensuring that the provided parameters are strictly those required, the shields limit ids evasion techniques like http parameter pollution [45], which is a new kind of attack that involves exploiting parameters in the URL (by duplicating them and injecting attacks).
This kind of enforcement reduces the attack surface of the shielded web application. Using of the shield in front of WebGoat [46] (which is a vulnerable OWASP web application used for teaching security) is a good example to show how the bypass-shield provides extra security, and where it does not. As shown in WebGoat, the developers focus very often the fields that are under user's control (like text fields), and neglect performing input validation on other fields, like check boxes or select lists, which have predefined values. Enforcing constraints on these fields is relatively simple since the expected values are known. By these simple

constraints, the shield ensures that the application behaves as expected by the developer and protects against some attacks.

5.6 Automated bypass testing

This section details the automated generation of bypass testing. The client-side analysis provides useful information on constraints which can be used directly to generate data violating these constraints. On one hand, this data can be used within our bypass-testing too or other security tools like fuzzing tools in order to audit the web application. On the other hand, they could be used for evaluating the bypass-shield.

The data generation process involves three major steps. We start with the automatic generation of malicious test data that violate the client-side constraints. Then, we build complete requests that include the malicious data and for which all other tags contain valid data. These requests are sent to the server-side. The last step involves the analysis of the server responses and the automated classification of the results, in order to facilitate their interpretation by the testers.

5.6.1 The generation of malicious test data

The initial step involves generating test data that bypass the client-side constraints. For each constraint, we have created a data generator in our Bypass-AutoTest tool. This data generator is in charge of creating the data violating a specific constraint. The following example illustrates the approach.

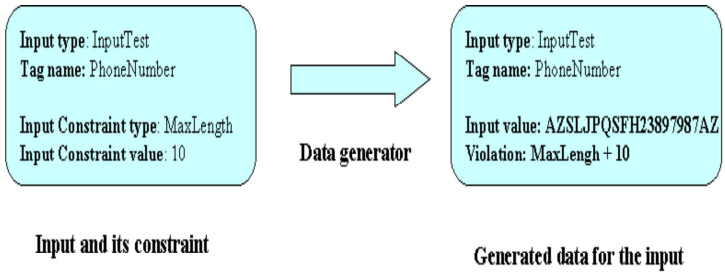


Figure 45 - Example of a generated input

In the example of Figure 45, the input is a phone number with a maximum length limit (10 characters). The data generator takes this input and its constraint and generates a random string with a length exceeding the required max length by ten. The interval of violation can be defined by the user (ten is the default value).

Table 25 - The way constraints are violated

Constraint	Violation
FormMethod	Use another form method
Disabled	Make it enabled and generate a random string
MaxLength	Generate data exceeding MaxLength
MultipleValue	Generate a different random value
SingleValue	Send more that one value
Interval	Create a value outside that interval
MinLength	Create a value shorter that MinLength (if MinLength > 0)
RegEx	Create a value not conform to RegEx
Date	Create a random value that is not a date
NumberFormat	Generate a string with alphabetic characters
ListOfValues	Generate a value not in the list of values
Range	Generate a value outside that range

Table 25 shows, for each constraint, the kind of data that is automatically generated in order to violate that constraint.

5.6.2 Construction and execution of bypass tests

This step requires the construction of suitable requests from the set of test input generated in previous step. For the request to be valid, all the form tags must be filled. In fact, each test request contains only one unique malicious input; all other inputs are valid with respect to the constraints.

The presence of a single malicious data in each test request is of cardinal importance. This way, we avoid any side-effects due to the server-side rejecting the request. Also, if the test fails, revealing the lack of input validation or a serious security flaw, the fact that each request contains only one maliciously input facilitates the localization of the source of this problem.

The process of preparing and executing the request is detailed in Figure 46.

The malicious and genuine data are combined to create the requests. These latter are sent to the server side to be processed.

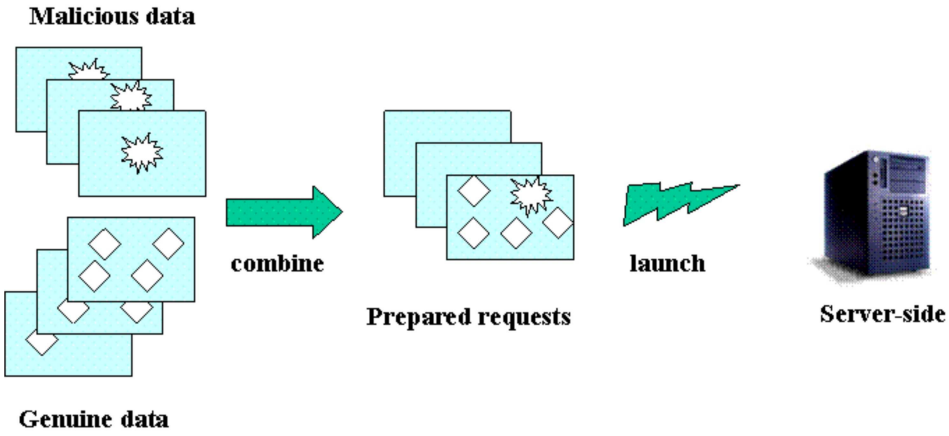


Figure 46 - Execution of requests

After sending the requests, the server responds and all these responses are stored in order to be interpreted, to filter the responses revealing flaws and bugs and to automatically classify these results and generate the result report.

5.6.3 Filtering and classifying the results

This step aims at analyzing the server responses (returned HTML pages). The objective is to automatically filter the suspicious responses. The analysis and classification of the results allows finding the responses revealing bug and security flaws in the web application. The approach involves two steps:

1. Filtering results: It is based on the signature of known issues; for example the Java and PHP exception stack traces or the HTML error code (like HTML 404: page not found). This step separates results revealing flaws from the others.
2. Classifying the filtered results: Each result is classified in a category of results. Table 26 shows the three categories of results.

Table 26 - The three categories of results

Category	Description
Database	Bugs and flaws related to the database
Business logic	Bugs and flaws involving the core application code
Web Server	Bugs and flaws due to the web server (apache, IIS etc.)

The results are classified according to the origin of the bug or the security flaw. This classification helps the testers to define priorities on bugs and to know where most of the bugs are located. Then the tester can choose which bugs have to be handled first. The database bugs are usually the most important ones because they can lead to data leakage, or even worse granting a full access to the database. They have to be treated first.
The Business logic issues are related to the way the business code deals with the inputs. The lack of input validation in the server-side code leads to several kinds of exceptions and can provide hackers with means to hack into the system and access unauthorized content or take control of the application. Furthermore, they can exploit these flaws to take down the system by DOS attacks (deny of service which leads to shutting down the web application).
The web server bugs and flaws are rather related to the configuration issues. A bad configuration leads to some known security flaws, like revealing the web server version. When the web server is not batched, a malicious user can find the exploits and attacks for that version (the attack scripts are available online) and use them against the Web Server.

5.7 Experiments and results

In this section, we present a summary of the first results conducted. We chose to use our tool on four popular web applications. The main goal of this study is to evaluate the efficiency of the shield. The objective is to show that the shield is able to effectively block bypass attacks. We used the bypass-testing tool to evaluate four popular web applications. The results show some Since Offutt already studied the impact of robustness testing, we mainly illustrate some example of the robustness issues compared to security flaws, discovered using the tool.. All the tests were performed with and without the shield. As expected, by implementing all the constraints, the shield implements all the constraints was able to block all the invalid/malicious test data. The fact is that the shield enforces the same constraints used to generate the tests, and is thus a mirror of the test cases. However, duplicating constraints in an 'in-the-middle' shield may create an overhead. The results of conducted experiments are detailed in this section and provide evidence of the fact that the shield is a lightweight solution.

5.7.1 The five case studies and the bypass some examples of testing results

We chose to use four popular web applications. JForum and PhpBB are widely used to create forums (Table 27). The JForum website claims that this application is 'very secure'. However, we succeeded to find many security and robustness issues. To give an idea of the capacity of our testing tool to generate the test data. The last column shows the number of inputs automatically generated.

Table 27 - The four Web Applications

Web app.	# of lines of code& technology	# of form	# of inputs	# Inputs Generated
JForum	63870 (JSP)	33	223	1985
Roller	143865 (JSP)	39	271	2252
PhpBB	230286 (PHP)	35	192	1616
MyReview	53149 (PHP)	31	186	1889

The bypass testing results are shown in Table 28. Using the bypass testing tool, we were not able to discover any serious issue in both phpBB3 and WordPress. However, we were able to

find some robustness problems, especially in the JForum application. In fact, the tests provoked 353 failures related to three kinds of Java exceptions:

- Null Pointer Exception: A method call or use of a null variable. This occurred when null values were sent to the server.
- Class Cast Exception: incompatible class type cast. This happened when wrong values were sent instead of one the specified values (in the case when the user has to choose one value in an input select).
- Number Format exception: The server tried to convert a string into an integer. This occurred when non numeric values were sent to the server while numeric values were expected.

Table 28 – Bypass testing results

App.	#Failures	#SQL failures	#Null Response	Responses codes
JForum 2.1.8	Java: 353	1	0	[302, 404]
phpBB3	0	0	183	-
Myreview	0	1	650	-
Wordpress	0	0	0	[405,500]

These failures are due to bugs in the input validation code located in the server-side. The server did not check correctly the user inputs.
In addition, for two web applications (phpBB3 MyReview) we received 'Null responses'. The server returned empty responses.
Furthermore, according to the response code, there were three kinds of responses:

- Response 404: The requested page is not found. This occurred when hidden values were modified. The server uses them to reach certain kinds of pages. When the hidden value is not correct, this leads to the response 404.
- Response 405: The method is not allowed (using GET method when submitting a form instead of POST).
- Response 500: Internal server error.

We were able to find two SQL problems in MyReview and in JForum. The JForum one originated from a form used to submit new posts in the forum where the input subject length is not checked by the server side. When a long string is sent to the server an SQL Exception occurs and the SQL request is exposed to the end user.

5.7.2 Performance results

To measure the overhead due to the use of the bypass shield, we generated 50 instances of each form in JForum and run it with and without the shield in order to record differences in the execution time. We repeated this process ten times and calculated the average of execution times. The results presented in Table 29 show that overhead is 15%.

Table 29 - Performance results

The app.	With Bypass-Shield	Without Bypass-Shield
JForum	20 ms (overhead 15%)	17 ms

5.8 Summary

In this chapter, we presented a new approach that aims at automating the shielding and testing of web-applications against bypass attacks. The novelty of the approach resides in the analysis of the HTML code to extract constraints on the user inputs in addition to the JavaScript

validation code. These inputs are used to build a shield that executes as a reverse proxy to enforce these constraints.

This chapter provides an efficient solution for auditing web applications and for shielding and protecting them against bypass attacks. This approach targets securing the external part of the web application and completes the process of testing and validating access control policies.

However, these two approaches focus on testing existing and legacy systems. The goal is to be able to test, validate security or even shield an existing system. When building a new system from scratch, it is important to consider the issue of testing. New approaches should be used in order to improve the quality and testability of security mechanisms. The next two chapters will tackle this issue by providing new approaches for building and testing systems.

Chapter 6
An MDE process for specifying, deploying and testing access control policies

The previous chapters focused on testing the security in legacy or newly developed systems. Several issues were tackled concerning bypass testing and specially access control policy testing. Concerning this last issue, we proposed methodologies for test generation, test transformation and detection of hidden security mechanisms. For bypass testing, we presented how to leverage the bypass testing to protect and shield web application.
All these approaches are to be applied to existing systems. However, when building new systems it is important to consider how the security mechanisms should be build in order to facilitate the testing step, and hence make it easy for testers to eventually find problems and fix them. Ideally, there should be no need to testing. This could be reached only using a fully automated approach for generation and integration of the security mechanisms.
In the next two chapters, we consider the issue of building and integrating access control mechanisms into applications. As a first take on the problem, a new approach is proposed that allows access control policy to be automatically specified and deployed. Integration of the policy is manually woven in the application code using aspect oriented programming (AOP). The approach is model-based and takes into account the testing process at the early stage of the development lifecycle. In addition, it provides a generic certification process that could be applied for various kinds of access control formalisms.
This chapter introduces this model-based approach. It targets limiting the risk and improving the testability of the system by proposing a model-driven approach for deriving access control mechanisms.

6.1 Introduction

Ensuring confidence in the implemented security mechanisms is the key objective when deploying a security concern. This objective can be reached by improving the design and implementation process via automation, such as security code generation from models, and by systematic qualification criteria. Qualification criteria can be used for *a priori* verification of the specified security models' consistency, and for *a posteriori* validation of the quality of test cases executed on the final implementation. Qualification criteria should be independent of the many languages used to model security and – if possible – from the specificity of implementation languages. The same principles and same criteria should be applied to qualify the security mechanisms for various kinds of systems. Although such common principles and criteria are highly desirable, they are useless without the support of effective tools that make the qualification approach applicable in a final system implementation.
Several studies [64, 65, 70] have proposed to address access control policies in a model-driven way, but none of them study how to deal with several access-control formalisms in a unified way, especially to build a common qualification process. This chapter proposes an original model-driven process to implement and qualify an access control policy, independently from the specific formalism used to express this access control policy. This process covers two dimensions: (1) a generative process for deriving security components from the specification of the access control policy, (2) a generic method for the test qualification of access control mechanisms.

The *first dimension* of the approach thus targets the automation of the process of transforming an abstract model of the access control policy into executable security components that can be plugged in the application. Recall that the standard architecture for a secured application involves designing the policy decision point (PDP), which can be configured independently from the rest of the implementation containing the business logic of the application. Finally, the execution of functions in the business logic includes calls to the PDP by PEPs. A mapping between the security concepts and the application specific concepts in required for the PEPs to perform the suitable calls to the PDP in order to enforce the policy.
The proposed MDE process is based on a domain-specific language (DSL) in order to model security formalisms/languages as well as security policies defined according to these formalisms. This DSL is based on a metamodel that captures all the necessary concepts for representing rule-based access control policies. The MDE process relies on several automated steps in order to avoid faults that can occur with repeated manual tasks. This includes the automatic generation of a specific security framework, the automatic generation of an executable PDP from an access control policy and the injection of PEPs into the business logic through aspect-oriented programming. The identification of the application's functionalities that need to be secured is manual, but AOP mechanisms automate the injection of PEPs at these specific places.
In this context, the challenge for qualification is to offer some guarantee that the PDP is consistent and that it interacts with the business logic as expected (e.g. hidden security mechanism may bypass some PEPs as shown in chapter 4). Since this interaction of the PDP with the business logic can be faulty, a second dimension is needed to improve the confidence in security mechanisms.
This *second dimension* consists of a qualification environment which provides (1) basic *a priori* verifications of security models before PDP component and PEP generation, (2) *a posteriori* validation of the test cases for the implementation of the policy. This qualification environment is independent of access control policy languages and is based on a metamodel for the definition of access control policies. It provides model transformations that make it possible for the qualification techniques to be applied to several security modeling languages (e.g. RBAC, OrBAC, DAC or MAC). The interest of metamodeling V&V artifacts (structure and semantics) is that the same certification process is applied for testing the system even when it combines policies expressed in different languages.
Next section presents the MDE process. Then, we detail the security framework based on a core security metamodel. Afterwards, we explain how this metamodel is used for security code generation (PDP and PEPs). Then we show the V&V certification process. The mutation operator, that we previously presented in the previous chapters are defined on the metamodel: the fault model is thus shared by any access control language, instance of the core metamodel. By construction, the validation is based on a common certification basis. Finally, we present the empirical results on three Java case studies and discuss the advantages and drawbacks of using a metamodel instead for specific language-dependant platforms.

6.2 MDE for Security

Errors in the access control policy can have several causes: an error in the policy definition; an error when translating the policy into an executable PDP; an error in the definition of PEPs (calls to the PDP at the wrong place in the business logic, calls to the wrong rule, missing call, etc.). In order to increase the efficiency of the validation task, we investigate an integrated approach based on two facets. First, we want to ensure as much quality as possible by construction. For that purpose we propose an MDE process based on a specific modeling language for security policies and automatic transformations to integrate this policy into the

core application. Second, we present a validation technique that can be applied early in the development cycle and that is independent of any particular security formalism.

6.2.1 Overview of the modeling architecture

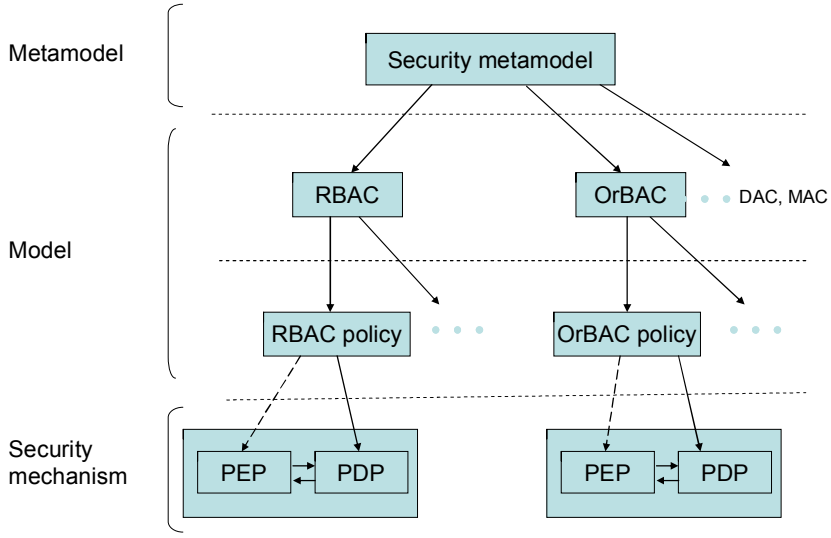


Figure 47 - The modeling framework

Figure 47 depicts the different levels of models used in our approach. The access control metamodel captures the common concepts used to define existing rule-based access control formalisms (such as RBAC, OrBAC access control languages). The core security metamodel can be instantiated to define an access control policy formalism and the expression of rules in this formalism. A specific access control policy for a given system can thus be expressed, and automatically exported to generate the PDP component which embeds this access control policy. The PEP is then manually created to weave the calls to the PDP in order to enforce the access control policy.

6.2.2 MDE security design process

Several approaches exist which promote the use of models for building a secure system. First, they aim at expressing access control or more general security concepts on a model of the system, for instance by using specific UML profiles [69] or by weaving security access control into the final design model itself [70]. We address the same overall issue, but we consider that model-driven security should be compatible with the most used access control languages, especially for proposing a unified qualification scheme. Instead of weaving the security elements in the design models (e.g. class diagram), we aim at maintaining a clear separation of the PDP and the business model. Figure 48 presents an overview of the proposed approach. From top to bottom, the process starts with the requirements for the system and ensures the integration of the security concerns in the running code.

The first step of the approach (1) is to build the security model for the application. The security model is a platform independent model which captures the access control policies defined in the requirements of the system. To allow security experts to create, edit and check this security model, it is modeled in a security domain specific modeling language. This language is based on a generic metamodel which allows several types of rule-based access-control policy to be expressed. In practice, the meta-model allows the type of rules to be modeled as well as the rules themselves.

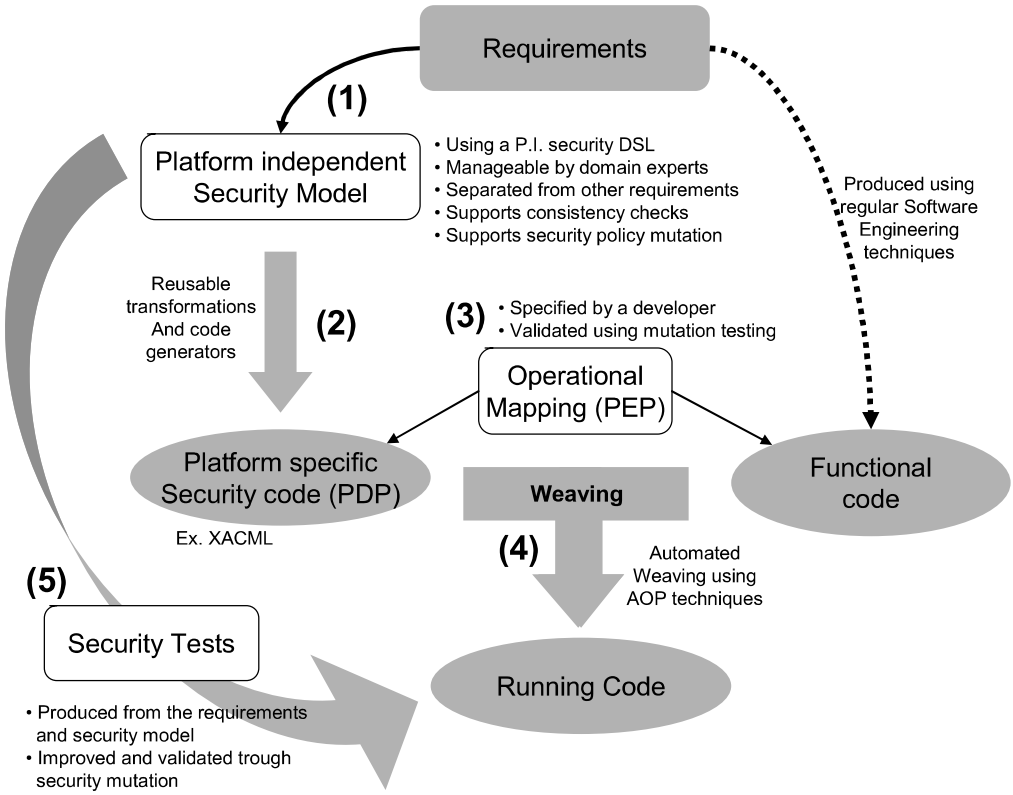


Figure 48 - Overview of the model-based approach

Once the platform independent security model has been validated, automated transformations are used to produce a PDP specific to the system to be secured (2). In practice, the PDP is not usually fully generated but based on reusable security frameworks. These frameworks are configured with the specific access control policy and connected to the application through the PEP. In the example presented in the figure, the output of the transformation is an XACML file that contains the access control policy.

A critical remaining step for implementing the security of the application is to connect the security framework with the functional code of the application (3). In practice, this corresponds to adding the PEPs in the main code of the application. It is a critical step because any mistakes in the PEP can compromise the overall security of the application. This step is manually performed using AOP to make the security PEP introduction systematic (4). Then, mutation testing is used to ensure that the final running code conforms to the security model (5). An important reason to compose the security concern at the code level is to limit the assumptions on how the system is developed prior to this composition. Another reason is that this makes it possible to change the access control policy after the deployment.

The use of aspect-oriented programming allows a separation to be kept between the business logic of the application and the security code. The use of *pointcuts* allows calls to be introduced to the security code systematically without having to list all the specific locations. This makes the PEP easier to specify and understand which avoids many potential mistakes. However, the use of AOP does not provide a warranty that the access control policy is appropriately integrated into the code of the application. A careful validation of the resulting code with respect to the security model is required.

6.2.3 Validation of access control mechanism

The proposed approach aims at building a secure application, with integrated security access control mechanisms. We propose validation of the code through testing to ensure the correctness of the security mechanisms.

The final running code is tested with security-specific test cases. To properly validate the security of the application, these test cases have to cover all the security features of the application. We use mutation analysis to qualify tests.
Compared to the way we used mutation previously, the originality of this proposed approach is to perform mutations on the platform independent security model using generic mutation operators. Since the transformation and weaving of the access control policy in the application are partially automated, the tests can be automatically executed on the mutants of the application. If the tests are not able to detect a mutant then new test cases should be added to exercise the part of the access control policy which has been modified to create this mutant. In practice the undetected mutants provide valuable information for creating new tests and effectively validating the access control mechanisms.
The objective of the test cases in a partly automated generation process (PDP and PEPs) is to check that the access control mechanisms of the application are fully synchronized with the access control policy. Indeed, in the mutation testing process, the policy is modified and the tests are required to detect the modification in the running code. Thus, if any of the access control rules are hard-coded in the application or if the application code bypasses the security framework, the modification will not be observable in the running code. In that case, the mutants corresponding to the hard-coded or bypassed security feature are not detected and the tester has to fix the application to make sure that all access control rules are correctly enforced.
Because testing is performed on the final running code, it validates both that the PDP is consistent with the model but also that the PEPs, i.e. the integration with the rest of the application, are correct. The following sections detail the main steps of the approach, beginning with the security metamodel and its instances, then focusing on the MDE security design process and finally on the validation process.

6.3 The access control metamodel

In the literature, a multitude of access control formalisms were defined such as RBAC, OrBAC, MAC [28, 29] or DAC [30], which are all based on the definition of policies. All these formalisms allow the definition of rules that control the access to data, resources or any type of entity that should be protected. The formalisms differ by the type of rules and entities they manipulate. In order to provide generic validation mechanisms, this work is based on a generic policy metamodel which allows capturing both the specificities of particular access-control formalism and the corresponding access-control policies. This section introduces this generic access-control metamodel and illustrates how it supports the definition of RBAC, OrBAC, DAC and MAC policies.

6.3.1 Generic security metamodel

Figure 49 displays the proposed meta-model. The meta-model is divided into two parts, which correspond to two levels of instantiation:

- The first level of instantiation (classes POLICYTYPE, PARAMETERTYPE and RULETYPE) allows modeling particular policy formalisms such as (RBAC OrBAC, MAC or DAC). A POLICYTYPE defines a set of parameter types (PARAMETERTYPE) and a set of rule types (RULETYPE). Each rule type has a set of parameters that are typed by parameter types.
- The second level (classes POLICY, RULE and PARAMETER) allows instantiating a specific access control policy using a defined formalism. A POLICY must have a type (an instance of POLICYTYPE) and defines rules and parameters. The type of a policy constrains the types of parameters and rules it can contain. Each parameter has a type which must belong to the parameter types of the policy type. If the *hierarchy* property of the parameter type is true, then the parameter can contain children of the same type as itself. Policy rules can be

defined by instantiating the RULE class. Each rule has a type that belongs to the policy type and a set of parameters whose types must match the types of the parameters of the type of the rule.

In practice, the two parts of the metamodel have to be instantiated sequentially: first define the formalism, and then define a policy according to this formalism.

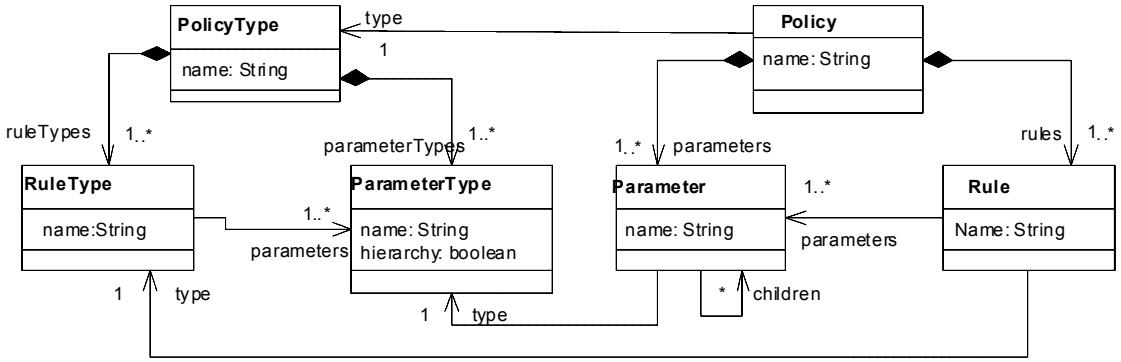


Figure 49 - The meta-model for rule-based security formalisms

6.3.2 Instantiating the metamodel

The fact that the proposed meta-model supports both modeling policy formalisms and policies in these formalisms makes it possible to represents any rule-based access control policy. Rule-based formalisms have to be modeled using the part of the meta-model which captures policy types in order to be supported. This has to be done once for each one of the formalisms. This way, it is possible to model corresponding access control policies with the policy part of the proposed metamodel. This section demonstrates how four existing security formalisms (OrBAC, RBAC, MAC and DAC) have been modeled using the proposed approach.

Modeling OrBAC

We redefine the OrBAC formalism as an instance of the generic metamodel and then we define an OrBAC access control policy.

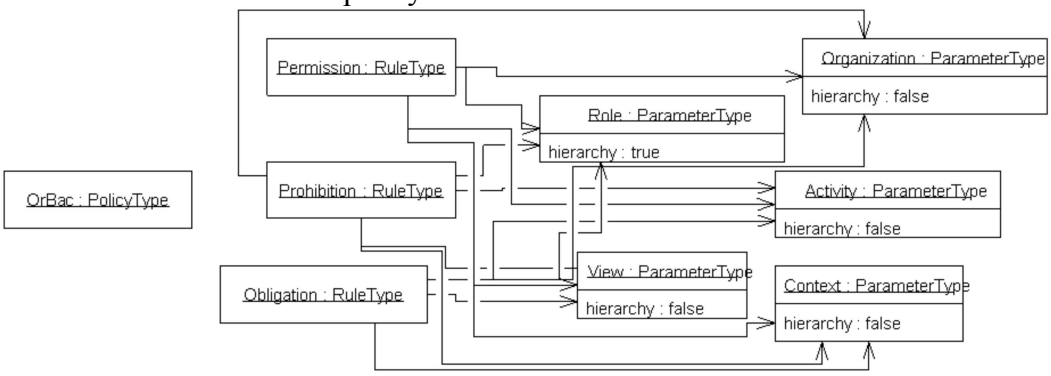


Figure 50 - The OrBAC security formalism

Figure 50 shows how the metamodel was instantiated to model OrBAC access control policies.

To illustrate our approach, we reuse the example of the library management system. We use a simplified version of the application with just five security rules as defined below:

POLICY LibraryOrBAC (OrBAC)

R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Permission(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

Figure 51 shows an instance of the metamodel for rule R1. Rule R1 is well-formed since it has five parameters, each of them of the correct type: Organization, Role, Activity, View, and Context. `Borrower` and `BorrowerActicity` are hierarchies of parameters. Rule R1 uses the sub-parameter `Student` defined in the `Borrower` hierarchy and `Borrow` in the `BorrowerActivity` hierarchy.

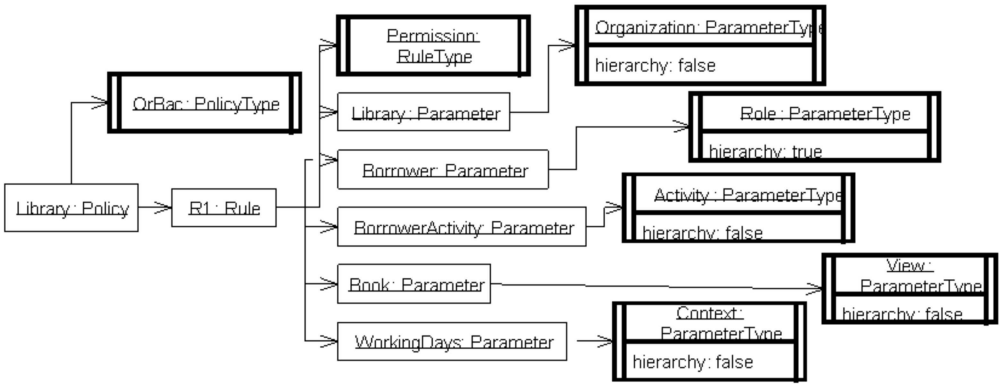


Figure 51 - Rule R1 for the library access control policy

Modeling RBAC

Figure 52 presents the RBAC model as expressed using our metamodel. In the same way as for the OrBAC model, the PolicyType class is instantiated to model RBAC. We consider RBAC model augmented with notion of constraints over permissions. Four types of entities are defined: users, permissions, roles and constraints.

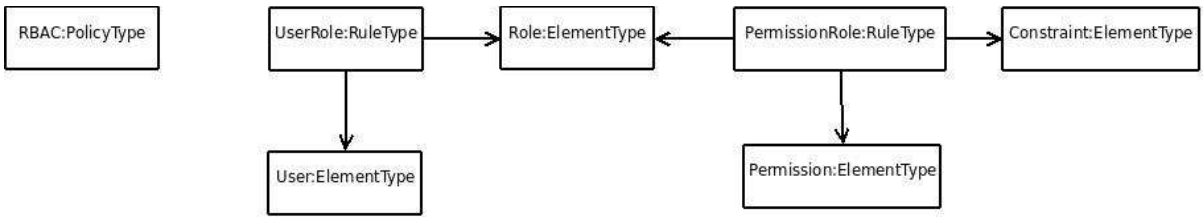


Figure 52 - RBAC model

A similar access control policy was modeled based on RBAC. It includes:

- Three users: alice, yves and romain.
- The same four roles as the OrBAC model.
- Three permissions: BorrowBook, ModifyUserAccount and CreateUserAccount.
- Two constraints: WorkingDays and Holidays.

Six rules were defined to associate users with roles on one hand and permissions with roles on the other hand:

POLICY LibraryRBAC (RBAC)

R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

According to the standard [27], the RBAC model we consider in this work, is flat RBAC augmented with constraints. RBAC may include other features such as SOD (Separation of Duty). Our approach allows us to express SOD, either static separation of duty (SSD) or dynamic separation of duty (DSD). Our model allows expressing rules corresponding to SOD. However, the generated policy in XACML has to take into account these rules.

Modeling DAC

We consider DAC as used for file systems. Objects include files, directories or ports (or others) and Subjects include users or processes. The policy can be seen as a matrix where the values are access types. Access types include three access types (r: read, w: write and x: execute) and two special ones, which are *control* and *control with passing ability*. The control access type enables its holder to modify the users' access types to that object. In addition to this, the control with passing ability enables the user to pass this control ability to other users.
The access types of the DAC:

- r : permission to read the object.
- w: permission to write.
- x: permission to execute.
- c: control permission, the ability to modify 'r w x' permission for this object.
- cp: control and the passing ability of control.

Figure 53 shows the DAC modeled using our generic metamodel. There is only one type of rule. This rule contains a Subject, an access type (r: read, w: write, x: execute, c: control or cp: control and passing the control ability) and an object (a file or a port etc.).

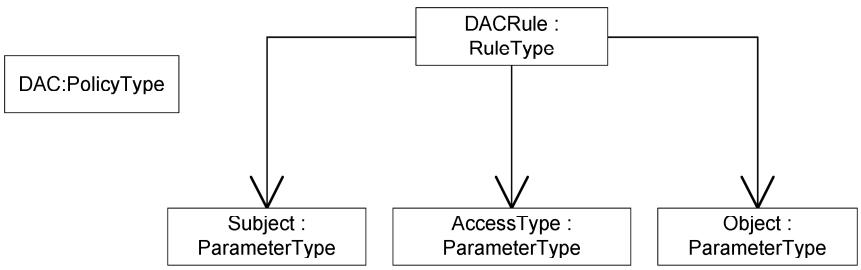


Figure 53 - The DAC formalism

Here is a simple example of DAC policy. The following policy defines two subjects (Tim and Admin). Tim can read or execute file1, while admin has the right to read, write and execute the file in addition to the control and passing ability.

POLICY systemDAC (DAC)

R1 -> DACRule(Tim r file1)
R2 -> DACRule(Tim x file1)
R3 -> DACRule(Admin cp file1)
R4 -> DACRule(Admin r file1)
R5 -> DACRule(Admin w file1)
R6 -> DACRule(Admin x file1)

Modeling MAC

We consider MAC (Mandatory Access Control) policies as they are used in multi-level systems (MLS) [135].

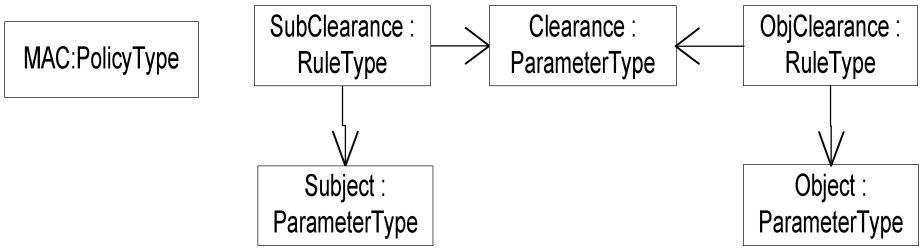


Figure 54 - The MAC formalism

Figure 54 displays the MAC metamodel. There are two types of rules:

- SubjClearance: An association between a subject and a clearance.
- ObjClearance: An association between an Object and a clearance.

There is a static classification of clearance. According to this classification, access is granted to subjects (to read or write). We consider two access types: read and write.
Here is an example of a MAC policy. The following policy defines two users and two objects and the rules specify their clearances.

POLICY systemMAC (MAC)

R1 -> SubClearance(process1 low)
R2 -> SubClearance (process2 high)
R3 -> ObjClearance (report1 low)
R4 -> ObjClearance (report2 high)

6.4 Description of the security mechanisms

In this section, we describe the security framework implementing the access control policy. To illustrate this, we focus on Role-based access control policies, which are the most relevant for service-oriented applications. The security framework includes the PDP which interacts with the PEPs. The PEP is manually woven into the application code. The PDP includes the policy that is implemented in XACML. The policy is automatically generated during the design step.

6.4.1 The architecture of the security mechanism

Figure 55 depicts the overall architecture of the security architecture. The PDP encapsulates the XACML policies and offers services to access the policy. The PEPs are implemented in the application code. A mapping between the application concepts (user roles, methods, resources and contexts) and the OrBAC/RBAC policy concepts (roles, activities, views and contexts, or roles, permissions and constraints) is required to make the interactions feasible.

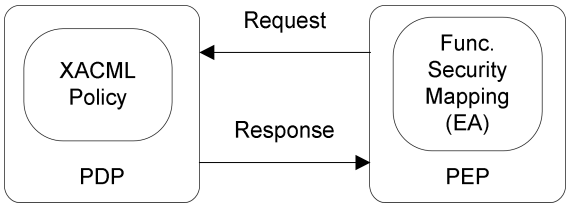


Figure 55 - PEP and PDP

The PEP requires the mapping between the application concepts (user roles, methods, resources and contexts) and the OrBAC abstract policy (roles, activities, views and context) to get the OrBAC abstract entities. Then it sends a request to the PDP containing the role, the activity, the view and the context. The PDP responds with permit or deny.
The concrete classes that compose the security framework are presented in the Figure 56. The framework makes the connection between the application (business logic) concepts and the security concepts used in the access control policy language.

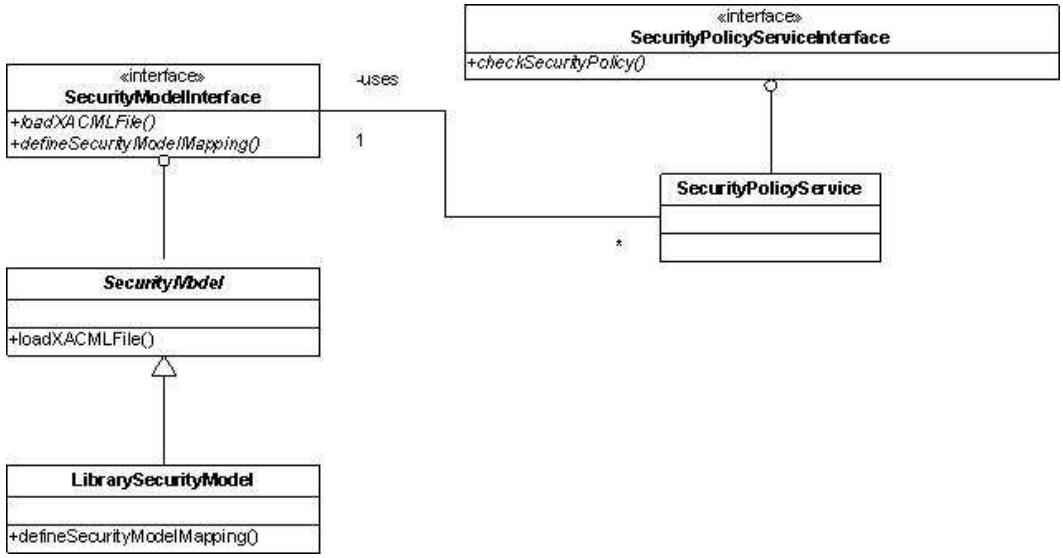


Figure 56 - The security framework class diagram

The PDP involves an access control policy service and a security model. The access control policy service is implemented by "SecurityPolicyService" which contains the method "checkSecurityPolicy" that is in charge of receiving requests from a PEP and responding with the suitable access control decision based on the policy. The security model is implemented by the SecurityModel class that contains the policy (stored in the XACML file). This class handles the access to the XACML files and handles the XML requests and responses.
The mapping between the application and the OrBAC/RBAC policy concepts is done in the SecurityModel subclass (in this example it is the LibrarySecurityModel). This mapping is used by the PEP before performing the call to the access control policy service.
The PEP is woven into the application using AOP (with AspectJ). An aspect is composed of two main parts: (1) advice that implements the behavior of the cross-cutting concern, and (2) the pointcut descriptor (PCD) that designates a set of joinpoints in the base program where the advice should be woven. The weaving is performed automatically by a specific compiler. In the context of the development of secured applications, we propose to define PEPs as aspects, as shown in the following listing. The call to the PDP is woven before the execution of the method. If the access is granted, the execution continues, otherwise a security exception must be raised. The PEPAspect aspect (implemented in AspectJ) shown bellow, defines an advice

that is woven before each execution of BookService.borrowBook. This advice calls the checkSecurity method that will then call the PDP by sending the role, activity, view and context given in the parameter. Note that the advice throws a SecurityPolicyViolationException. This exception is raised by checkSecurity when the PDP responds with a prohibition.
To sum up, the weaving is semi-automated since the location of the service to be secured has to be expressed manually in a joinpoint, while the advice is a generic call to the security framework.

```
public aspect PEPAspect {
// PEP Joinpoint for borrow
before(User user,Book book) throws
SecuritPolicyViolationException :
borrowBookCall(user,book)  {

// Call to check for security rule
checkSecurityPolicy(user.getRole(), BORROWMETHOD ,BOOKVIEW,
getTemporalContext());
     }
}
```

6.4.2 PDP XACML code generation from the metamodel

We implemented in our tool *securityMDK* [136] the ability to generate XACML (Extended Access Control Markup Language) policies from an instance of the metamodel. The XACML file is added to the PDP. XACML is an OASIS standard dedicated to defining access control policies in XML files. Among the existing frameworks (the sun XACML implementation and Google enterprise Java XACML), we chose the sun XACML tool [137] that supports and helps to access, write and analyze XACML files. Writing XACML files manually is an error-prone task, due to the language complexity. We use the existing XACML profiles for RBAC and OrBAC to generate XACML files from our metamodel.

Generating XACML files
XACML was not defined according to a specific access control model (RBAC, MAC, DAC or OrBAC). Therefore, some adaptation is necessary to use XACML for RBAC or ORBAC policies. There exists actually an XACML profile for RBAC defined by OASIS [43] and for OrBAC [44] which is actually an extended version of RBAC profile. These profiles define the way an RBAC or an OrBAC policy can be expressed using the XACML language.
We illustrate our approach for generating XACML files using OrBAC profile. The profile defines two way policies, requests and responses are expressed and how concrete entities are mapped to the abstract ones (which corresponds to the operational mapping).
As a first step, the tool generates the XACML policy file containing the abstract OrBAC policy as specified by the profile. Table 30 shows the mapping between the OrBAC entities and the XACML elements:

Table 30 - Mapping between OrBAC entities and XACML elements

OrBAC	XACML
Role	Subject
Activity	Action
View	Resource
Context	Environment

Limitations
Depending on the underlying model (OrBAC or RBAC), the appropriate profile is used to generate XACML files. Therefore the XACML generator is not generic and a specific generator is needed for each access control policy language. However, it can be reused because it is defined at model level.
It is important to note that the tool allows XACML files to be generated for both the actual policy and the mutant policies, as it will be detailed in the following section. These mutants are generated at generic level and the generator tool is used to create XACML files for these mutant policies. Thanks to this *SecurityMDK* tool, the mutant PDP is generated, the mutation analysis becomes a 'push-button' technology. The mutant systems are automatically generated from the semantics of the mutation operators defined on the security metamodel. Then they are exported to XACML to be used by the PDP to evaluate the existing security tests. Therefore, our approach offers a powerful framework for evaluating access control policy tests using mutation independently of the underlying access control language.

6.5 Validation of the access control mechanisms

The verification of the consistency of a given security policy is performed at early stage, during the specification by security experts, who express who can access to what. The validation process then aims at qualifying the test cases used to ensure the correctness of the security mechanisms. The fault model is defined at the meta-level. Based on this fault model, mutants are generated to simulate faults in the access control policy.

6.5.1 The verification checks

Verifications are performed in order to check the soundness of the security policy and its adequacy with regards to the requirements of the application. In particular, we are able to detect conflicts between rules. Detecting conflicts is a tedious task when done manually. A simple case of conflict occurs when a specific permission is granted by a rule and denied by another. These verifications also include checking that each business operation can be performed by at least one type of user or that a specific set of operations can only be performed by specific users (e.g. administrators). All these verifications are carried out by the security metamodel and are thus language independent: verifications and transformations are generic and apply to all types of rule. The main verification is offered by construction, when the conformity of a security model to its metamodel is checked. In this way, we have a minimum consistency that is obtained with the conformance relationship of a model to its metamodel. We also add some extensible verification functions. They constitute preconditions which are checked before deploying the policy and generating the mutants. Three verification functions are implemented:

- policy_is_conform(): the policy conformance to the underlying policy type (OrBAC or RBAC etc.). In order to guarantee that the defined rules meet the types of parameters of rules defined by the policy type. The flexibility of the model can lead to having incorrect rules which do not conform to the types of rules supported by the policy.
- no_conflicts(): checks the absence of conflicts. It essentially involves checking that there are no rules having the same parameters and having different types. This is especially useful for OrBAC where both prohibition and permission rules can be defined.
- no_redundancies(): checks that the security policy is minimal, which means that no rule appears more than once. This inconsistency can happen when high level hierarchy parameters are used to specify rules in addition to the others descendant parameters, leading to have two or more of the same rule for that descendant parameter.

The conformance verification function detects simple erroneous rules such as wrong parameters or wrong numbers of parameters.

This verification step is important in order to detect faults in the specified policies. These faults are detected early during the modeling and are corrected before the deployment and the generation of the XACML policy. XACML is an OASIS standard [138] for expressing policy using the XML language.

6.5.2 Mutation analysis applied to access control testing

The validation process aims at qualifying the test cases used to ensure the correctness of the security mechanisms. The validation process is based on mutation analysis that we have previously presented (see Chapter 3).

Figure 57 shows the MDE validation process. To have a common certification process, we generate mutants from mutation operators defined on the metamodel. The idea is to extend the security metamodel with the definition of mutation operator, using Kermeta [139]. Kermeta is an open source metamodeling environment that is fully integrated with Eclipse. It has been designed as an extension to the meta-data language EMOF [140] with an action language that allows specifying semantics and behavior of metamodels. An instance of such a metamodel automatically embeds this semantics and behavior: this is this facility offered by Kermeta. This facility allows us to define the mutation operators' semantics at meta-level.

The access control policy languages (such as RBAC on the figure, or any other access control language) are instances of the metamodel and thus embed the way a specific access control policy can be mutated. Code generation from the mutants security policies create as many faulty PDPs as there are mutant policies. The test qualification process can then be applied on the set of faulty systems, embedding mutant PDPs.

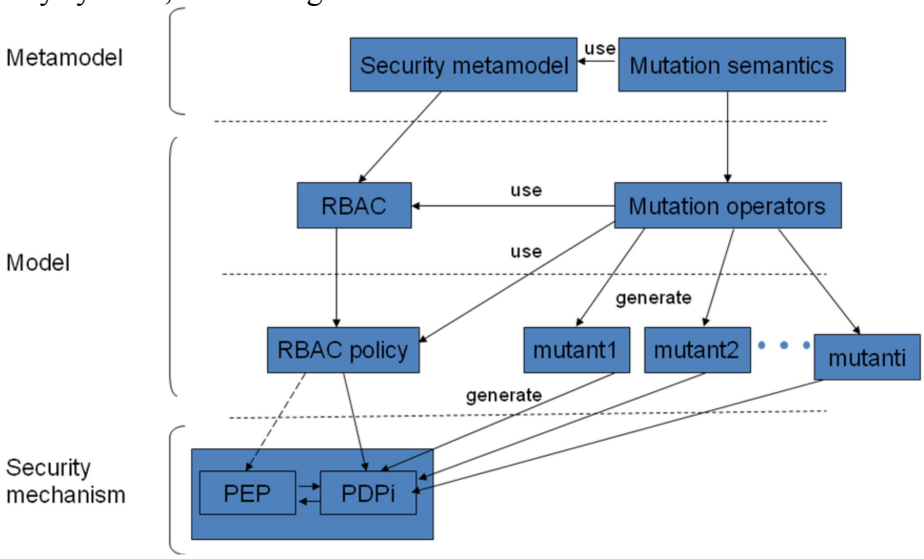


Figure 57 - MDE validation process

We define five mutation operators for access control policy testing, shown in Table 31. These operators are defined only in terms of the concepts present in the security metamodel, which means that they are independent of specific access control formalism. Thus, these operators can be applied to inject faults into any policy expressed with any formalism that can be defined as an instance of our metamodel. The definition of mutation operators at this meta-level is critical for us since it allows the qualification of test cases with the same standard, whatever the formalism used to define the policy.

Table 31 - The mutation operators

Operator Name	Definition
RTT	Rule type is replaced with another one
PPR	Replaces one rule parameter with a different one
ANR	Adds a new rule
RER	Removes an existing rule
PPD	Replaces a parameter with one of its descending parameters

In order to generate faulty policies according to these operators, we have added one class to the metamodel for each operator:

- **RTT:** Finds a first rule type that has the same parameter as the type of another rule type. Then it replaces the rule parameter of one rule having the first rule type with the other rule type.
- **PPR:** Chooses one rule from the set of rules, and then replaces one parameter with a different parameter. It uses the knowledge provided by the metamodel (by ruleType and parameterType classes) about how rules are constructed.
- **ANR:** Uses the knowledge about the defined parameters and the way rules are built. Then it adds a new rule that is not specified.
- **RER:** Chooses one rule and removes it.
- **PPD:** Chooses one rule that contains a parameter that has descendant parameters (based on the parameter hierarchies that are defined) then replaces it with one of the descendants. The consequence here is that the derived rules will be deleted and only the rule with the descendant parameter remains.

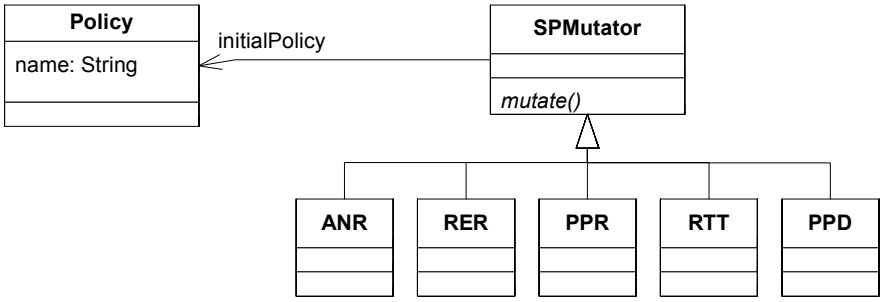


Figure 58 - Extension of the security metamodel with mutation operators

The Kermeta action language is imperative and object-oriented and is used to provide an implementation of operations defined in metamodels. It includes both OO features and model specific features. Convenient constructions of the Object Constraint Language (OCL) such as closures (e.g. each, collect, select) are also available in Kermeta. Figure 58 shows the operator classes. Figure 59 shows the implementation for the RER operator. The `mutate()` method is implemented in Kermeta. What is important to notice in this method is that it is defined only using concepts expressed in the metamodel. Thus, this method can generate a set of mutated policies, completely independently of the formalism they are defined with.

The impact of the mutation operator depends on the access control formalism used to define a policy. The faults that are simulated can be very different. The same operators emulate very different flaws in the policies. For instance, the ANR operator applied to RBAC simulate the adding a new permission, while for OrBAC it will simulate adding a new prohibition or a new permission. In addition, the RER operator simulates adding a new prohibition when used for an RBAC policy, but may lead to removing a permission when used with an OrBAC policy. The impact of the operator depends on the semantic and the logic of the access control model.

```
class RER inherits SPMutator {


    method mutate(p : Policy) : set Policy[*] is do
        var mutant : Policy
        result := Set<Policy>.new
        // loop on rules
        p.rules.each{ r |
            // create mutated policy
            mutant := p.copy
            mutant.name := p.name + "-RER-" + r.name
            // remove one rule
            mutant.rules.remove(mutant.rules.detect{x | x.name == r.name})
            // adds the mutant policy
            result.add(mutant)
        }
    end
}
```

Figure 59 - The RER operator

6.5.3 Generated mutants

In order to illustrate the mutants we get when we apply our approach, we show examples of mutants for OrBAC, RBAC, DAC and MAC.

OrBAC mutants

We show here some examples of mutants obtained for an OrBAC policy:

RTT mutant

POLICY LibraryOrBAC-RTS-R4-Prohibition (OrBAC)
R1 -> Permission(Library Student Borrow Book WorkingDays)
R2 -> Prohibition(Library Student Borrow Book Holidays)
R3 -> Prohibition(Library Secretary Borrow Book Default)
R4 -> Prohibition(Library Personnel ModifyAccount UserAccount WorkingDays)
R5 -> Permission(Library Director CreateAccount UserAccount WorkingDays)

These mutants simulate errors in the OrBAC policy. In this example, RTT replaces rule status which simulates denying authorized access.

RBAC mutants

Next, we present some examples of mutants related to an RBAC policy. The initial RBAC policy is presented bellow:

POLICY LibraryRBAC (RBAC)

R1 -> UserRole(romain Student)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount
WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

Here are some examples of the generated mutants

PPR mutant:

POLICY LibraryRBAC-RDD-R1-Student-Personnel (RBAC)
R1 -> UserRole(romain Personnel)
R2 -> UserRole(yves Director)
R3 -> UserRole(alice Secretary)
R4 -> RolePermission(Student BorrowBook WorkingDays)
R5 -> RolePermission(Personnel ModifyUserAccount WorkingDays)
R6 -> RolePermission(Director CreateAccount AllTime)

In this example, the PPR mutant replaces R1 parameter and modifies Romain's role, allowing him to have the same access right as the personnel.

DAC mutants

Some mutation operators cannot be applied to DAC policies. In fact, the RTT and PPD operator cannot be used since there is only one type of rule, and no hierarchy.
It is interesting to study the impact of the three mutation operators. For instance the RER operator will remove R1 resulting in a mutant policy that implies that Tim will no longer have the right to read "file1". RER operator will produce 5 mutant policies, as there are 5 rules. The PPR operator will replace one of the rule parameter with a different one. One example of its mutant policies will be the one containing R1':

DACRule(Tim w File1)

The mutant policy enables Tim to write in "File1" but denies him reading this file.
The ANR operator will produce mutants by adding one new rule to policy. One possible mutant is the one containing this new rule:

DACRule(Tim cp File1)

This will result in granting Tim the control and the passing ability.

MAC mutants

As for DAC policies, the mutation operators PPD and RTT cannot be applied in this case. The RER operator is not relevant either because it would create undefined policy responses. For instance, if R1 is removed, the subject 'process1' clearance will be unknown, resulting in undefined policy decision.
The relevant operators are PPR and ANR. The PPR operator will for example replace R1 second parameter with another one, which will produce this rule instead of R1:

R1' -> SubClearance(process1 high)

This implies process1 having a high clearance. This simulates a flaw in the access control policy.
The ANR operator adds for example this new rule:

R5 -> SubClearance(report1 high)

This new rule is with conflict with the R3. So, the result depends on the implementation of the security mechanism, on the way it handles conflicts. If priority is given to most restrictive rule, then this implies report1 having high clearance.
Next section presents the experiment of our MDE process. Two main points are studied: the *feasibility* of the approach through three case studies and the *relevance* of the validation technique compared to language-specific approaches.

6.6 Case studies and results

We applied the MDE security design and validation processes on three systems used in the previous chapters:

- LMS: A Library Management System.
- VMS: A Virtual Meeting System.
- ASMS: An Auction Sale Management System.

6.6.1 Validation results

The validation of the PDP interacting with the business logic was done for the three case studies. The objective of these empirical studies is twofold:

1. Is the approach feasible and what results do we obtain when we modify the access control policy language?
2. What are the differences between MDE and language specific approaches?

To answer both questions, we have an existing basis, which contains the functional tests and the security tests which were generated to validate the specific OrBAC-based mutation approach (see chapter 2). So we have two sets of test cases: functional test cases and access control policy test cases.

Examples of tests

In order to understand the difference between functional and security tests, we show two examples of tests. The test scenario aims at testing, for the LMS system, a teacher borrowing and returning a book.
Although the testing scenario which includes the data initialization and the system calls is the same for the functional and the security test cases, the intent of the tests differ resulting in different testing *oracles*.
The testing oracle aims at checking whether the test succeeds of fails by verifying that the system behaved as expected.
The two oracles are different. On one hand, the security oracle will check that the access is granted and that the right security rule is activated using the right parameters. On the other hand, the functional oracle checks that the system state is updated and that data are correctly stored in the database. The security mechanism implicitly impacts the execution of the functional testing scenario. If the access is not granted, this leads to the failure of the functional tests.

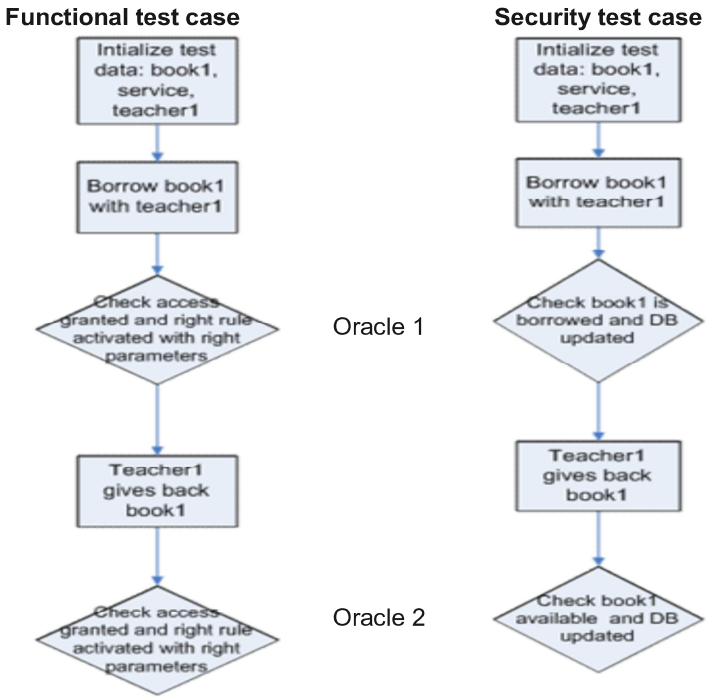


Figure 60 - Examples of security and functional test

Process feasibility and results

For the first question, we applied the full design and generation process for the three case studies, with RBAC and OrBAC access control languages. The feasibility of the approach is thus demonstrated by these successful deployment and validation steps. The limit of this feasibility analysis is that the targeted systems are homogeneous, which means that the final execution platform is based only on Java. The number and categories of mutants that are generated for both access control policy languages are given in Table 32. In this table we separate ANR mutants from non-ANR because they both generate very different types of mutant: non-ANR operators mutate the existing rules whereas ANR adds new rules. It is interesting to note that the number of mutants may vary significantly depending on the language chosen. However, based on this observation, we cannot conclude which language is likely to be error-prone.

Table 32 - Number of mutants generated for RBAC/OrBAC policies

	RBAC				OrBAC			
System\ mutants	rules	NON ANR	ANR	All	rules	NON ANR	ANR	All
LMS	23	437	257	694	42	330	714	1044
VMS	36	972	396	1368	106	426	1046	1572
ASMS	89	2937	647	3584	130	1138	1950	3088

Figure 61 presents the comparative mutation scores (percentage of mutants that are detected as faulty by the test cases) when we execute functional and security test cases on both implementations. We can thus compare how efficient each category of test generation is by using this qualification technique. Here, efficiency is evaluated by the ability of test cases at detecting specific types of faults. OrBAC non-ANR mutants (basic mutation operators) are more difficult to detect with functional test cases than RBAC mutants. This is interesting, since it shows that qualifying functional test cases using OrBAC may be a good choice. Clearly, security test cases are qualified for both languages. Second, concerning ANR

mutants, which qualify whether the test cases exercise the “by default” policy of a system, it appears that functional test cases are not efficient, whatever the access control policy language is. Circles pinpoint the very low mutation scores obtained especially with OrBAC policies. This is partly due to the fact that some ANR mutants we generate for OrBAC do not modify the behavior of the application, in other words, they are *equivalent mutants*. However, even after filtering equivalent mutants, both functional and security test cases obtain low scores. In the case of the LMS system, the differences in terms of quality between functional and security test cases for RBAC are quite visible (3% compared with 95 % mutation scores). Test cases sets that do not reach a high mutation score have to be completed to be qualified. In a previous work [11], we presented the real security faults we found in these cases studies, especially hidden security mechanisms. In summary, a same fault model leads to the injection of specific faults, which shows the interest of the technique.

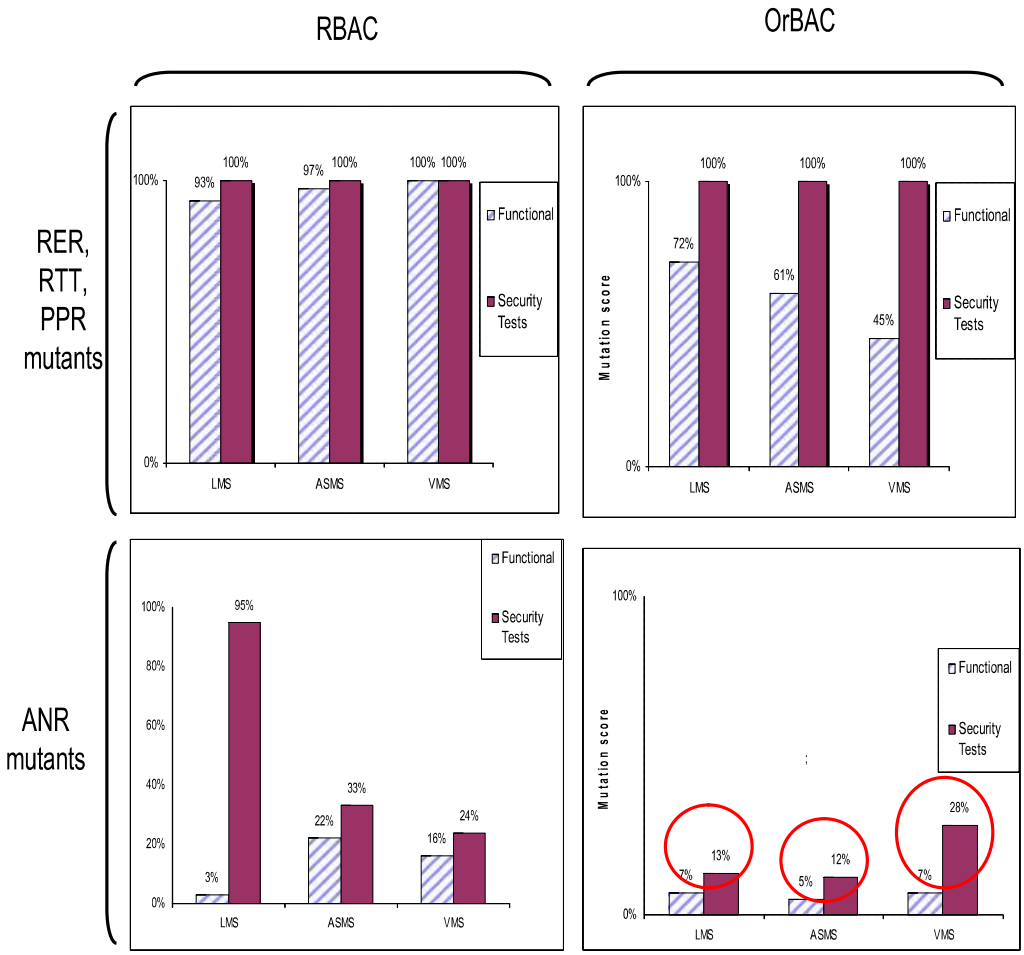


Figure 61 - Mutation scores per mutation operator and language

6.7 Discussion about language-independent V&V vs. language specific ones

There are some limitations and drawbacks when using a generic metamodel for expressing rule-based access control languages. The issues that we met are the following:

- Sharing V&V treatments in a language-independent model permits common certification to be performed
- Sharing these V&V does not allow dealing with complex and specific potential flaws.
- The result is that a trade off must be found between genericity and specificity in case of intensive use of MDE.

6.7.1 Limitations of the verification

The verification steps we propose allow detecting simple inconsistencies but cannot replace language-specific verifications, such as the ones performed for example for access control models like Or-BAC with a tool called MotOrBAC [141]. Access control languages do not usually provide such specific verification environment which makes our approach useful. Thus, the verification functions we propose have the advantage of being embedded in the metamodel. For instance, compared to a direct XACML code production, generic verifications offer a systematic way of detecting inconsistencies. In Table II, we highlight the relevance of the proposed verification functions for each of the targeted access control policy language. We notice that the no-conflicts verification is only useful for OrBAC policies, even if the semantics of this verification function is attached to the metamodel and is thus generic. If a new access control language including the notions of prohibition was modeled using the security metamodel, it would embed this useful verification function. Concerning the no-redundancies verification, while it has a meaning for each language even if the relevance is weak when the language includes no hierarchy (it only detects that there is no identical rules in the security policy). In conclusion, we have the following paradoxes:

Some verification checks expressed on the meta-model have no meaning for a specific language

Most advanced checks (e.g. logic conflicts detection, behavioral properties) cannot be expressed on our meta-model due to the fact the specific concepts of access control languages are not captured by the meta-model,

The mutation operators we propose may be used to create faults for any language which is based on rules, even if it has no link with security and access control. In fact, if we analyze the metamodel we propose, we notice that there is no notion specific to security, but only concepts to capture the notion of rule and its parameters.

Table 33 - specific mutants vs. Generic mutants

System	generic mutants	specific mutants
LMS	1044	371
VMS	1572	1426
ASMS	3088	2056

The solution to that problem might be metamodels composition/specialization. It would allow attaching partial semantics elements to each metamodel. For example, if we manipulate a real-time language with access control operations, we would like to combine in the same metamodel real-time notions and access control notions, as well as their semantics. Each partial metamodel could focus on each aspect and the final metamodel would be obtained by composition of these elements.

Table 34 - Relevance of generic verification for specific access control languages

	Policy_is_conform	No_conflicts	No_redundancies
RBAC	y	n	y
OrBAC	y	y	y
DAC	y	n	n
MAC	y	n	n

6.7.2 Language-independent vs. language specific qualification

Table 33 compares the number of mutants we obtain with OrBAC policies using a specific approach instead of the generic approach (based on the language-independent metamodel presented in [6]).

Table 35 presents the mutation scores, with functional and security test cases, obtained with the generic and the specific approach. The specific approach has been presented in [10, 16] and it benefits the MotOrBAC tool used to generate mutants.
The delta reveals that, for all cases, the variation of mutation scores is lower than 10%. This delta is due to the generation of more mutants with the generic approach, and reflects the proportion of equivalent mutants when using the generic approach. We could discuss if the lack of quality of some mutants generated using the MDE process is counter-balanced by the interest of having a certification process that is language-independent. Moreover, this distance that separates generic and specific can be reduced by the addition of a mutant filtering function at the language level that will remove irrelevant mutants.

Table 35 - OrBAC mutation results vs. Generic mutation results

Mutants	**Basic Mutants (func. tests)**			**ANR mutants (sec. tests)**		
System	**LMS**	**VMS**	**ASMS**	**LMS**	**VMS**	**ASMS**
Generic mutants	72%	61%	45%	13%	12%	28%
Specific mutants	78%	69%	55%	17%	19%	33%
Delta	**-6%**	**-8%**	**-10%**	**-4%**	**-7%**	**-4%**

6.8 Conclusion

We have presented a new approach that uses a generic metamodel for the definition of rule-based access control formalisms and the specification of access control models. The access control policy is exported to an XACML file that will be included in the PDP. Then, the PEP is implemented manually using AOP. We also define a fault model at a generic level and use it to qualify the security tests needed to validate the implementation of the access control policy. The feasibility of the approach and the benefits of the common validation process are both illustrated by applying the approach to 3 case studies.
This approach can be extended toward two directions; one is related to the automatic generation of security tests, the other to the improvement of the flexibility of the security design and deployment.
To improve security test automation, security test cases could be automatically derived from security rules: the problem which must be solved is the construction of the oracle. This problem has been partly studied in [7] using AOP techniques to weave an oracle function to an existing functional test case. The full generation of a security test case (data + oracle) is still an open issue.
A second possible direction involves integrating the security modeling with the application modeling process. A new methodology for implementing security-driven applications can be proposed which exploits the access control policy model at runtime. From a policy defined by a security expert, an architectural model could be directly generated, reflecting the access control policy. The idea is to leverage the advances in the models@runtime domain to keep this model synchronized with the running system. When the policy is updated, the architectural model is updated, which in turn reconfigures the running system. This approach would guarantee an increased flexibility in the code and the correctness of the implemented access control policy by construction. The next chapter will present this approach in details.

Chapter 7
Security-Driven Model-Based Dynamic Adaptation

There are two main limitations for the MDE approach proposed in the previous chapter. The first one concerns the integration of the access control mechanisms. This is still done manually to adapt to the application architecture and hence this process is error prone. The other main limitation is about the fact that this approach leaves the door open to hidden security mechanisms since it relies of the classical PEP-PDP architecture and the development of the security mechanisms and the business logic are separated. This affects the flexibility of the system and makes it difficult for the access control policy to evolve.
This chapter proposes a new approach that provides a new methodology to tackle these two issues. From a policy defined by a security expert, we generate an architectural model, reflecting the access control policy. We leverage the advances in the *models@runtime* domain to keep this model synchronized with the running system. When the policy is updated, the architectural model is updated, which in turn reconfigures the running system.

7.1 Introduction

The current implementation techniques consist in using a standard architecture (PDP+PEPs).
The main limitation of the current implementation techniques is that they do not allow flexible access control mechanisms. In fact, the access control policy cannot be modified without previously modifying the code to support the new access control rule. This is an expected effect of the separation between the security code (access control mechanism) and the functional code. In fact, as demonstrated in chapter 4, the functional code may contain some hard-coded access control mechanisms implementing some access control rules. Modifying these specific rules requires locating these hidden mechanisms and removing them. This chapter addresses this issue by providing flexible access control mechanisms.
This chapter proposes to leverage Model-Driven Engineering (MDE) techniques to provide a very flexible approach for managing access control. On one side, access control policies are defined by security experts, using a Domain-Specific Modeling Language (DSML), which describes the concepts of access control, as well as their relationships. On the other side, the application is designed using another DSML for describing the architecture of a system in terms of components and bindings. This component-based software architecture only contains the business components of the application, which encapsulate the functionalities of the system, without any security concern. Then, we define mappings between both DSMLs describing how security concepts are mapped to architectural concepts. We use these mappings to fully generate an architecture that enforces the security rules. When the security policy is updated, the architecture is also updated. Finally, we leverage the notion of *models@runtime* in order to keep the architectural model (itself synchronized with the access control model) synchronized with the running system. This way, we can dynamically update the running system in order to reflect changes in the security policy. Only users who have the right to access a resource can actually access this resource.

The remainder of the chapter is organized as follows. Next section presents an overview of our approach. Then, we present the security and the architecture metamodels, and how they are composed. Afterwards, we describe how the architecture model is impacted when the security policy is updated, and how these changes are automatically reflected at runtime.

7.2 Overview

In commercial and government environments, any change to the security policies normally requires impact assessments, risk analysis and such changes should go through the RFC (Request for Change) process. However, in case of urgency (crisis events, intrusion detection, server crashes, interoperability with external systems to deal with a critical situation), the adaptation of a security policy at runtime is a necessity. This adaptation may or may not have been already predicted or planned.

The proposed approach and the combination of composition and dynamic adaptation techniques allow the security policy to be adapted based on predefined adaptation plan or in an unplanned way. The security adaptation mechanisms we propose deal with the challenging issue of how to provide running systems supporting planned and unplanned security policy adaptations. The inputs of the process are two independent models: the business architecture model of the system and the security model. These two models are expressed in different domain-specific modeling languages: the core business architecture with an architecture modeling language and the security policy with an access-control language. By dynamically composing the security model with the architecture model, the approach allows adapting the application security policy according to pre-defined adaptation rules but also to cope with any unplanned changed to the security model.

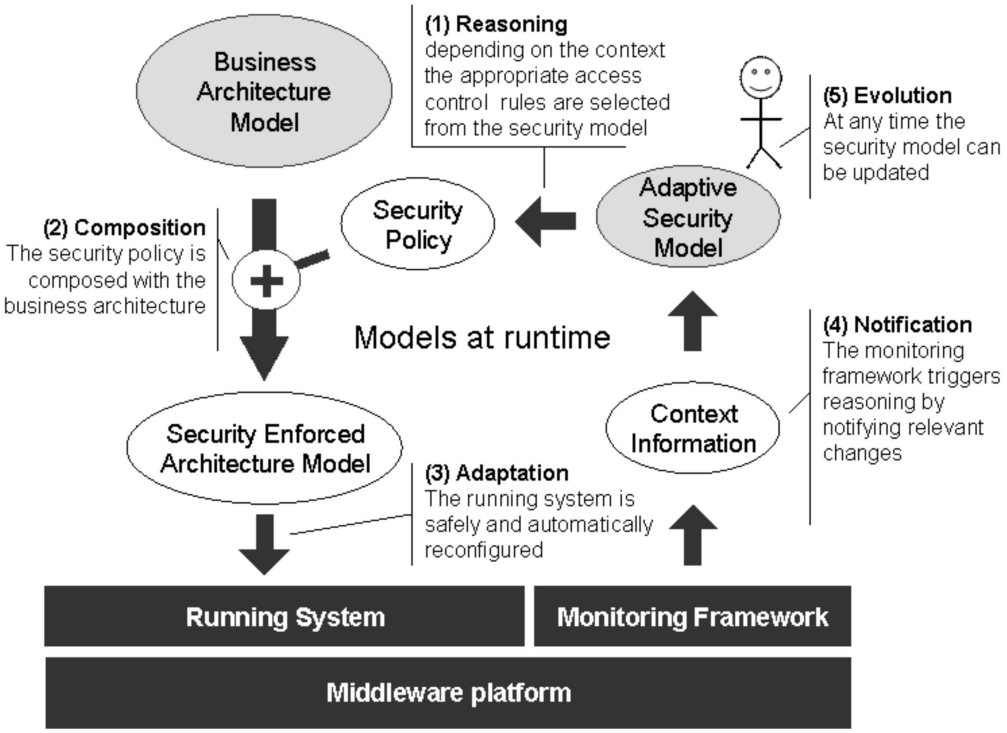


Figure 62 - Overview of the proposed approach

The adaptive security model contains a set of access control rules and a description of the context in which these rules apply. At runtime, depending on the context information coming from the system the appropriate set of rules has to be chosen. This is the reasoning shown as (1) on Figure 62. The reasoning phase processes the security model based on the context information coming from the system to produce the security policy to be enforced. Basically,

when some events are triggered, security rules can be activated or deactivated. Once the appropriate security policy has been defined, it has to be composed into the architecture model of the application, see (2) on Figure 62. The models to compose here are of different nature: an architecture model on one side and a security policy on the other side. The composition is done in two steps:

1. The security policy is transformed into an architecture model realizing it,
2. This architecture model is composed with the business architecture model

The implementation of this complex composition operator is discussed in the next section. The output of the composition is a plain architecture model which enforces the security policy in the application. Step (3) and (4) on Figure 62 corresponds to the synchronization of the architecture model with the running system and the monitoring of the system environment. For both of these tasks, the approach proposed in this chapter reuses existing runtime adaptation techniques. Basically the idea is to leverage monitoring and runtime reconfiguration mechanisms offered by middleware platforms in order to extract context information and update the running system to match the desired architecture. Finally, point (5) corresponds to an evolution of the security model. The proposed approach allows changing the security model at any point in time. Here, it does not only consists in activating/deactivating existing rules, but also consists in adding, removing or updating rules or roles. This chapter focuses on how the architecture (and the running system) causally evolves when the security policy is updated.

7.3 Composing Security and Architecture Metamodels

We chose to have a new metamodel for access control policies instead of the generic metamodel previously introduced in Chapter 6 for two reasons. Firstly, we needed a metamodel which expresses the security concepts (role, context etc.) in a way that facilitates the model composition and the generic metamodel does not include directly these concepts. Secondly, one of the main advantages of the previous generic metamodel is that it allows mutation in a generic way. We did not need mutation for this new approach. That is why we chose to use another one that is more expressive and easier to use.

In this section, we describe how we compose the architecture metamodel into the security metamodel. We first introduce a new metamodel for describing access control policies. Then, we introduce our generic metamodel for describing component-based architecture. Finally, we detail how we map security concepts to architectural concepts.

7.3.1 A Metamodel for Describing Access Control Policies

The metamodel we use to describe access control policies is illustrated in Figure 63. It defines the concepts and their relationships needed to design access control models. The root concept is the *Policy*. A policy contains *Resources*, *Roles*, *Users* and *Rules*. A resource (*e.g.*, a book) defines some actions (*e.g.*, borrow). Each user of the system has one role. Rules allow specifying the action the users (via their role) can perform on the resources of the system. We distinguish two types of rules:

- **Permission**: specifies the actions that the user of a given role can perform. By default, users can access the actions associated with their roles.
- **Delegation**: specifies the actions that a user (delegator) delegates to another user (delegatee). By default, no delegation is active.

Each rule can be associated with a context. A context specifies when a rule should be active, depending of the environment. For example, users can borrow books from Monday to Friday if they have not exceeded their quota. A context is a Boolean expression involving some context variables. Rules with no associated context are active by default, but could be deactivated.

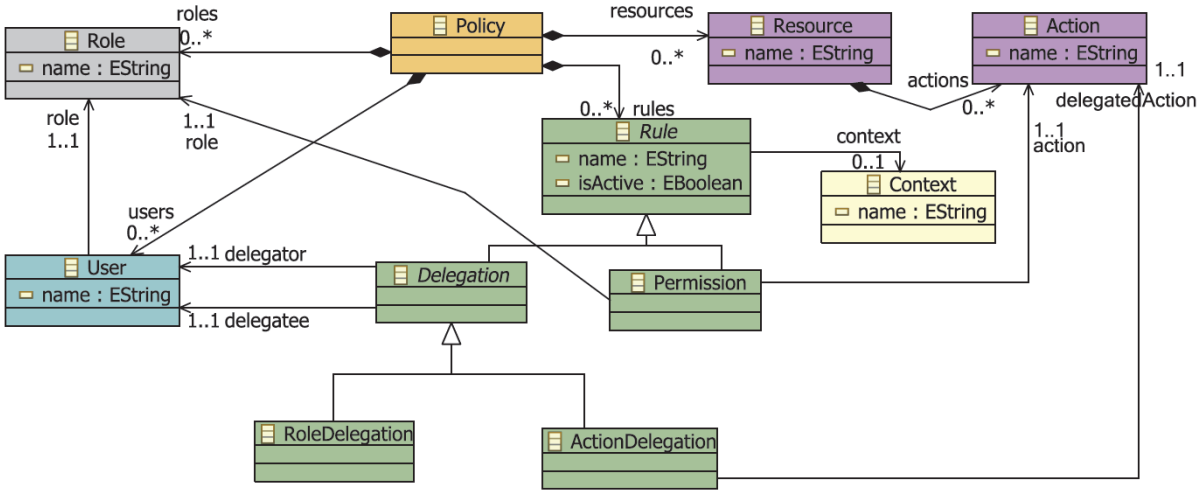


Figure 63 - Access Control Metamodel

7.3.2 A Core Metamodel for (Runtime) Architectures

Our generic metamodel is illustrated in Figure 64. A component type contains some ports. Each port has a UML-like cardinality (upper and lower bounds) indicating if the port is optional (lowerBound = 0) or mandatory (lowerBound > 0). It also indicate if the port only allows single bindings (upperBound = 1) or multiple bindings (upperBound > 1). A port also declares a role (client or server) and is associated to a service. A service encapsulates some operations, defined by a name, a return type and some parameters. A service has a similar structure as Java interface.

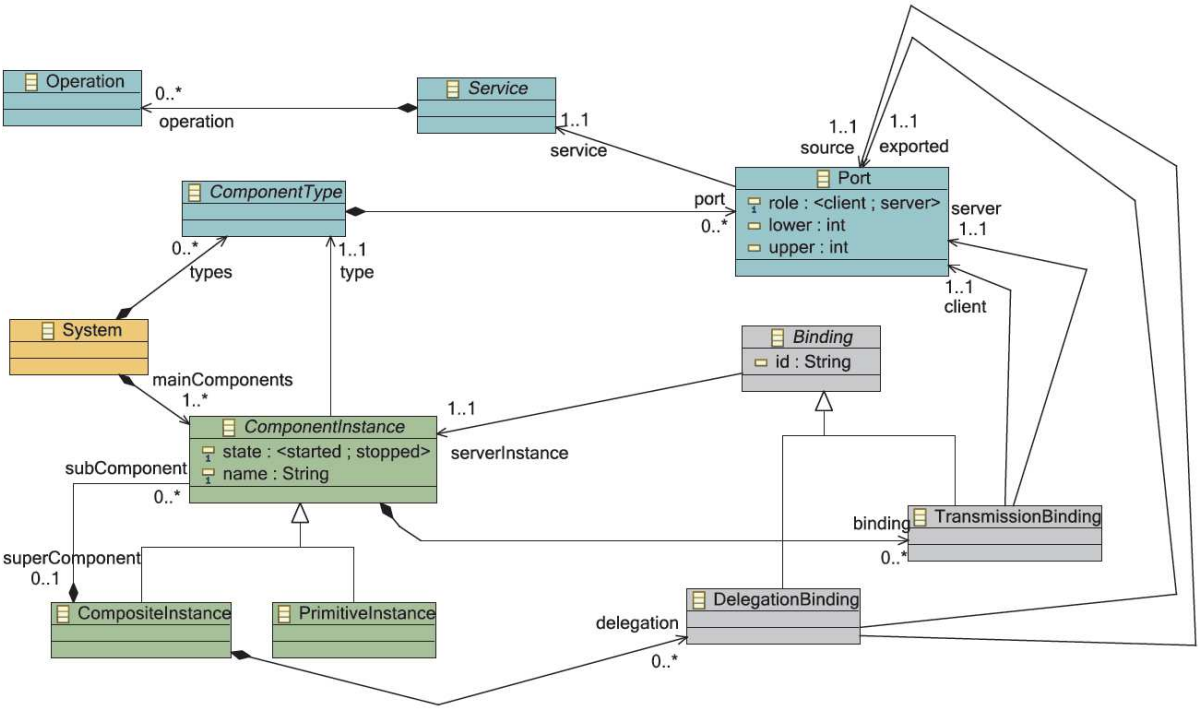


Figure 64 - A Core Metamodel for Describing Runtime Architectures

A component instance has a type and a state (ON/OFF), specifying whether the component is started or stopped. It can be bound to other instances by a transmission binding, linking a provided service (server port) to a required service (client port). A composite instance can

additionally declare sub-instances and delegation bindings. A delegation binding specifies that a service from a sub-component is exported by the composite instance.

7.3.3 Mapping Security Concepts to Architectural Concepts

In this sub-section we propose to map the security concepts to architectural concepts. This mapping is performed by the seamless weaving engine provided by Kermeta that allows designers to extend a metamodel with additional elements: attributes, references, contracts (invariants and pre/post conditions), operations (or implement an already existing abstract operation). The rationale of this mapping is to automatically reflect the access control policy at the architectural level. This mapping is realized as follows, (as shown in Figure 65):

- Each resource is mapped to a component. This proxy component provides and requires all the services offered by the resource. Additionally, each resource is mapped to a set of business components, from the base architecture, realizing the resource. The mapping with business components is provided by the designer, via a graphical editor.
- Each action is mapped to an operation. An OCL constraint ensures that every action (belonging to a single resource) is mapped to an operation belonging to the component type of a component realizing the resource. This mapping is also provided by the designer.
- Each role is mapped to a component. This proxy component provides and requires all the services that a user of this role can potentially access.
- Each user is mapped to a component. By default, this component is connected to the corresponding role component.
- Each permission is mapped to a pair of ports and a binding:
    - Each permission granted to a role is mapped to a pair of ports: a required port associated with the role component, a provided port associated with the resource, and a binding linking these ports.
    - Each permission granted to a user is also mapped to a pair of ports: a required and a provided port associated with the user component. The binding links the required port to the corresponding provided port of the component corresponding to the user's role.
- Each delegation is mapped to a pair of ports and a binding. A required and a provided port are associated with the user (delegatee) component. The binding links the required port to the corresponding port provided by another (delegator) component.

The initial architecture is automatically created by using the mappings provided by the designer (resources and actions), and by visiting the security model to set the woven references. Figure 65 illustrates a simple architecture generated from a security policy. The architecture has 4 layers: business, resource, role and user. The bindings between components from different layers are related to the permissions, and the bindings between user components are related to the delegations currently active in the system. During the generation of the architecture, bindings are not created if the rule is not active.

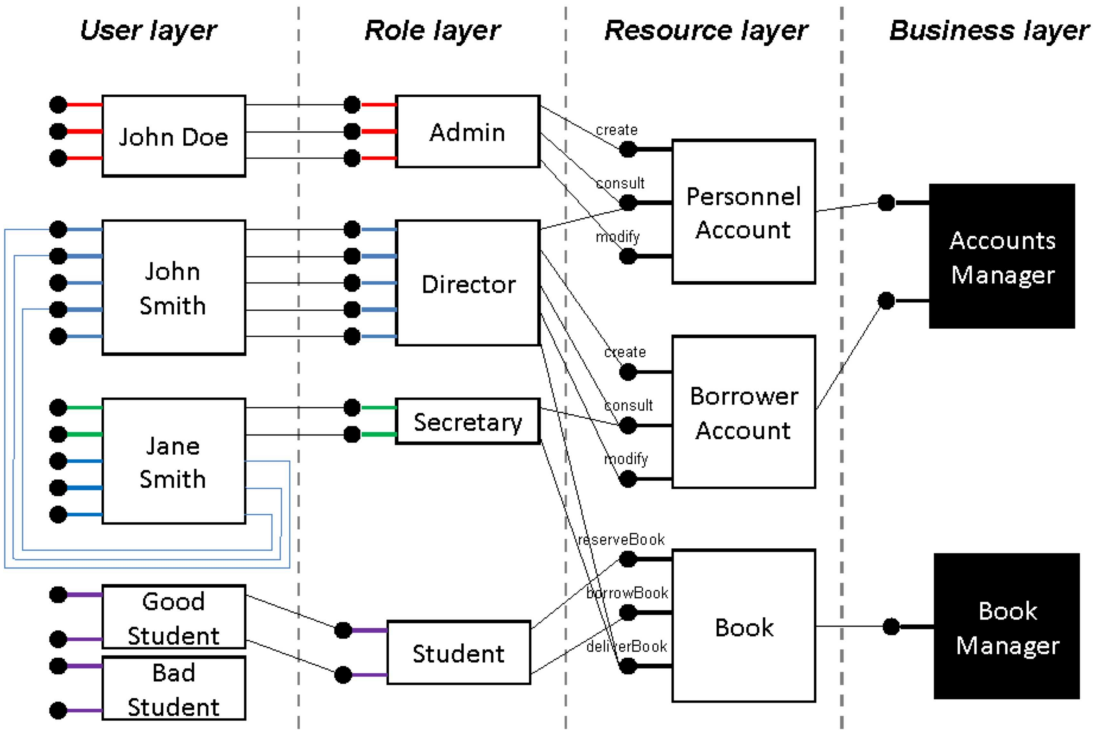


Figure 65 - 3-Layered architecture reflecting access control policies

In this section, we showed how the security (access control) concern is composed with a base architecture, to finally end up with 4-layered software architecture. This mapping leverages the meta-level aspect weaving facilities provided by the Kermeta language, as well as its model transformation capabilities.

7.4 Security-Driven Dynamic Adaptation

In this section, we first present how we synchronize architectural models with a running system. Then, we explain how the architecture evolves when access control rules are activated or deactivated. Finally, we show how we handle deeper evolutions of the access control policy *i.e.*, when rules or roles are removed, created or updated.

7.4.1 Synchronizing the Architecture Model with the Running System

Modern adaptive execution platforms like OSGi [142] propose low-level APIs to reconfigure a system at runtime. It is possible to dynamically reconfigure applications running on these platforms by executing platform-specific reconfiguration scripts specifying which components have to be stopped, which components and/or bindings should be added and/or removed. These scripts have to be carefully written in order to avoid life-cycle exceptions (*e.g.* when a component is removed while still active), dangling bindings (*e.g.*, when a component is removed while it is still connected to other (client) components).
To prevent errors in writing such error-prone scripts, we leverage MDE techniques to generate safe reconfiguration scripts, as illustrated in Figure 66.

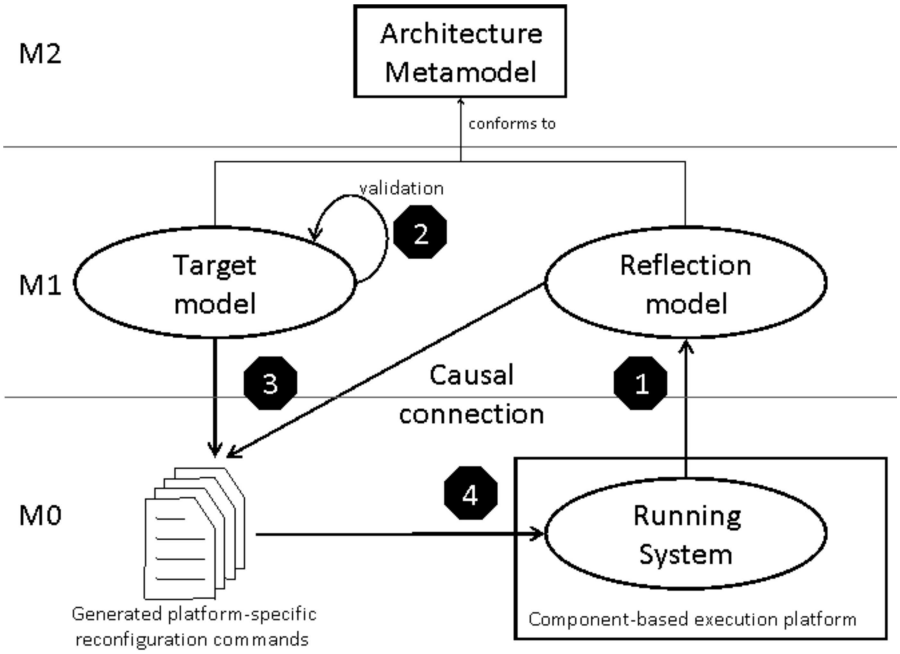
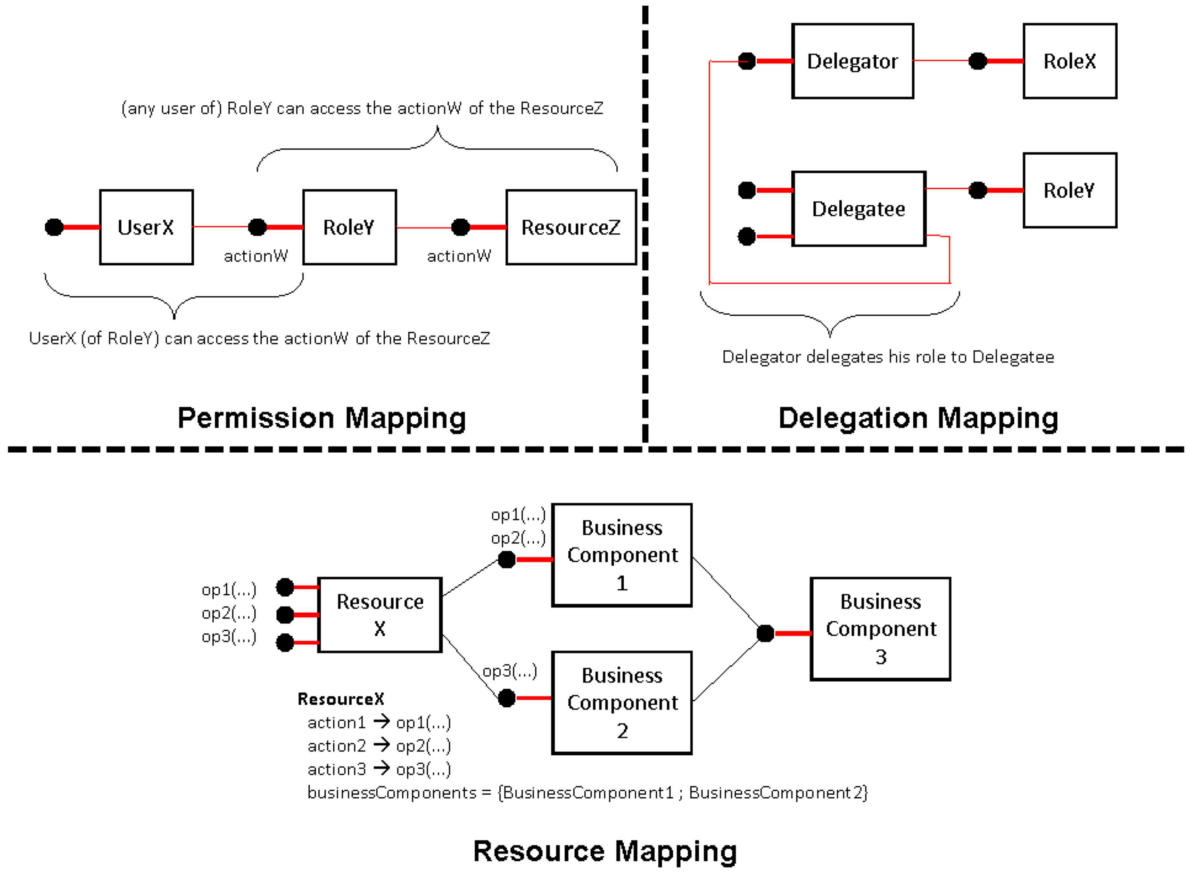


Figure 66 - Leveraging MDE to generate safe reconfiguration scripts

The key idea is to keep an architectural model synchronized with the running system. This model, which conforms to the architecture metamodel, is updated (Figure 66, 1) when significant changes appears in the running system (addition/removal of components/bindings). It is important to note that the reflection model can only be modified according to runtime events. Still, it is possible to work on a copy of this reflection model and modify it: model transformation, aspect model weaving, manipulation by hand in a graphical editor, etc. In other words, the reflection model is really a mirror of the reality (the running system) and not a mean to manipulate the reality.

When a target architectural model is defined (*e.g.* after updating the access control policy), it is first validated (Figure 66, 2) using classic design-time validation techniques, such as invariant checking [143] or simulation. This new model, if valid, represents the target configuration the running system should reach. We automatically generate the reconfiguration script, which allows switching the system from its current configuration to the new target. We first perform a model comparison between the source configuration (the reflection model) and the target configuration (Figure 66, 3), which produce an ordered set of reconfiguration commands. This safe sequence of commands is then submitted (Figure 66, 4) to the running system in order to actually reconfigure it. Finally, the reflection model is automatically updated and becomes equivalent to the target model (Figure 66, 1).



Resource Mapping

Figure 67 - Permissions, Delegations and Resources to Architectural Concepts

In the next two sub-sections we leverage this causal connection and the mappings in order to automatically update the architecture when the security policy is modified.

7.4.2 Activation/deactivation of security rules

Default permissions can temporarily be deactivated for several reasons: maintenance of a resource that requires the resource to be "offline", repression of bad or abusive behavior, etc. In some cases, default permissions should be deactivated for a particular role, while in some other cases they should be deactivated for a particular user. Delegations are temporarily activated by a user when he goes on holidays, when he is not available, etc, and deactivated when the user is back to work.

Deactivating a permission, either for a role of for a user, is straightforward: it simply consists in removing the binding associated with the rule. This way, the chain-of-responsibility cannot process to the business components. Re-activating a permission is exactly the opposite: we have to create a binding an insert it at the right place. Delegations are handled exactly in the same way.

Figure 68 shows a code snippet written in Kermeta. The aspect keyword means that we *re-open* the Permission meta-class, defined in the security metamodel. We add 3 new references, corresponding to the mappings we have previously defined, and an operation. The `activate` operation describes the impact on the architecture when a permission is activated. We first create a binding (Line 10) and set the four references needed to properly introduce the binding: the client and the server ports (Lines 11 and 12), and the client and server component instances (Lines 13 and 14). The activation and deactivation of permissions for particular users follow the same principle.

```
aspect class Permission {
// mappings to architectural concepts
reference server : Port [1..1]
reference client : Port [1..1]
reference binding : TransmissionBinding [1..1]

operation activate () i s do
var b : TransmissionBinding init TransmissionBinding . new
b. client := self . client
b. server := self . server
b. serverInstance := self .action.container.asType ( Resource ). resourceComponent
self . role . roleComponent . binding . add (b)
binding := b
end
```

Figure 68 - The Permission meta-class aspectized with Kermeta

In Figure 65, we can see that John Smith has delegated his role to Jane Smith. At the architectural level, this means the component associated with Jane Smith has now 3 additional ports that are delegated to the component associated to John Smith. Jane Smith still has the permissions associated with her role (Secretary), but she now also has the permissions associated to the Director role, via the *John Smith* component. In the same figure, we can see that the "bad student" user has lost all his permissions. However, the permissions have not been deactivated for all the students. At the architectural level, all the bindings formerly connected to the component associated with the "bad student" have been removed. In other words, the "bad student" is now totally isolated from the system. The modifications of the architecture, implied by the (de)activation of rules, are automatically reflected to the running system using the causal connection.

At runtime, the removal/addition of binding is very fast. It involves calling a setter method on the client component to link it (or unlink it) to (or from) a server. In the case where the permissions of all the users of a given role should be removed, we simply disconnect the role component from the resource components. This way, we do not have to remove all the (numerous) bindings between the user components and the role component. In the case where a given user loses his permissions, we remove the binding between its component and its associated role component. This way, other users are not impacted by this modification.

7.4.3 Evolution of the Access Control Policy

In the previous sub-section, we explained how the activation and deactivation of rules update the architecture, which in turn dynamically reconfigure the system. However, our model-driven approach also makes it possible to deeply modify the access control policy, by adding, removing, updating, rules, resources, roles or users, which is very difficult and often impossible using traditional security architectures.

For example, if the policy is too permissive, the security manager may want to remove some permissions granted to a role, or create a new role with less permissions. On the other hand, a policy could be too restrictive to be actually usable in practice. In the former security policy, only the director could create and update borrower accounts. To help the director in this task, it has been decided that the administrator of the system could now also perform these tasks. One possible solution would be to ask the director (John Smith) to delegate his role (or at least the actions related to borrower accounts) to John Doe, the administrator. Since delegations are temporary by nature, this solution is not well suited. The best solution is to modify the security policy by creating 3 new permissions to allow the administrator to create, consult and modify borrower accounts.

Figure 69 focuses on the architectural elements related to the administrator role before and after the change in the policy. After the new architecture has been re-generated, we can see that the administrator can now access to the borrower accounts.

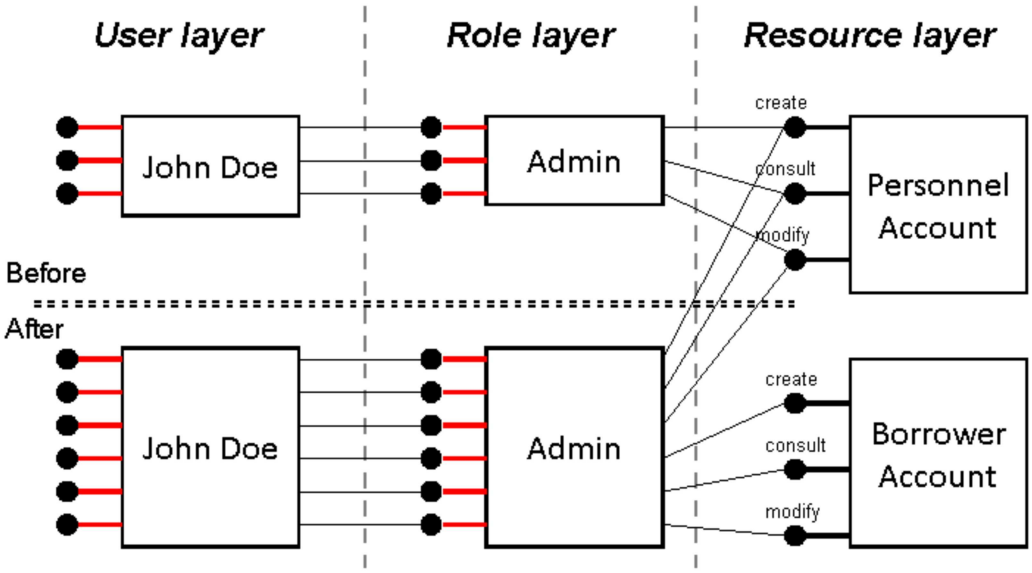


Figure 69 - Architecture after 3 new permissions have been added

When the new architecture is submitted to our causal connection, this produces the following reconfiguration script:

```
stop component John Doe
stop component Admin
unbind component John Doe from createPersonnelAccount interface
unbind component John Doe from consultPersonnelAccount interface
unbind component John Doe from modifyPersonnelAccount interface
//same actions for the Admin component
remove component John Doe
remove component Admin
generate John Doe
generate Admin
add component John Doe
add component Admin
bind component John Doe to Admin via createPersonnelAccount interface
bind component John Doe to Admin via consultPersonnelAccount interface
bind component John Doe to Admin via modifyPersonnelAccount interface
bind component John Doe to Admin via createBorrowerAccount interface
bind component John Doe to Admin via consultBorrowerAccount interface
bind component John Doe to Admin via modifyBorrowerAccount interface
//same actions between Admin and PersonnelAccount
//same actions between Admin and BorrowerAccount
start Admin
start John Doe
```

7.4.4 Discussion

Unlike the classic architecture (based on PDP and PEP), our architecture introduces several additional components to properly manage access control. Indeed, we introduce a component for each resource and each role defined in the access control policy. These components are implemented as OSGi bundles, and we rely on the OSGi API to manage these components. The code of these components is straightforward and efficient: it simply delegates all their

provided services to another component. For example, the following script illustrates the pseudo-code of the *ResourceX* in Figure 67.

```
op1(…){ myBusinessComponent1.op1(…)}
op2(…){ myBusinessComponent1.op2(…)}
op2(…){ myBusinessComponent2.op3(…)}
```

This code contains no logic (no reflection, no *if-then-else*, etc to dispatch the calls), but a very simple indirection. User components delegates to role components (permissions) or to other user components (delegation). Role components delegate to resource components, which finally delegate to the business components, which contains all the business logic. The code of these proxy components is automatically generated using a code template. Since OSGi offers powerful mechanisms to manage the *classpath* at runtime, it is possible to generate, compile and package this code at runtime. In the case where a user has the permission to use a resource, it thus has to transit through 3 simple indirections: user → role → resources → business code. This is comparable to the classic architecture: user → eval(PDP) → business code, where we have 2 indirections, plus the evaluation of the result of the PDP.
The activation/deactivation of permission simply consists in setting a Java reference using a setter method, which is automatically invoked during the reconfiguration step. Disabling a permission thus "physically" breaks the chain between the user and the real resources.
Our approach also reifies users as components. Unlike resources and roles, we do not encapsulate these components as OSGi bundles. Instead, we simply manipulate these components as pure objects, similarly to session objects we can find in application that should handle multiple users. These light-weight components are instantiated when needed *e.g.*, when users log in, with no overhead comparing to classic session objects.

7.5 Conclusions and Future Work

This chapter proposed an approach that leverages Domain-Specific Languages, Model-Driven Engineering and models@runtime. Access control policies are expressed by a security expert using a dedicated DSML. This DSML is mapped to another DSML describing (runtime) software architectures. When the access control policy is modified, the architecture is causally impacted. Architectural models are kept synchronized with a system running on a component-based platform. This way, the running system always reflects the access control policy.

Conclusions and perspectives

This thesis focused on proposing new approaches for access control testing and bypass-testing. In a design-for-testability logic, we proposed new methodologies for developing and integrating security mechanisms (aspect-oriented programming and modeling).
While it firstly focused on testing, my thesis promotes security as a first-class concern, from early design stages to final security testing, This chapter summarizes the contributions in that direction and discusses the perspectives and the open research questions.

8.1 Summary

This thesis tackled in its first part the issue of security testing, focusing on the internal accesses to the system, then on testing and shielding the interface of a system against bypass attacks. The second part of the thesis takes benefit from the results of the first part to go beyond testing and contribute to a more rational model-driven security process.

Chapter 3 evaluated whether is it possible to reuse functional tests to validate access control mechanisms. These functional test cases trigger the access control mechanisms and can be reused to test security. In addition, a second research question is about the choice of a testing strategy, in terms of cost. Between the two testing strategies, what is the best one; the independent strategy that relies only on generating new security test cases or the incremental one which involves reusing, adapting and completing the functional test cases?
To answer these research questions, one should use an evaluation method. Mutation analysis allows performing this task by assessing the quality of test cases by evaluating their ability to detect errors. An empirical study was conducted to compare between functional tests and security specific tests. These security tests were generated according to test criteria to cover access control rules. Based on the empirical results, several important trends were observed which led to the following first conclusions: functional test cases are not sufficient to test security mechanisms and security tests is a specific testing task, and the incremental strategy seems better that the independent one, especially if the transformation of functional test cases into “security” test cases can be automated.
Chapter 4 takes into account these findings and proposes new methodologies for access control testing. Three approaches are proposed. The first two are targeting access control test case generation. An approach based on mutation analysis allows functional test cases to be selected and transformed into security test cases. This allows functional tests to be easily reused for testing security, and thus reinforce the interest of the incremental test strategy. To complete these tests, it is important to be able to generate specific access control tests. To this end, a second approach leverages pair-wise sampling to generate test targets that are then instantiated to create concrete security tests. These two approaches allow testing existing and legacy access control mechanisms. However, in order for the access control policy to evolve, hidden security mechanisms harming the flexibility of the system should be detected and removed or adapted. The third approach tackles this issue and proposes a testing process for locating these hidden mechanisms.
Chapter 4 proposed a combined solution that addresses the issue of testing access control mechanisms. Compared to the state of the art, this solution considers testing the whole access control mechanisms (PDP+PEP) while other approaches (like Xie et al. [41, 42]) consider testing the PDP only. We believe that our approach is a big step forward.
Testing the internal accesses to a system must be completed by the insurance that security mechanisms also protect the external part of web applications. The issue of the bypass testing

is thus tackled in the last chapter dedicated to security testing. The idea is to rely on the analysis of the applications web page to collect all the HTML and JavaScript constraints. These constraints are lifted from the client side and can be completed or improved both manually or automatically. Then, they are enforced inside the bypass-shield, which is a reverse-proxy that is located in the server side. This way the bypass attacks can be stopped. In addition to protecting the web applications, this process can be used to audit. The constraints that are automatically collected are used to generate test data violating these very constraints. In addition, they could be used with other fuzzing tools.
The bypass-shield solves the issue of bypass-attacks and provides an automatically build protection, that is more importantly adapted to each web applications. Others solutions in the industry exist. The most known are ModSecurity. However, it is a generic solution that needs to be adapted (manually) to the web applications in order to be able to enforce the specific constraints. Our tool is build automatically by parsing the web pages and automatically extracting constraints to be enforced on the server side within the bypass-shield.
This chapter ends the first part of this thesis that focused on testing security mechanisms in web applications.
The second part goes beyond testing. Understanding the testing issues encountered in the first part of the thesis leads to think about better development and integration methodologies. In fact, the testability of the system is tightly linked to the way it was designed and deployed. Moreover, mutation analysis has shown that it is an effective and powerful certification tool. However, there are several access control models. In order to be able to apply mutation in a generic way, Chapter 6 proposes an original metamodel for access control policies which allows defining mutation operators in a generic way. Any access control model that is instance of this access control metamodel “knows” how to mutate itself by embedding the operational semantics of mutation operators issued by the metamodel. This metamodel is included in a framework for specifying and deploying the access control policy in the XACML standard format. The integration of the policy with the business logic is still performed manually using aspect oriented programming to weave the PDP calls. The testing certification process relies of the mutation analysis. Mutants are generated automatically and they can be used to assess the quality of the security tests.
The main originality of this approach, when compared to existing ones (like SecureUML [69]) is two fold. Firstly, it considers the testing at an early stage, during the modeling of the access control policy and provides the possibility to build testing artifacts (the mutants) to be used for security test cases qualification. Secondly, this approach can be applied to several access control models, as demonstrated in Chapter 6. We were able to apply it to RBAC, OrBAC, MAC and DAC.
The final chapter addresses the issue of hidden mechanisms by using model composition and by leveraging the advances in the field of model@runtime. The objective is to be able to maintain the alignment of access control policies with the business logic. To reduce this problem, a new methodology is proposed for implementing security-driven applications. From a policy defined by a security expert it is possible to generate an architectural model which reflects the access control policy. To do this some mappings are defined. The architectural model is kept synchronized with the running system, behaving like an adaptive system. When the access control policy is modified the running system components are reconfigured to reflect the new access control policy.
This approach is an important step forward and improves the state of the art related to the integration of access control policy in applications. It makes it possible to have a fully flexible access control mechanisms. When the access control policy is change the system automatically takes into account these changes and is reconfigured. Another main advantage of this approach is that it is fully automated. The integration of the access control mechanisms

and the construction of the system components are fully automated using MDE tools and techniques.

8.2 New Research directions

This thesis work can be extended towards three main research avenues that are related to security testing in addition to new modeling approaches to build secure systems.

8.2.1 Mutation analysis for Usage Control

Firstly, it would be interesting to apply mutation analysis to usage control [25, 26]. The same concepts that we used for testing access control policies could be applied to usage control which is an important issue. Usage control takes place once access is granted to users in order to check how the service or the data is used. It is interesting to apply mutation analysis in order to assess security tests for usage control.

8.2.2 Extending the bypass-shield tool

In addition, concerning bypass testing tools. There are several ways to improve and complete the bypass-shield tool. Other attacks can be handled using the bypass-shield like for instance Cross Site Scripting (XSS) [144] or Session Hijacking [145]. The reverse proxy is ideally located to monitor all the client requests and the server responses and hence can be used to locate and track any trace of attacks and stop it. One other advantage of this solution is that it is completely transparent to the web applications. Furthermore, the tool can be completed in its audit functions. New and advanced attack patterns could be added to perform advanced security assessment of the web application. The new attacks may include any code-injection based attacks, remote-file inclusion (RFI), direct path access etc. The tool can be used for both launching these attacks and automatically classifying the results by checking the server responses.

8.2.3 Model-Driven approaches

Concerning the modeling part, the model based process can be applied to other contexts. The idea of using generic mutation can be applied to other language and contexts. When there is a set of languages that share some properties, it is possible to define a metamodel that encapsulates their common properties. Then, mutation analysis can be applied to this metamodel and consequently applied to any model instance of that metamodel.
In addition, the model-based approach presented in this thesis can be adapted to usage control as well. The framework could be used to specify and automatically deploy usage control policies.
The approach for model composition is very promising and opens the door to several research directions. A lot of new features are still to be included in the framework. It is possible to include new features like an intrusion detection system to enable the modification of the access control policy to react when the system is under attack or when an intrusion is detected. The solution proposed in Chapter 7, is interesting for building security-driven adaptative systems and for critical systems like banking and health care applications.

References

1. *Websense Security Labs report - State of Internet Security, Q1-Q2 2009*. Available from: http://securitylabs.websense.com/content/Blogs/3476.aspx.
2. *PandaLabs Annual Malware Report*. Available from: http://www.pandasecurity.com/homeusers/media/press-releases/viewnews?noticia=10010.
3. *Symantec blocks an average of 100 potential attacks per second in 2009*. Available from: http://www.symantec.com/about/news/release/article.jsp?prid=20100419_02.
4. *Businesses Lose More Than $1 Trillion in Intellectual Property*. Available from: http://www.mcafee.com/us/about/press/corporate/2009/20090129_063500_j.html.
5. Zitser Misha, Lippmann Richard, and Leek Tim, *Testing static analysis tools using exploitable buffer overflows from open source code.* SIGSOFT Softw. Eng. Notes, 2004. **29**(6): p. 97-106.
6. 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.
7. T. Mouelhi, Y. Le Traon, and B. Baudry, *Transforming and selecting functional test cases for security policy testing*, in *2nd International Conference on Software Testing, Verification, and Validation, ICST*. 2009.
8. A. Pretschner, T. Mouelhi, and Y. Le Traon, *Model-Based Tests for Access Control Policies*, in *1st International Conference on Software Testing, Verification, and Validation, ICST 2008*. 2008.
9. T. Mouelhi, Y. Le Traon, and B. Baudry, *Transforming and Selecting Functional Test Cases for Security Policy Testing*, in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*. 2009, IEEE Computer Society.
10. 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.
11. 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.
12. B. Morin, T. Mouelhi, F. Fleurey, O. Barais, Y. Le Traon, and J. M. Jezequel, *Security-Driven Model-Based Dynamic Adaptation*, in *25nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2010*. 2010.
13. T. Mouelhi, Y. Le Traon, and B. Baudry. *Utilisations de la mutation pour les tests de contrôle d'accès dans les applications.* in *SARSSI 2009 : 4ème conférence sur la sécurité des architectures réseaux et des systèmes d'information*. 2009.
14. 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.
15. T. Mouelhi, F. Fleurey, B.Baudry, and Y. Le Traon, *Mutating DAC And MAC Security Policies: A Generic Metamodel Based Approach.*, in *Modeling Security Workshop In Association with MODELS '08*. 2008.
16. 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.
17. Y. Le Traon, T. Mouelhi, F. Fleurey, and B. Baudry. *Language-Specific vs. Language-Independent Approaches: Embedding Semantics on a Metamodel for Testing and*

Verifying Access Control Policies. in *Software Testing Verification and Validation Workshop, IEEE International Conference on*
2010.
18. *CWE/SANS TOP 25 Most Dangerous Programming Errors*. Available from: http://www.sans.org/top25errors/.
19. *Common Vulnerabilities and Exposures http://cve.mitre.org/*.
20. *OWASP TOP10 http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project*.
21. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, *HTTP Authentication: Basic and Digest Access Authentication*. 1999: RFC Editor.
22. M. Burrows, M. Abadi and R. Needham, *A logic of authentication.* ACM Trans. Comput. Syst., 1990. **8**(1): p. 18-36.
23. K. Fu, E. Sit, K. Smith, and N. Feamster. *Dos and Don'ts of Client Authentication on the Web*. in *Proceedings of the 10th USENIX Security Symposium*. 2001.
24. R. S. Sandhu and P. Samarati, *Access control: principle and practice.* Communications Magazine, IEEE, 1994. **32**(9): p. 40-48.
25. A. Pretschner, M. Hilty, and D. Basin, *Distributed usage control.* Commun. ACM, 2006. **49**(9): p. 39-44.
26. J. Park and R. Sandhu, *Towards usage control models: beyond traditional access control*, in *Proceedings of the seventh ACM symposium on Access control models and technologies*. 2002, ACM: Monterey, California, USA.
27. 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.
28. K. J. Biba, *Integrity consideration for secure computer systems*, in *Tech. Rep. MTR-3153, The MITRE Corporation,*. 1975.
29. D. E. Bell and L. J. LaPadula, *Secure computer systems: Unified exposition and multics interpretation*, in *Tech. Rep. ESD-TR-73-306, The MITRE Corporation*. 1976.
30. B. Lampson. *Protection*. in *5th Princeton Symposium on Information Sciences and Systems,*. 1971.
31. C.S. Jordan., *A guide to understanding discretionary access control in trusted systems. Technical Report Library No.S-228, 576, National Computer Security Center (NCSC), Fort George G. Meade, Maryland,* . 1987.
32. A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin, *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
33. M. Ben Ghorbel-Talbi, F. Cuppens, N. Cuppens-Boulahia, and A. Bouhoula, *Managing Delegation in Access Control Models*, in *Proceedings of the 15th International Conference on Advanced Computing and Communications %@ 0-7695-3059-1*. 2007, IEEE Computer Society. p. 744-751.
34. F. Cuppens and A. Miège, *Administration Model for Or-BAC*, in *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*. 2003. p. 754-768.
35. J. Sushil, S. Pierangela, S. Maria Luisa, and V. S. Subrahmanian, *Flexible support for multiple access control policies.* ACM Trans. Database Syst. %@ 0362-5915, 2001. **26**(2): p. 214-260.
36. C. Bertolissi, M. Fernández, and S. Barker, *Dynamic Event-Based Access Control as Term Rewriting*, in *Data and Applications Security XXI*. 2007. p. 195-210.
37. S. Barker and J. Stuckey Peter, *Flexible access control policy specification with constraint logic programming.* ACM Trans. Inf. Syst. Secur., 2003. **6**(4): p. 501-546.

38. E. Bertino, S. Ajodia and P. Samarati, *A flexible authorization mechanism for relational data management systems.* ACM Trans. Inf. Syst., 1999. **17**(2): p. 101-140.
39. S. Barker and M. Fernández, *Term Rewriting for Access Control*, in *Data and Applications Security XX*. 2006. p. 179-193.
40. *Sun's XACML implementation.* http://sunxacml.sourceforge.net/.
41. V. 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.
42. 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.
43. A Anderson, *XACML profile for role based access control (RBAC)*, in *OASIS Access Control TC committee draft*. 2004.
44. D. Abi Haidar, N. Cuppens-Boulahia, F. Cuppens and H. Deba. *An extended RBAC profile of XACML*. in *Proceedings of the 3rd ACM workshop on Secure web services*. 2006.
45. L. Mike Ter and V. N. Venkatakrishnan, *Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers*, in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. 2009, IEEE Computer Society.
46. P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, *SWAP: Mitigating XSS attacks using a reverse proxy*, in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*. 2009, IEEE Computer Society.
47. B. Prithvi and V. N. Venkatakrishnan, *XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks*, in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2008, Springer-Verlag: Paris, France.
48. A. Barth, J. Collin, and C. Mitchell John, *Robust defenses for cross-site request forgery*, in *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, ACM: Alexandria, Virginia, USA.
49. W. Maes, T. Heyman, L. Desmet, and W. Joosen, *Browser protection against cross-site request forgery*, in *Proceedings of the first ACM workshop on Secure execution of untrusted code*. 2009, ACM: Chicago, Illinois, USA.
50. M. Ziqing, L. Ninghui, and I. Molloy, *Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection*, in *Financial Cryptography and Data Security: 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*. 2009, Springer-Verlag. p. 238-255.
51. M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti and C. Kruegel, *A solution for the automated detection of clickjacking attacks*, in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ACM: Beijing, China.
52. S. W. Boyd and A. D. Keromytis. *SQLrand : Preventing SQL Injection Attacks* in *ACNS*. 2004.
53. F Valeur, D Mutz, and G Vigna. *A Learning-Based Approach to the Detection of SQL Attacks*. in *Intrusion and Malware Detection and Vulnerability Assessment*. 2005.
54. A. Brad, S. Scott, and G. McGraw, *Software Penetration Testing.* IEEE Security and Privacy, 2005. **3**(1): p. 84-87.
55. R. Fewster and E. Mendes, *Measurement, Prediction and Risk Analysis for Web Applications*, in *Proceedings of the 7th International Symposium on Software Metrics*. 2001, IEEE Computer Society.

56. F. Braber, I. Hogganvik, M. S. Lund, K. St, len, and F. Vraalsen, *Model-based security analysis in seven steps --- a guided tour to the CORAS method.* BT Technology Journal, 2007. **25**(1): p. 101-117.
57. R. Peltier Thomas, *Information Security Risk Analysis*. 2000: CRC Press, Inc. 296.
58. Hope Paco, McGraw Gary, and I. Ant Annie, *Misuse and Abuse Cases: Getting Past the Positive.* IEEE Security and Privacy, 2004. **2**(3): p. 90-92.
59. J. Pauli Joshua and X. Dianxiang, *Misuse Case-Based Design and Analysis of Secure Software Architecture*, in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*. 2005, IEEE Computer Society.
60. R. Croo, C. Ince Darrel, L. Luncheng, and B. Nuseibeh, *Security Requirements Engineering: When Anti-Requirements Hit the Fan*, in *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*. 2002, IEEE Computer Society.
61. L. Lin, Yu Eric, and J. Mylopoulos, *Security and Privacy Requirements Analysis within a Social Setting*, in *Proceedings of the 11th IEEE International Conference on Requirements Engineering*. 2003, IEEE Computer Society.
62. B. Haley Charles, D. Moffett Jonathan, R. Laney, and B. Nuseibeh, *A framework for security requirements engineering*, in *Proceedings of the 2006 international workshop on Software engineering for secure systems*. 2006, ACM: Shanghai, China.
63. A. van Lamsweerde, *Elaborating Security Requirements by Construction of Intentional Anti-Models*, in *Proceedings of the 26th International Conference on Software Engineering*. 2004, IEEE Computer Society.
64. D. Thuong, S. Demurjian, T. C. Ting, and A. Ketterl, *MAC and UML for secure software design*, in *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*. 2004, ACM: Washington DC, USA.
65. R. Indrakshi, N. Li, R.France Robert, and Kim Dae-Kyoo, *Using uml to visualize role-based access control constraints*, in *Proceedings of the ninth ACM symposium on Access control models and technologies*. 2004, ACM: Yorktown Heights, New York, USA.
66. Dae-Kyoo Kim, R. Indrakshi, F. Robert, and N. Li, *Modeling Role-Based Access Control Using Parameterized UML Models*, in *Fundamental Approaches to Software Engineering*. 2004. p. 180-193.
67. J. Juerjens, *Secure Systems Development with UML*. 2003: SpringerVerlag.
68. J. Jürjens, *UMLsec: Extending UML for Secure Systems Development*, in *«UML» 2002 — The Unified Modeling Language*. 2002. p. 1-9.
69. T. Lodderstedt, D. Basin, and J. Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002.
70. R. Indrakshi, R. France, N. Li, and G. Georg, *An aspect-based approach to modeling access control concerns.* Information and Software Technology, 2004. **46**(9): p. 575-587.
71. A. Bertolino, *Software Testing Research and Practice*, in *Abstract State Machines 2003*. 2003. p. 1-21.
72. S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. *Genetic Algorithms Applications to Software Testing*. in *Fifth International Conference. Software Engineering and Its Applications*. 1992. Toulouse, France.
73. G. J. Myers, *The art of Software Testing*. 1979, New York, NY, USA: John Wiley & Sons, Inc. 177.

74. R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns and Tools*. 1999: Addison-Wesley.
75. B. Beizer, *Software Testing Techniques*. 1990: Van Norstrand Reinhold.
76. *JUnit: http://www.junit.org/*.
77. *EasyMock: http://easymock.org/*.
78. *Selenium: http://seleniumhq.org/*.
79. *JWebUnit: http://jwebunit.sourceforge.net/*.
80. S. Bradbury Jeremy, R. Cordy James, and Dingel Juergen, *Mutation Operators for Concurrent Java (J2SE 5.0)*, in *Proceedings of the Second Workshop on Mutation Analysis*. 2006, IEEE Computer Society.
81. J. Changbing, Ch. Zhenyu, X. Baowen, and W. Ziyuan, *A New Mutation Analysis Method for Testing Java Exception Handling*, in *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*. 2009, IEEE Computer Society.
82. M. Yu-Seung, K. Yong-Rae, and J. Offutt, *Inter-Class Mutation Operators for Java*, in *Proceedings of the 13th International Symposium on Software Reliability Engineering*. 2002, IEEE Computer Society.
83. M. Ellims, D. Ince, and M. Petre, *The Csaw C Mutation Tool: Initial Results*, in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 2007, IEEE Computer Society.
84. A. Derezińska, *Advanced mutation operators applicable in C# programs*, in *Software Engineering Techniques: Design for Quality*. 2007. p. 283-288.
85. A. Derezinska, *Quality Assessment of Mutation Operators Dedicated for C\# Programs*, in *Proceedings of the Sixth International Conference on Quality Software*. 2006, IEEE Computer Society.
86. J. Offutt, *Investigations of the software testing coupling effect.* ACM Transactions on Software Engineering and Methodology, 1992. **1**(1): p. 5 - 20.
87. J. H. Andrews, L. C. Briand, and Y. Labiche, *Is mutation an appropriate tool for testing experiments?*, in *Proceedings of the 27th international conference on Software engineering*. 2005, ACM: St. Louis, MO, USA.
88. A. Jefferson Offutt, Pan Jie, Tewary Kanupriya, and Zhang Tong, *An experimental evaluation of data flow and mutation testing.* Softw. Pract. Exper., 1996. **26**(2): p. 165-176.
89. W. E. Wong, *On Mutation and Data Flow*. 1993.
90. A. P. Mathur and W. Eric Wong, *An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria.* Software Testing, Verification and Reliability, 1994. **4**: p. 9-31.
91. L. Nan, P. Upsorn, and J. Offut, *An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage*, in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*. 2009, IEEE Computer Society.
92. M. Yu-Seung, J. Offutt, and Y. R. Kwon, *MuJava : An Automated Class Mutation System.* Software Testing, Verification and Reliability, 2005.
93. *PlexTest: http://www.itregister.com.au/products/plextest.htm*.
94. *Heckle: http://glu.ttono.us/articles/2006/12/19/tormenting-your-tests-with-heckle*.
95. W. Westley, N. ThanhVu, C. Le Goues and S. Forrest, *Automatically finding patches using genetic programming*, in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. 2009, IEEE Computer Society.
96. H. H. Thompson, *Why security testing is hard.* IEEE Security & Privacy Magazine, 2003. **1**(4): p. 83 - 86.

97. C. Wysopal, L. Nelson, D. Zovi Dino and E. Dustin, *The Art of Software Security Testing: Identifying Software Security Flaws (Symantec Press)*. 2006: Addison-Wesley Professional.
98. G. McGraw and B. Potter, *Software Security Testing*. IEEE Security and Privacy, 2004. **2**(5): p. 81-85.
99. *OWASP security and testing tools: http://www.owasp.org/index.php/Phoenix/Tools*.
100. D. Wenliang and A. P. Mathur. *Testing for Software Vulnerability Using Environment Perturbation*. in *International Conference on Dependable Systems and Networks*. 2000.
101. A. Ghosh, T. O'Connor, and G. McGraw, *An automated approach for identifying potential vulnerabilities in software*, in *IEEE Symposium on Security and Privacy*. 1998.
102. *JBroFuzz http://www.owasp.org/index.php/Category:OWASP_JBroFuzz*.
103. *Sulley http://code.google.com/p/sulley/*.
104. P. Godefroid, *Random testing for security: blackbox vs. whitebox fuzzing*, in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 2007, ACM: Atlanta, Georgia.
105. P. Godefroid, A. Kiezun, and M. Y. Levin, *Grammar-based whitebox fuzzing*, in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. 2008, ACM: Tucson, AZ, USA.
106. P. Godefroid, M. Levin, and D. Molnar. *Automated Whitebox Fuzz Testing*. Available from: citeulike-article-id:4510843
http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf.
107. J. Offut, Y. Wu, X. Du, and H. Huang. *Bypass testing of Web applications*. in *15th International Symposium on Software Reliability Engineering ISSRE 2004* 2004.
108. J. Offutt, Q. Wang, and J. Ordille, *An Industrial Case Study of Bypass Testing on Web Applications*, in *Software Testing, Verification, and Validation, 2008 International Conference on*. 2008.
109. A. Kieyzun, P. J. Guo, J. Karthick, and M. D. Ernst, *Automatic creation of SQL Injection and cross-site scripting attacks*, in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. 2009, IEEE Computer Society.
110. 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.
111. A. Masood, A. Ghafoor, and A. Mathur, *Technical report: Scalable and Effective Test Generation for Access Control Systems that Employ RBAC Policies*. 2005.
112. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. *Verification and change-impact analysis of access-control policies*. in *ICSE*. 2005.
113. K. Li, L. Mounier, and R. Groz. *Test Generation from Security Policies Specified in Or-BAC*. in *31st Annual International Computer Software and Applications Conference. COMPSAC 2007.* . 2007.
114. W. Mallouli, J. M. Orset, A. Cavalli, N. Cuppens and Cuppens F. *A Formal Approach for Testing Security Rules*. in *SACMAT*. 2007.
115. 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.
116. W. Mallouli, J. M. Orset, A. Cavalli, N. Cuppens, and F. Cuppens, *A formal approach for testing security rules*, in *Proceedings of the 12th ACM symposium on Access control models and technologies*. 2007, ACM: Sophia Antipolis, France.

117. J. Jacques, P. A. Masson, and R. Tissot, *Generating security tests in addition to functional tests*, in *Proceedings of the 3rd international workshop on Automation of software test*. 2008, ACM: Leipzig, Germany.
118. D. P. Guelev, M. Ryan, and P. Y. Schobbens, *Model-Checking Access Control Policies*, in *Information Security*. 2004. p. 219-230.
119. N. Zhang, M. Ryan, and D. P. Guelev, *Evaluating Access Control Policies Through Model Checking*, in *Information Security*. 2005. p. 446-460.
120. J. Offutt, Q. Wang, and J. Ordille. *An Industrial Case Study of Bypass Testing on Web Applications*. in *Software Testing, Verification, and Validation, 2008 International Conference on*. 2008.
121. 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.
122. E. J. Coyne R. Sandhu, H. L. Feinstein, and C.E. Youman, *Role-based access control models.* IEEE Computer, 1996. **29**(2): p. 38-47.
123. R. Thomas and R. Sandhu, *Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management*, in *11 th IFIP Working Conference on Database Security*. 1997.
124. 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.
125. L. Briand and Y. Labiche, *A UML-based approach to System Testing.* Software and Systems Modeling, 2002. **1**(1): p. 10 - 42.
126. C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jézéquel, *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
127. F. Cuppens, N. Cuppens-Boulahia, and M. Ben Ghorbel. *High-level conflict management strategies in advanced access control models*. in *Workshop on Information and Computer Security (ICS'06)*. 2006.
128. J. Offutt, Ammei Lee, G. Rothermel, Roland H. Untch, and Christian Zapf, *An Experimental Determination of Sufficient Mutant Operators.* ACM Transactions on Software Engineering and Methodology, 1996. **5**(2): p. 99 - 118.
129. J. M. Voas, *PIE : A Dynamic Failure-Based Technique.* IEEE Transactions on Software Engineering, 1992. **18**(8): p. 717 - 727.
130. A. Williams and R. Probert, *A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces*, in *ISSRE*. 1996.
131. M. Grindal., J. Offutt, and S. Andler, *Combination Testing Strategies: A Survey. Technical Report ISE-TR-04-05*. 2004, George Mason University.
132. E. Martin and T. Xie. *Inferring Access-Control Policy Properties via Machine Learning.* in *7th IEEE Workshop on Policies for Distributed Systems and Networks* 2006.
133. P. T. Devanbu and S. Stubblebine. *Software engineering for security: a roadmap*. in *International Conference on Software Engineering*. 2000.
134. J. M. Jézéquel, M. Train, and C. Mingins, *Design Patterns and Contracts*. 1999: Addison-Wesley. 348.
135. D.E BELL and L.J. LaPADULA,
Secure Computer Systems: Unified Exposition and Multics Interpretation. 1976, The MITRE Corporation.
136. *Kermeta MDK* http://www.kermeta.org/mdk/.
137. Sun's XACML implementation. http://sunxacml.sourceforge.net/.
138. *XACML:* http://www.oasis-open.org/committees/xacml/.

139. P. A. Muller, F. Fleurey, and J. M. Jézéquel. *Weaving executability into object-oriented meta-languages*. in *MoDELS'05*. 2005. Montego Bay, Jamaica: LNCS.
140. OMG. *MOF 2.0 Core Final Adopted Specification.* 2004 2005; Available from: http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
141. *MotOrBAC: http://motorbac.sourceforge.net/index.php?page=home&lang=en*.
142. *The OSGi Alliance. OSGi Service Platform Core Specification. Release 4.1, May 2007 http://www.osgi.org/Specifications/*.
143. B. Morin, O. Barais, G. Nain and J.-M. Jezequel, *Taming Dynamically Adaptive Systems using models and aspects*, in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. 2009, IEEE Computer Society.
144. G. Wassermann and S. Zhendong, *Static detection of cross-site scripting vulnerabilities*, in *Proceedings of the 30th international conference on Software engineering*. 2008, ACM: Leipzig, Germany.
145. E. Ikpeme, *Browser-Based Intrusion Prevention System*, in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. 2009, Springer-Verlag: Saint-Malo, France.

List of figures

Figure 1 - Overview of the considered architecture 10
Figure 2 - Security levels 13
Figure 3 - Overview of application security 14
Figure 4 - Architecture of the Access control mechanisms 19
Figure 5 - SQLRand architecture 21
Figure 6 - Overview of the learning based approach 22
Figure 7 - Secure software development lifecycle 23
Figure 8 - Sequence diagram for an XSS attack 23
Figure 9 - RBAC template 25
Figure 10 - The V-Model software development process 27
Figure 11 - The process of mutation analysis 28
Figure 12 - Context for testing a secured system 35
Figure 13 - OrBAC entities for the LMS 38
Figure 14 - Qualifying security tests 41
Figure 15 - Testing the access control policy 41
Figure 16 - Mutation scores for functional tests 46
Figure 17 - Mutation scores for All-primary-rules and All-concrete-rules security tests 47
Figure 18 - Independent vs. Incremental strategies 49
Figure 19 - MO1 operator strictly subsumes MO2 50
Figure 20 - Testing mutants causing conflicts. 52
Figure 21 - Relation between mutation operators 53
Figure 22 - Overview of the three analysis steps 56
Figure 23 - First step: test cases selection 57
Figure 24 - Second step: dynamic analysis of test cases impact on security mechanisms 58
Figure 25 - Woven security oracle function 60
Figure 26 - The overall process 64
Figure 27 – The first generation strategy 64
Figure 28 – The second generation strategy 65
Figure 29 - The second generation strategy using different rule elements 66
Figure 30 - Meeting System (432 exhaustive tests) 68
Figure 31 - Library System (280 exhaustive tests) 69
Figure 32 - Hospital System (450 exhaustive tests) 69
Figure 33 - Auction System (736 exhaustive tests) 70
Figure 34 - An explicit and visible security mechanism 75
Figure 35 - Explicit hidden security mechanism 75
Figure 36 - Implicit security mechanism 76
Figure 37 - Decomposition of the SP evolution into micro steps 79
Figure 38 - Micro-evolutions of a legacy system access control policy 80
Figure 39 - Input validation model 87
Figure 40 - Constraint-based Testing and shielding web applications 88
Figure 41 - Collecting constraints using manual navigation and functional tests 90
Figure 42 - Collecting HTML constraints from the GUI 91
Figure 43 - The process for extracting JavaScript validation code 92
Figure 44 - Overview of the shield 94

Figure 45 - Example of a generated input 95
Figure 46 - Execution of requests 96
Figure 47 - The modeling framework 102
Figure 48 - Overview of the model-based approach 103
Figure 49 - The meta-model for rule-based security formalisms 105
Figure 50 - The OrBAC security formalism 105
Figure 51 - Rule R1 for the library access control policy 106
Figure 52 - RBAC model 106
Figure 53 - The DAC formalism 107
Figure 54 - The MAC formalism 108
Figure 55 - PEP and PDP 109
Figure 56 - The security framework class diagram 109
Figure 57 - MDE validation process 112
Figure 58 - Extension of the security metamodel with mutation operators 113
Figure 59 - The RER operator 114
Figure 60 - Examples of security and functional test 117
Figure 61 - Mutation scores per mutation operator and language 118
Figure 62 - Overview of the proposed approach 122
Figure 63 - Access Control Metamodel 124
Figure 64 - A Core Metamodel for Describing Runtime Architectures 124
Figure 65 - 3-Layered architecture reflecting access control policies 126
Figure 66 - Leveraging MDE to generate safe reconfiguration scripts 127
Figure 67 - Permissions, Delegations and Resources to Architectural Concepts 128
Figure 68 - The Permission meta-class aspectized with Kermeta 129
Figure 69 - Architecture after 3 new permissions have been added 130

List of tables

Table 1 - Retained Mutation operators for OrBAC rules 44
Table 2 - The size of the three case study applications 45
Table 3 - # Mutants per mutation type and operator 46
Table 4 - Number of tests cases by category 47
Table 5 - Overlap of CR2 and adv. test cases 48
Table 6 - Sizes of minimal test suites 51
Table 7 - Test suites overlap for PPR-PRP and ANR operators 51
Table 8 - # of generated mutants with and without conflicts 53
Table 9 - The three quality level of oracle 59
Table 10 - First step analysis results 60
Table 11 - Execution time saved with the first analysis 61
Table 12 - Examples of detailed results 61
Table 13 - Tests and impacting rule for VMS application 61
Table 14 - Automated and manual approaches 62
Table 15 - Strategy 1 and 2 vs. the random strategy 70
Table 16 - The percentage of generated test of strategy 2 71
Table 17 - # of flexible and rigid rules for VMS 82
Table 18 - # of flexible and rigid rules for each resource/view for VMS 82
Table 19 - Flexibility rate for each function/activity for VMS 82
Table 20 - # of flexible and rigid rules for ASMS 83
Table 21 - # of flexible and rigid rules for each resource/view for ASMS 83
Table 22 - Flexibility rate for each function/activity for ASMS 83
Table 23 - HTML predefined constraints 91
Table 24 – Predefined constraints 93
Table 25 - The way constraints are violated 95
Table 26 - The three categories of results 96
Table 27 - The four Web Applications 97
Table 28 – Bypass testing results 98
Table 29 - Performance results 98
Table 30 - Mapping between OrBAC entities and XACML elements 110
Table 31 - The mutation operators 113
Table 32 - Number of mutants generated for RBAC/OrBAC policies 117
Table 33 - specific mutants vs. Generic mutants 119
Table 34 - Relevance of generic verification for specific access control languages 119
Table 35 - OrBAC mutation results vs. Generic mutation results 120